Operational methods in semantics

Roberto Amadio

Operational methods in semantics

Roberto M. Amadio
Université Paris-Diderot

May 3, 2017

Contents

Preface

The focus of these lecture notes is on abstract models and basic ideas and results that relate to the *operational semantics* of programming languages largely conceived. The approach is to start with an abstract description of the computation steps of programs and then to build on top semantic equivalences, specification languages, and static analyses. While other approaches to the semantics of programming languages are possible, it appears that the operational one is particularly effective in that it requires a moderate level of mathematical sophistication and scales reasonably well to a large variety of programming features. In practice, operational semantics is a suitable framework to build portable language implementations and to specify and test program properties (see, *e.g.*, [MTH90]). It is also used routinely to tackle more ambitious tasks such as proving the correctness of a compiler or a static analyzer (see, *e.g.*, [Ler06]).

These lecture notes contain a selection of the material taught by the author over several years in courses on the semantics of programming languages, foundations of programming, compilation, and concurrency theory. They are oriented towards master students interested in fundamental research in computer science. The reader is supposed to be familiar with the main programming paradigms (imperative, functional, object-oriented,...) and to have been exposed to the notions of concurrency and synchronization as usually discussed in a course on operating systems. The reader is also expected to have attended introductory courses on automata, formal languages, mathematical logic, and compilation of programming languages.

Our goal is to provide a compact reference for grasping the basic ideas of a rapidly evolving field. This means that we concentrate on the simple cases and we give a self-contained presentation of the proof techniques. Following this approach, we manage to cover a rather large spectrum of topics within a coherent terminology and to shed some light, we hope, on the connections among apparently different formalisms.

Chapter 1 introduces, in the setting of a very simple *imperative programming language*, some of the main ideas and applications of operational semantics. A *sequential* programming language is a formalism to define a system of computable functions; the closer the formalism to the notion of function, the simpler the semantics. The first formalism we consider is the one of *term rewriting systems* (chapters 2–6). On one hand, (term) rewriting is ubiquitous in operational semantics and so it seems to be a good idea to set on solid foundations the notions of *termination* and *confluence*. On the other hand, under suitable conditions, term rewriting is a way of defining *first-order functions* on inductively defined data structures.

The second formalism we introduce (chapters 7–9) is the λ-*calculus*, which is a notation to represent higher-order functions. In this setting, a function is itself a datum that can be passed as an argument or returned as a result. We spend some time to explain the mechanisms needed for correctly implementing the λ-calculus via the notion of *closure*. We then address the issue of program equivalence (or refinement) and claim that the notion of *contextual equivalence*

provides a natural answer to this issue. We also show that the co-inductively defined notion of *simulation* provides an effective method to reason about contextual equivalence.

Chapters 10–13 introduce increasingly expressive *type systems* for the λ-calculus. The general idea is that types express properties of program expressions which are invariant under execution. As such, types are a way of documenting the way a program expression can be used and by combining program expressions according to their types we can avoid many run-time errors. In their purest form, types can be connected with logical propositions and this leads to a fruitful interaction with a branch of mathematical logic known as *proof theory*. Sometimes types lead to verbose programs. To address this issue we introduce *type inference* techniques which are automatic methods discharging the programmer from the task of explicitly writing the types of the program expressions. Also sometimes types are a bit of a straight jacket in that they limit the way programs can be combined or reused. We shall see that *polymorphic* types (and later *subtyping*) address, to some extent, these issues.

Chapters 14–15, introduce various standard *program transformations* that chained together allow to compile a higher-order (functional) language into a basic assembly language. We also show that the type systems presented in the previous chapters shed light on the program transformations.

Chapters 16–18 consider the problem of formalizing the operational semantics of *imperative* and *object-oriented* programming languages. We show that the notion of higher-order computable function is still useful to understand the behavior of programs written in these languages. We start by enriching the functional languages considered with record and variant data types. This is an opportunity to discuss the notion of *subtyping* which is another way of making a type system more flexible. Concerning functions with *side-effects*, we show that they can be compiled to ordinary functions by expliciting the fact that each computation takes a memory as argument and returns a new memory as result. Concerning objects, we show that they can be understood as a kind of recursively defined records.

Starting from chapter 19, we move from *sequential* to *concurrent* programming models where several threads/processes *compete* for the same resources (e.g. write a variable or a channel). Most of the time, this results into *non-deterministic* behavior which means that with the same input the system can move to several (incomparable) states. Chapters 20–23 focus on a concurrent extension of the simple model of imperative programming introduced in chapter 1. In particular, we introduce a compositional *trace semantics*, *rely-guarantee* assertions, and mechanisms to implement *atomic* execution.

Chapters 24–27 take a more abstract look at concurrency in the framework of *labelled transition systems*. We develop the notion of *bisimulation* and we consider its logical characterization through a suitable *modal logic*. Labelled transition systems extended with a *rendez-vous* synchronization mechanism lead to a simple calculus of concurrent systems known as *CCS*. We rely on this calculus to explore the connections between *determinacy* and *confluence*.

Chapters 28 and 29 describe two relevant extensions of *CCS*. In the first one, we consider the notion of *timed (or synchronous)* execution where processes proceed in lockstep (at the same speed) and the computation is regulated by a notion of instant. In the second one, we consider systems which exhibit both non-deterministic and *probabilistic* behaviors.

Chapters 30–31 introduce another extension of *CCS*, known as π*-calculus*, where processes can communicate channel names. We show that the theory of equivalence developed for *CCS* can be lifted to the π-calculus and that the π-calculus can be regarded as a concurrent extension of the λ-calculus.

Finally, chapter 32 builds on chapter 18 to formalize a fragment of the concurrency avail-

able in the *Java* programming language and to discuss the notion of *linearization* of concurrent data structures.

While the choice of the topics is no doubt biased by the interests of the author, it still provides a fair representation of the possibilities offered by *operational semantics*. Links between operational semantics and more 'mathematical' semantics based, *e.g.*, on domain and/or category theory are not developed at all in these lecture notes; we refer the interested reader to, *e.g.*, [AC98, Gun92, Win93].

Most topics discussed in these lecture notes can form the basis of interesting *programming experiences* such as the construction of a compiler, a static type analyzer, or a verification condition generator. The *proofs* sketched in these lecture notes can also become the object of a programming experience in the sense that they can be formalized and checked in suitable *proof assistants* (experiments in this direction can be found, *e.g.*, in the books [Chl13, NK14, PCG+15]).

Each chapter ends with a summary of the main concepts and results introduced and a few bibliographic references. These references are suggestions for further study possibly leading to research problems. Quite often we prefer to quote the 'classic' papers that introduced a concept than the most recent ones which elaborated on it. Reading the 'classics' is a very valuable exercise which helps in building some historical perspective especially in a discipline like computer science where history is so short.

These lecture notes contain enough material for a two semesters course; however, there are many possible shortcuts to fit just one semester course. The chapters 1–18 cover *sequential languages*. Chapters 1, 2, and some of 4 are a recommended introduction and the chapters 7–10 constitute the backbone on the λ-calculus. The remaining chapters can be selected according to the taste of the instructor and the interests of the students. Topics covered include: term rewriting systems (chapters 3, 5, 6), type systems (chapters 12, 13, 16), type inference (chapters 3, 11, 12), program transformations (chapters 14, 15), and imperative and object-oriented languages (chapters 17, 18). The chapters 19–32 focus on *concurrent languages* and assume some familiarity with the basic material mentioned above. Chapter 19 is a recommended introduction to concurrency. Chapters 20–23 cover a simple model of shared memory concurrency while chapters 24–26 lead to the calculus *CCS*, a basic model of message passing concurrency. The following chapters explore the notions of deterministic (chapter 27), timed (chapter 28), and probabilistic (chapter 29) computation. The final chapters move towards models of concurrency that integrate the complexity of a sequential language. In particular, we discuss the π-calculus (chapters 30, 31) which relates to the λ-calculus and a concurrent object oriented language (chapter 32) which extends the object-oriented language presented in chapter 18.

Notation

Set theoretical

$\emptyset$	empty set
$\mathbf{N}$	natural numbers
$\mathbf{Z}$	integers
$\cup, \cap$	union, intersection of two sets
$\bigcup, \bigcap$	union, intersection of a family of sets
X^c	complement of X
$\mathcal{P}(X)$	subsets of X
$\mathcal{P}_{fin}(X)$	finite subsets of X
$\sharp X$	cardinality of X
R^*	reflexive and transitive closure of R

Order theoretical

$(P, <)$	transitive partial order
$(P, \leq)$	reflexive and transitive partial order
$f : (P, \leq) \to (P', \leq')$	f is monotonic if it preserves the preorder
$\bigvee X$	least upper bound (lub)
$\bigwedge X$	greatest lower bound (glb)

Syntax

We introduce a number of operators that *bound* variables. The rules for renaming bound variables and for substituting a term in a term with bound variables are the same that apply in, say, first-order logic. If $T, S, \ldots$ are terms and x is a variable, we denote with $\mathsf{fv}(T)$ the set of variables occurring free in T and with $[T/x]S$ the substitution of T for x in S.

Semantics

$$f[e/d] \quad \text{function update, } f[e/d](x) = \begin{cases} e & \text{if } x = d \\ f(x) & \text{otherwise} \end{cases}$$

Chapter 1

Introduction to operational semantics

The goal of this introductory chapter is to present at an elementary level some ideas of the operational approach to the semantics of programming languages and to illustrate some of their applications.

To this end, we shall focus on a standard toy imperative language called Imp. As a first step we describe formally and at an abstract level the computations of Imp programs. In doing this, we identify two styles known as *big-step* and *small-step*. Then, based on this specification, we introduce a suitable notion of pre-order on statements and check that this pre-order is preserved by the operators of the Imp language.

As a second step, we introduce a specification formalism for Imp programs that relies on so called *partial correctness assertions* (pca's). We present sound rules for reasoning about such assertions and a structured methodology to reduce reasoning about pca's to ordinary reasoning in a suitable theory of (first-order) logic. We also show that the pre-order previously defined on statements coincides with the one induced by pca's.

As a third and final step, we specify a toy compiler from the Imp language to an hypothetical *virtual machine* whose semantics is also defined using operational techniques. We then apply the developed framework to prove the correctness of the compiler.

1.1 A simple imperative language

We assume the reader is familiar with the idea that the *syntax* of a programming language can be specified via a context-free grammar. The syntax of the Imp language is described in Table 1.1 where we distinguish the syntactic categories of identifiers (or variables), integers, values, numerical expressions, boolean conditions, statements, and programs. We shall not dwell on questions of grammar ambiguity and priority of operators. Whenever we look at a syntactic expression we assume it contains enough parentheses so that no ambiguity arises on the order of application of the operators.

We also assume the reader is familiar with the notion of *formal system*. A formal system is composed of *formulae* specified by a certain syntax and *inference rules* to derive formulae from other formulae. Depending on the context, the formulae may be called assertions or

judgments. We often rely on the following suggestive notation to describe inference rules:

$$\frac{A_1, \ldots, A_n}{B} \ ,$$

which means that if we can infer formulae $A_1, \ldots, A_n$ (the hypotheses) then we can also infer formula B (the conclusion). To bootstrap the inference process we need some rule with no hypothesis, *i.e.*, where $n = 0$. Such rules are called *axioms*. A rule with m conclusions is regarded as an abbreviation for m rules which share the same hypotheses:

$$\frac{A_1, \ldots, A_n}{B_1, \ldots, B_m} \quad \text{is equivalent to} \quad \frac{A_1, \ldots, A_n}{B_1}, \cdots, \frac{A_1, \ldots, A_n}{B_m} \ .$$

The Imp language is a rather standard *imperative language* with while loops and if-then-else. We call the language *imperative* because the execution of a program is understood as the execution of a sequence of statements whose effect is to modify a global entity known as the *state*. We can regard the *state* as an abstract model of the computer's memory. As in every modeling activity, the name of the game is to have a simple model but not too simple. In other words, the model should not contain too many details and still be able to make useful predictions on programs' behaviors. For the Imp language, we shall assume the state is a total function from identifiers to integers. Notice that by representing the state as a *total* function we avoid some technicalities, namely we make sure that the evaluation of a variable in a state is always defined.

If s is a state, x an identifier, and n an integer, then we denote with $s[n/x]$ an elementary *state update* defined as follows:

$$s[n/x](y) = \begin{cases} n & \text{if } x = y \\ s(y) & \text{otherwise.} \end{cases}$$

A first approach at specifying the execution of Imp programs relies on the following judgments (or assertions):

$$(e, s) \Downarrow v, \quad (b, s) \Downarrow v, \qquad (S, s) \Downarrow s', \quad (P, s) \Downarrow s',$$

and it is described in Table 1.2. The defined predicates $\Downarrow$ are often called *evaluations*. They specify the final result of the execution (if any) while neglecting the intermediate steps. Thus, for a given state, a (boolean) expression evaluates to a value while a statement (or a program) evaluates to a state. This specification style is called *big-step*.

By opposition, the *small-step* approach is based on the definition of 'elementary' reduction rules. The final result, if any, is obtained by iteration of the reduction rules. In order to

id	$::= x \mid y \mid \ldots$	(identifiers)
n	$::= 0 \mid -1 \mid +1 \mid \ldots$	(integers)
v	$::= n \mid \mathsf{true} \mid \mathsf{false}$	(values)
e	$::= id \mid n \mid e + e$	(numerical expressions)
b	$::= e < e$	(boolean conditions)
S	$::= \mathsf{skip} \mid id := e \mid S; S \mid \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S \mid \mathsf{while}\ b\ \mathsf{do}\ S$	(statements)
P	$::= \mathsf{prog}\ S$	(programs)

Table 1.1: Syntax of the Imp language

$$\frac{}{(v,s)\Downarrow v} \qquad \frac{}{(x,s)\Downarrow s(x)} \qquad \frac{(e,s)\Downarrow v \quad (e',s)\Downarrow v'}{(e+e',s)\Downarrow (v +_{\mathbf{Z}} v')} \qquad \frac{(e,s)\Downarrow v \quad (e',s)\Downarrow v'}{(e<e',s)\Downarrow (v <_{\mathbf{Z}} v')}$$

$$\frac{}{(\mathsf{skip},s)\Downarrow s} \qquad \frac{(e,s)\Downarrow v}{(x:=e,s)\Downarrow s[v/x]} \qquad \frac{(S_1,s)\Downarrow s' \quad (S_2,s')\Downarrow s''}{(S_1;S_2,s)\Downarrow s''}$$

$$\frac{(b,s)\Downarrow \mathsf{true} \quad (S,s)\Downarrow s'}{(\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S',s)\Downarrow s'} \qquad \frac{(b,s)\Downarrow \mathsf{false} \quad (S',s)\Downarrow s'}{(\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S',s)\Downarrow s'}$$

$$\frac{(b,s)\Downarrow \mathsf{false}}{(\mathsf{while}\ b\ \mathsf{do}\ S,s)\Downarrow s} \qquad \frac{(b,s)\Downarrow \mathsf{true} \quad (S;\mathsf{while}\ b\ \mathsf{do}\ S,s)\Downarrow s'}{(\mathsf{while}\ b\ \mathsf{do}\ S,s)\Downarrow s'}$$

$$\frac{(S,s)\Downarrow s'}{(\mathsf{prog}\ S,s)\Downarrow s'}$$

Table 1.2: Big-step reduction rules of Imp

$$\begin{array}{lcl}
(x:=e,K,s) & \to & (\mathsf{skip},K,s[v/x]) \qquad \text{if } (e,s)\Downarrow v \\
(S;S',K,s) & \to & (S,S'\cdot K,s) \\
(\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S',K,s) & \to & \begin{cases} (S,K,s) & \text{if } (b,s)\Downarrow \mathsf{true} \\ (S',K,s) & \text{if } (b,s)\Downarrow \mathsf{false} \end{cases} \\
(\mathsf{while}\ b\ \mathsf{do}\ S,K,s) & \to & \begin{cases} (S,(\mathsf{while}\ b\ \mathsf{do}\ S)\cdot K,s) & \text{if } (b,s)\Downarrow \mathsf{true} \\ (\mathsf{skip},K,s) & \text{if } (b,s)\Downarrow \mathsf{false} \end{cases} \\
(\mathsf{skip},S\cdot K,s) & \to & (S,K,s)
\end{array}$$

Table 1.3: Small-step reduction rules of Imp statements

describe the intermediate steps of the computation, we introduce an additional syntactic category of continuations. A *continuation* K is a list of statements which terminates with a special symbol halt:

$$K ::= \mathsf{halt} \mid S\cdot K \qquad \text{(continuation)}.$$

A continuation keeps track of the statements that still need to be executed. Table 1.3 defines small-step reduction rules for Imp statements whose basic judgment has the shape:

$$(S,K,s)\to(S',K',s')\ .$$

Note that we still rely on the big-step reduction of (boolean) expressions; the definition of a small step reduction for (boolean) expressions is left to the reader. We define the reduction of a program prog S as the reduction of the statement S with continuation halt. We can derive a big-step reduction from the small-step one as follows:

$$(S,s)\Downarrow s' \quad \text{if } (S,\mathsf{halt},s)\xrightarrow{*}(\mathsf{skip},\mathsf{halt},s')\ ,$$

where $\xrightarrow{*}$ denotes the reflexive and transitive closure of the relation $\to$.

Let us pause to consider some properties of both the big-step and the small-step reductions. In both cases, reduction is driven by the syntax of the object (program, statement,...) under

consideration. Moreover it is easy to check that for each state and program (or statement, or expression, or boolean expression) at most one rule applies. This entails that the computation is *deterministic*. In some situations the computation is stuck, *i.e.*, no rule applies. This happens if we try to add or compare two expressions whose values are *not* integers. Also in some situations the computation diverges and this happens because of the unfolding of the while loop.

To summarize, given a program and a state, 3 mutually exclusive situations may arise: (1) the computation terminates producing a new state, (2) the computation is stuck in a situation where no rule applies, and (3) the computation diverges. Because, our computation rules are *deterministic*, in situation (1) there is exactly one state which is the outcome of the computation. Situation (2) corresponds to an erroneous configuration. Rather than leaving the computation stuck it is always possible to add rules and values so that the computation actually terminates returning some significant *error message*. As for situation (3), in the big-step approach, it arises as an infinite regression in the proof tree. For instance, assuming $S = \mathsf{while\ true\ do\ skip}$, we have:

$$\dfrac{\dfrac{(\mathsf{skip}, s) \Downarrow s \quad (S, s) \Downarrow ?}{(\mathsf{skip}; S, s) \Downarrow ?}}{(S, s) \Downarrow ?}$$

In the small-step approach, a diverging computation is an infinite reduction as in:

$$(S, \mathsf{halt}, s) \to (\mathsf{skip}, S \cdot \mathsf{halt}, s) \to (S, \mathsf{halt}, s) \to \cdots$$

Specifying the way programs compute is actually only the first step in the definition of an operational semantics. The second step consists in defining a notion of *program equivalence*. To answer this question, we need to decide what exactly is *observable* in the computation of a program. In general, *sequential programs* are regarded as functions that transform input data into output data. In particular, sequential *imperative* programs such as those of the Imp language can be interpreted as partial functions from states to states. Following this idea, we also interpret statements as partial functions from states to states and (boolean) expressions as total functions from states to numerical (boolean) values.

Definition 1 (IO interpretation) *The IO interpretation of* Imp *programs, statements, and (boolean) expressions is defined as follows:*

$$\begin{array}{llll} [\![P]\!]^{IO} & = \{(s, s') \mid (P, s) \Downarrow s'\}, & [\![S]\!]^{IO} & = \{(s, s') \mid (S, s) \Downarrow s'\}, \\ [\![b]\!]^{IO} & = \{(s, v) \mid (b, s) \Downarrow v\}, & [\![e]\!]^{IO} & = \{(s, v) \mid (e, s) \Downarrow v\}\ . \end{array}$$

A third step consists in checking the *compositionality properties* of the proposed interpretation. For instance, suppose we have shown that the IO-interpretations of two statements S and S' coincide. Does this guarantee that we can always replace any occurrence of the statement S in a program with the statement S' without affecting the overall behavior of the program? To make this idea precise we introduce the notion of *statement context*.

Definition 2 (context) *A* statement context *(or* context *for short) C is defined by the following grammar:*

$$C ::= [\ ] \mid C; S \mid S; C \mid \mathsf{if}\ b\ \mathsf{then}\ C\ \mathsf{else}\ S \mid \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ C \mid \mathsf{while}\ b\ \mathsf{do}\ C$$

where $[\ ]$ is a fresh symbol which stands for a placeholder (or a hole).

If C is a context and S a statement then $C[S]$ is the statement resulting from replacing the special symbol [ ] with S in C. For instance, if $C = S'; [\,]$ then $C[S] = S'; S$.

Proposition 3 (compositionality) *For all statements S and S' and context C, if $[\![S]\!]^{IO} \subseteq [\![S']\!]^{IO}$ then $[\![C[S]]\!]^{IO} \subseteq [\![C[S']]\!]^{IO}$.*

Proof. We proceed by induction on the height of the proof of the judgment $(C[S], s) \Downarrow s'$ and case analysis on the shape of the context C. For instance, suppose $C = \mathsf{while}\ b\ \mathsf{do}\ C'$. We distinguish two cases.

- If $(b, s) \Downarrow \mathsf{false}$ then $s' = s$ and $(C[S'], s) \Downarrow s$.
- If $(b, s) \Downarrow \mathsf{true}$, $(C'[S], s) \Downarrow s''$, and $(C[S], s'') \Downarrow s'$. Then, by inductive hypothesis, we have that: $(C'[S'], s) \Downarrow s''$ and $(C[S'], s'') \Downarrow s'$. Hence $(C[S'], s) \Downarrow s'$. □

Exercise 4 *Implement in your favorite programming language the big-step and small-step reduction rules (Tables 1.2 and 1.3) of the* Imp *language.*

Exercise 5 *Suppose we extend the* Imp *language with the commands* break *and* continue*. Their informal semantics is as follows:*

break *causes execution of the smallest enclosing* while *statement to be terminated. Program control is immediately transferred to the point just beyond the terminated statement. It is an error for a* break *statement to appear where there is no enclosing* while *statement.*

continue *causes execution of the smallest enclosing* while *statement to be terminated. Program control is immediately transferred to the end of the body, and the execution of the affected* while *statement continues from that point with a reevaluation of the loop test. It is an error for* continue *to appear where there is no enclosing* while *statement.*

Define the big-step and small-step reduction rules for the extended language. Hint: *for the big-step, consider extended judgments of the shape $(S, s) \Downarrow (o, s')$ where o is an additional information indicating the* mode *of the result, for the small-step consider a new continuation* endloop(K)*, where K is an arbitrary continuation.*

1.2 Partial correctness assertions

Most programming languages support the insertion of logical assertions in the control flow. At run time, whenever a logical assertion is crossed its validity is checked and an exception is raised if the check fails. Inserting assertions in programs is an excellent way of documenting the expectations on the input (pre-conditions) and the guarantees on the output (post-conditions). Moreover, assertions are quite helpful in nailing down bugs. In the following, we consider systematic methods to compose pre and post conditions and possibly prove for a given statement and pre-condition that a certain post-condition will always hold. We denote with $A, B, \ldots$ *assertions*. When we regard them as *syntax* they are formulae with variables ranging over the set program variables. For instance:

$$\exists\, y\ \ (x = 3 \wedge z > y > x)\ . \tag{1.1}$$

We write $s \models A$ if the assertion A holds in the interpretation (state) s. Thus a syntactic assertion such as (1.1) is *semantically* the set of states that satisfy it, namely:

$$\{s \mid s(x) = 3, s(z) \geq 5\} \ .$$

Definition 6 (pca) *A* partial correctness assertion *(pca) is a triple* $\{A\}\ S\ \{B\}$. *We say that it is* valid *and write* $\models \{A\}\ S\ \{B\}$ *if:*

$$\forall\, s\ \ (s \models A \text{ and } (P, s) \Downarrow s' \text{ implies } s' \models B) \ .$$

The assertion is *partial* because it puts no constraint on the behavior of a non-terminating, *i.e.*, partial, statement. Table 1.4 describes the so called Floyd-Hoare rules (logic). The rules are formulated assuming that $A, B, \ldots$ are *sets of states*. It is possible to go one step further and replace the sets by predicates in, say, first-order logic, however this is not essential to understand the essence of the rules.

We recall that if S is a statement then $[\![S]\!]^{IO}$ is its input-output interpretation (definition 1). This is a binary relation on states which for Imp statements happens to be the graph of a *partial* function on states. In particular, notice that for an assignment $x := e$ we have:

$$[\![x := e]\!]^{IO} = \{(s, s[v/x]) \mid (e, s) \Downarrow v\} \ ,$$

which is the graph of a *total* function. In the assertions, we identify a boolean predicate b with the set of states that satisfy it, thus b stands for $\{s \mid s \models b\}$. We denote with $A, B, \ldots$ unary relations (predicates) on the set of states and with $R, S, \ldots$ binary relations on the set of states. We combine unary and binary relations as follows:

$$\begin{array}{ll} A; R & = \{s' \mid \exists\, s \ \ s \in A \text{ and } (s, s') \in R\} \quad \text{(image)} \\ R; A & = \{s \mid \exists\, s' \ \ s' \in A \text{ and } (s, s') \in R\} \quad \text{(pre-image).} \end{array}$$

The first rule in Table 1.4 allows to weaken the pre-condition and strengthen the post-condition while the following rules are associated with the operators of the language. The rules are *sound* in the sense that if the hypotheses are valid then the conclusion is valid too.

Proposition 7 (soundness pca rules) *The assertions derived in the system described in Table 1.4 are valid.*

$$\frac{A \subseteq A' \quad \{A'\}\ S\ \{B'\} \quad B' \subseteq B}{\{A\}\ S\ \{B\}} \qquad \frac{\{A\}\ S_1\ \{C\} \quad \{C\}\ S_2\ \{B\}}{\{A\}\ S_1; S_2\ \{B\}}$$

$$\frac{\{A \cap b\}\ S_1\ \{B\} \quad \{A \cap \neg b\}\ S_2\ \{B\}}{\{A\}\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \{B\}} \qquad \frac{A \subseteq B}{\{A\}\ \mathsf{skip}\ \{B\}}$$

$$\frac{A; [\![x := e]\!]^{IO} \subseteq B}{\{A\}\ x := e\ \{B\}} \qquad \frac{(A \cap \neg b) \subseteq B \quad \{A \cap b\}\ S\ \{A\}}{\{A\}\ \mathsf{while}\ b\ \mathsf{do}\ S\ \{B\}}$$

Table 1.4: Floyd-Hoare rules for Imp

PROOF. We just look at the case for the while rule. Suppose $s \in A$ and $(\mathsf{while}\ b\ \mathsf{do}\ S, s) \Downarrow s'$. We show by induction on the height of the derivation that $s' \in B$. For the basic case, we have $s \in \neg b$ and we know $A \cap \neg b \subseteq B$. On the other hand, suppose $s \in b$ and

$$(S; \mathsf{while}\ b\ \mathsf{do}\ S, s) \Downarrow s' \ .$$

This means $(S, s) \Downarrow s''$ and $(\mathsf{while}\ b\ \mathsf{do}\ S, s'') \Downarrow s'$. By hypothesis, we know $s'' \in A$ and by inductive hypothesis $s' \in B$. □

Exercise 8 *Suppose A is a first-order formula. Show the validity of the pca $\{[e/x]A\}\ x := e\ \{A\}$. On the other hand, show that the pca $\{A\}\ x := e\ \{[e/x]A\}$ is* not *valid.*

Interestingly, one can read the rules *bottom up* and show that if the conclusion is valid then the hypotheses are valid up to an application of the first 'logical' rule. This allows to reduce the proof of a pca $\{A\}\ S\ \{B\}$ to the proof of a purely set-theoretic/logical statement. The task of traversing the program S and producing logical assertions can be completely automated once the loops are annotated with suitable invariants. This is the job of so-called *verification condition generators*.

Proposition 9 (inversion pca rules) *The following properties hold:*

1. *If $\{A\}\ S_1; S_2\ \{B\}$ is valid then $\{A\}\ S_1\ \{C\}$ and $\{C\}\ S_2\ \{B\}$ are valid where $C = (A; [\![S_1]\!]^{IO}) \cap ([\![S_2]\!]^{IO}; B)$.*
2. *If $\{A\}\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \{B\}$ is valid then $\{A \cap b\}\ S_1\ \{B\}$ and $\{A \cap \neg b\}\ S_2\ \{B\}$ are valid.*
3. *If $\{A\}\ \mathsf{skip}\ \{B\}$ is valid then $A \subseteq B$ holds.*
4. *If $\{A\}\ x := e\ \{B\}$ is valid then $A; [\![x := e]\!]^{IO} \subseteq B$ holds.*
5. *If $\{A\}\ \mathsf{while}\ b\ \mathsf{do}\ S\ \{B\}$ is valid then there is $A' \supseteq A$ such that (i) $A' \cap \neg b \subseteq B$ and (ii) $\{A' \cap b\}\ S\ \{A'\}$ is valid.*

PROOF. The case for while is the interesting one. We define:

$$A_0 = A \quad A_{n+1} = (A_n \cap b); [\![S]\!]^{IO} \quad A' = \textstyle\bigcup_{n \geq 0} A_n \ .$$

We must have: $\forall\, n \geq 0\ \ A_n \cap \neg b \subseteq B$. Then for the first condition, we notice that:

$$\begin{array}{ll} A' \cap \neg b & = (\bigcup_{n \geq 0} A_n) \cap \neg b \quad = \bigcup_{n \geq 0} (A_n \cap \neg b) \\ & \subseteq \bigcup_{n \geq 0} B \qquad\qquad\ = B \ . \end{array}$$

For the second, we have:

$$\begin{array}{lll} (A' \cap b); [\![S]\!]^{IO} & = (\bigcup_{n \geq 0} A_n \cap b); [\![S]\!]^{IO} & = (\bigcup_{n \geq 0} (A_n \cap b)); [\![S]\!]^{IO} \\ & = \bigcup_{n \geq 0} (A_n \cap b); [\![S]\!]^{IO} & = \bigcup_{n \geq 1} A_n \\ & \subseteq \bigcup_{n \geq 0} A_n & = A' \ . \end{array}$$

An assertion such as A' is called an *invariant* of the loop. In the proof, A' is defined as the limit of an iterative process where at each step we run the body of the loop. While in

theory A' does the job, in practice it may be hard to reason on its properties; finding a *usable* invariant may require some creativity. □

Given a specification language on, say, statements, we can consider two statements *logically* equivalent if they satisfy exactly the same specifications. We can apply this idea to pca's.

Definition 10 (pca interpretation) *The pca interpretation of a process P is:*

$$[\![P]\!]^{pca} = \{(A, B) \mid \models \{A\}\ P\ \{B\}\}\ .$$

So now we have two possible notions of equivalence for statements: one based on the input-output behavior and another based on partial correctness assertions. However, it is not too difficult to show that they coincide.

Proposition 11 (IO vs. pca) *Let S_1, S_2 be statements. Then:*

$$[\![S_1]\!]^{IO} = [\![S_2]\!]^{IO} \quad \textit{iff} \quad [\![S_1]\!]^{pca} = [\![S_2]\!]^{pca}\ .$$

Proof. ($\Rightarrow$) Suppose $(A, B) \in [\![S_1]\!]^{pca}$, $s \models A$, $(S_2, s) \Downarrow s'$. Then $(s, s') \in [\![S_2]\!]^{IO} = [\![S_1]\!]^{IO}$. Hence $s' \models B$ and $(A, B) \in [\![S_2]\!]^{pca}$.

($\Leftarrow$) First a remark. Let us write $s =_X s'$ if $\forall\, x \in X\ s(x) = s'(x)$. Further suppose $X \supseteq \mathsf{fv}(S)$ and $(S, s) \Downarrow s'$. Then:

1. The variables outside X are untouched: $s =_{X^c} s'$.
2. If $s =_X s_1$ then $(S, s_1) \Downarrow s_1'$ and $s' =_X s_1'$.

We now move to the proof. Given a state and a finite set of variables X, define:

$$IS(s, X) = \bigwedge_{x \in X} (x = s(x))\ .$$

Notice that: $s' \models IS(s, X)$ iff $s' =_X s$. We proceed by contradiction, assuming $(s, s') \in [\![S_1]\!]^{IO}$ and $(s, s') \notin [\![S_2]\!]^{IO}$. Let X be the collection of variables occurring in the commands S_1 or S_2. Then check that:

$$(IS(s, X), \neg IS(s', X)) \in [\![S_2]\!]^{pca}\ .$$

On the other hand: $(IS(s, X), \neg IS(s', X)) \notin [\![S_1]\!]^{pca}$. □

Exercise 12 *Let S be a statement and B an assertion. The* weakest precondition *of S with respect to B is a predicate that we denote with $wp(S, B)$ such that: (i) $\{wp(S, B)\}\ S\ \{B\}$ is valid and (ii) if $\{A\}\ S\ \{B\}$ is valid then $A \subseteq wp(S, B)$. Let us assume the statement S does* not *contain* while *loops. Propose a strategy to compute $wp(S, B)$ and derive a method to reduce the validity of a pca $\{A\}\ S\ \{B\}$ to the validity of a logical assertion.*

Rule	$C[i] =$
$C \vdash (i, \sigma, s) \to (i+1, n \cdot \sigma, s)$	$\mathsf{cnst}(n)$
$C \vdash (i, \sigma, s) \to (i+1, s(x) \cdot \sigma, s)$	$\mathsf{var}(x)$
$C \vdash (i, n \cdot \sigma, s) \to (i+1, \sigma, s[n/x])$	$\mathsf{setvar}(x)$
$C \vdash (i, n \cdot n' \cdot \sigma, s) \to (i+1, (n +_{\mathbf{Z}} n') \cdot \sigma, s)$	add
$C \vdash (i, \sigma, s) \to (i+k+1, \sigma, s)$	$\mathsf{branch}(k)$
$C \vdash (i, n \cdot n' \cdot \sigma, s) \to (i+1, \sigma, s)$	$\mathsf{bge}(k)$ and $n <_{\mathbf{Z}} n'$
$C \vdash (i, n \cdot n' \cdot \sigma, s) \to (i+k+1, \sigma, s)$	$\mathsf{bge}(k)$ and $n \geq_{\mathbf{Z}} n'$

Table 1.5: Small-step reduction rules of Vm programs

1.3 A toy compiler

We introduce a simple virtual machine Vm to execute Imp programs. The machine includes the following elements: (1) a fixed code C (a possibly empty sequence of instructions), (2) a program counter *pc*, (3) a state s (identical to the one of Imp programs), (4) a stack of integers σ which, intuitively, is used to evaluate boolean and numerical expressions. The machine includes the following instructions with the associated informal semantics where 'push' and 'pop' act on the stack:

$\mathsf{cnst}(n)$	push n
$\mathsf{var}(x)$	push value x
$\mathsf{setvar}(x)$	pop value and assign it to x
add	pop 2 values and push their sum
$\mathsf{branch}(k)$	jump with offset k
$\mathsf{bge}(k)$	pop 2 values and jump if greater or equal with offset k
`halt`	stop computation

In the branching instructions, k is an integer that has to be added to the current program counter in order to determine the following instruction to be executed. Given a sequence C, we denote with $|C|$ its length and with $C[i]$ its i^{th} element (the leftmost element being the 0^{th} element). The (small-step) reduction rules of the instructions are formalized by rules of the shape:

$$C \vdash (i, \sigma, s) \to (j, \sigma', s') \ ,$$

and are fully described in Table 1.5. As already mentioned, the Imp and Vm reduction rules share the same notion of state. We write, *e.g.*, $n \cdot \sigma$ to stress that the top element of the stack exists and is n. We denote with ϵ an empty stack or an empty sequence of Vm instructions. We write $(C, s) \Downarrow s'$ if $C \vdash (0, \epsilon, s) \xrightarrow{*} (i, \epsilon, s')$ and $C[i] = \mathsf{halt}$.

In Table 1.6, we define compilation functions $\mathcal{C}$ from Imp to Vm which operate on expressions, boolean conditions, statements, and programs. We write $sz(e)$, $sz(b)$, $sz(S)$ for the number of instructions the compilation function associates with the expression e, the boolean condition b, and the statement S, respectively. For instance, the statement $\mathsf{while}\ (0 < 1)\ \mathsf{do\ skip}$ is compiled as:

$$(\mathsf{cnst}(0))(\mathsf{cnst}(1))(\mathsf{bge}(1))(\mathsf{branch}(-4)) \ .$$

We now consider the question of proving the 'correctness' of the compilation function. The following proposition relates the big-step reduction of Imp programs to the execution of the compiled code.

$$\mathcal{C}(x) = \mathtt{var}(x) \qquad \mathcal{C}(n) = \mathtt{cnst}(n) \qquad \mathcal{C}(e + e') = \mathcal{C}(e) \cdot \mathcal{C}(e') \cdot \mathtt{add}$$

$$\mathcal{C}(e < e', k) = \mathcal{C}(e) \cdot \mathcal{C}(e') \cdot \mathsf{bge}(k)$$

$$\mathcal{C}(\mathsf{skip}) = \epsilon \qquad \mathcal{C}(x := e) = \mathcal{C}(e) \cdot \mathtt{setvar(x)} \qquad \mathcal{C}(S; S') = \mathcal{C}(S) \cdot \mathcal{C}(S')$$

$$\mathcal{C}(\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S') = \mathcal{C}(b, k) \cdot \mathcal{C}(S) \cdot (\mathsf{branch}(k')) \cdot \mathcal{C}(S')$$
$$\text{where: } k = sz(S) + 1, \quad k' = sz(S')$$

$$\mathcal{C}(\mathsf{while}\ b\ \mathsf{do}\ S) = \mathcal{C}(b, k) \cdot \mathcal{C}(S) \cdot \mathsf{branch}(k')$$
$$\text{where: } k = sz(S) + 1, \quad k' = -(sz(b) + sz(S) + 1)$$

$$\mathcal{C}(\mathsf{prog}\ S) = \mathcal{C}(S) \cdot \mathsf{halt}$$

Table 1.6: Compilation from Imp to Vm

Proposition 13 (soundness, big-step) *The following properties hold:*

(1) *If* $(e, s) \Downarrow v$ *then* $C \cdot \mathcal{C}(e) \cdot C' \vdash (i, \sigma, s) \xrightarrow{*} (j, v \cdot \sigma, s)$ *where* $i = |C|$ *and* $j = |C \cdot \mathcal{C}(e)|$.

(2) *If* $(b, s) \Downarrow \mathsf{true}$ *then* $C \cdot \mathcal{C}(b, k) \cdot C' \vdash (i, \sigma, s) \xrightarrow{*} (j + k, \sigma, s)$ *where* $i = |C|$ *and* $j = |C \cdot \mathcal{C}(b, k)|$.

(3) *If* $(b, s) \Downarrow \mathsf{false}$ *then* $C \cdot \mathcal{C}(b, k) \cdot C' \vdash (i, \sigma, s) \xrightarrow{*} (j, \sigma, s)$ *where* $i = |C|$ *and* $j = |C \cdot \mathcal{C}(b, k)|$.

(4) *If* $(S, s) \Downarrow s'$ *then* $C \cdot \mathcal{C}(S) \cdot C' \vdash (i, \sigma, s) \xrightarrow{*} (j, \sigma, s')$ *where* $i = |C|$ *and* $j = |C \cdot \mathcal{C}(S)|$.

Proof. All proofs are by induction on the derivation of the evaluation judgment. We detail the proof of property (4) in the case the command is a while loop $\mathsf{while}\ b\ \mathsf{do}\ S$ whose boolean condition b is satisfied. So we have:

$$(b, s) \Downarrow \mathsf{true}, \ (S, s) \Downarrow s'', \ (\mathsf{while}\ b\ \mathsf{do}\ S, s'') \Downarrow s' \ .$$

By definition of the compilation function, we have a code C'' of the shape:

$$C \cdot \mathcal{C}(b, k) \cdot \mathcal{C}(S) \cdot \mathsf{branch}(k') \cdot C' \ .$$

By property (2), we have for any σ, s:

$$C'' \vdash (i, \sigma, s) \xrightarrow{*} (j, \sigma, s) \ ,$$

where $i = |C|$, $j = |C \cdot \mathcal{C}(b, k)|$. By inductive hypothesis on property (4), we have:

$$C'' \vdash (j, \sigma, s) \xrightarrow{*} (j', \sigma, s'') \ ,$$

where $j' = |C \cdot \mathcal{C}(b, k) \cdot \mathcal{C}(S)|$. Since $k' = -(|\mathcal{C}(b, k) \cdot \mathcal{C}(S)| + 1)$, we have:

$$C'' \vdash (j, \sigma, s'') \to (i, \sigma, s'') \ .$$

Again, by inductive hypothesis on property (4) we have:

$$C'' \vdash (i, \sigma, s'') \xrightarrow{*} (j'', \sigma, s') \ ,$$

where $j'' = |C \cdot \mathcal{C}(\mathsf{while}\ b\ \mathsf{do}\ S)|$. □

We can prove similar results working with the small-step reduction of the Imp language. To this end, given a Vm code C, we define an 'accessibility relation' $\overset{C}{\rightsquigarrow}$ as the least binary relation on $\{0, \ldots, |C| - 1\}$ such that:

$$\frac{}{i \overset{C}{\rightsquigarrow} i} \qquad \frac{C[i] = \mathsf{branch}(k) \quad (i + k + 1) \overset{C}{\rightsquigarrow} j}{i \overset{C}{\rightsquigarrow} j} \ .$$

Thus $i \overset{C}{\rightsquigarrow} j$ if in the code C we can go from i to j following a sequence of unconditional jumps. We also introduce a ternary relation $R(C, i, K)$ which relates a Vm code C, a number $i \in \{0, \ldots, |C| - 1\}$, and a continuation K. The intuition is that relative to the code C, the instruction i can be regarded as having continuation K.

Definition 14 *The ternary relation R is the least one that satisfies the following conditions:*

$$\frac{i \overset{C}{\rightsquigarrow} j \qquad C[j] = \mathsf{halt}}{R(C, i, \mathsf{halt})} \qquad \frac{\begin{array}{c} i \overset{C}{\rightsquigarrow} i' \quad C = C_1 \cdot \mathcal{C}(S) \cdot C_2 \\ i' = |C_1| \quad j = |C_1 \cdot \mathcal{C}(S)| \quad R(C, j, K) \end{array}}{R(C, i, S \cdot K)} \ .$$

We can then state the correctness of the compilation function as follows.

Proposition 15 (soundness, small-step) *If $(S, K, s) \to (S', K', s')$ and $R(C, i, S \cdot K)$ then $C \vdash (i, \sigma, s) \overset{*}{\to} (j, \sigma, s')$ and $R(C, j, S' \cdot K')$.*

Proof. Preliminary remarks:

1. The relation $\overset{C}{\rightsquigarrow}$ is transitive.
2. If $i \overset{C}{\rightsquigarrow} j$ and $R(C, j, K)$ then $R(C, i, K)$.

The first property can be proven by induction on the definition of $\overset{C}{\rightsquigarrow}$ and the second by induction on the structure of K. Next we can focus on the proof of the assertion. The notation $C \overset{i}{\cdot} C'$ means that $i = |C|$. Suppose that:

$$(1) \quad (S, K, s) \to (S', K', s') \quad \text{and} \quad (2) \quad R(C, i, S \cdot K) \ .$$

From (2), we know that there exist i' and i'' such that:

$$(3) \quad i \overset{C}{\rightsquigarrow} i', \quad (4) \quad C = C_1 \overset{i'}{\cdot} \mathcal{C}(S) \overset{i''}{\cdot} C_2, \quad \text{and} \quad (5) \quad R(C, i'', K) \ .$$

And from (3) it follows that:

$$(3') \quad C \vdash (i, \sigma, s) \overset{*}{\to} (i', \sigma, s) \ .$$

We are looking for j such that:

$$(6) \quad C \vdash (i, \sigma, s) \overset{*}{\to} (j, \sigma, s'), \quad \text{and} \quad (7) \quad R(C, j, S' \cdot K') \ .$$

We proceed by case analysis on S. We just detail the case of the conditional statement as the remaining cases have similar proofs. If $S = \mathsf{if}\ e_1 < e_2\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$ then (4) is rewritten as follows:

$$C = C_1 \overset{i'}{\cdot} \mathcal{C}(e_1) \cdot \mathcal{C}(e_2) \cdot \mathsf{bge}(k_1) \overset{a}{\cdot} \mathcal{C}(S_1) \overset{b}{\cdot} \mathsf{branch}(k_2) \overset{c}{\cdot} \mathcal{C}(S_2) \overset{i''}{\cdot} C_2 \ ,$$

where $c = a + k_1$ and $i'' = c + k_2$. We distinguish two cases according to the evaluation of the boolean condition. We describe the case $(e_1 < e_2, s) \Downarrow \mathsf{true}$. We set $j = a$.

$C[i] =$	Conditions for $C : h$
$\mathsf{cnst}(n)$ or $\mathsf{var}(x)$	$h(i+1) = h(i) + 1$
add	$h(i) \geq 2, \quad h(i+1) = h(i) - 1$
$\mathsf{setvar}(x)$	$h(i) = 1, \quad h(i+1) = 0$
$\mathsf{branch}(k)$	$0 \leq i + k + 1 \leq \lvert C\rvert, \quad h(i) = h(i+1) = h(i+k+1) = 0$
$\mathsf{bge}(k)$	$0 \leq i + k + 1 \leq \lvert C\rvert, \quad h(i) = 2, \quad h(i+1) = h(i+k+1) = 0$
halt	$i = \lvert C\rvert - 1, \quad h(i) = h(i+1) = 0$

Table 1.7: Conditions for well-formed code

- The instance of (1) is $(S, K, s) \to (S_1, K, s)$.
- The reduction required in (6) takes the form $C \vdash (i, \sigma, s) \xrightarrow{*} (i', \sigma, s) \xrightarrow{*} (a, \sigma, s')$, and it follows from $(3')$, the fact that $(e_1 < e_2, s) \Downarrow \mathsf{true}$, and proposition 13(2).
- Property (7), follows from the preliminary remarks, fact (5), and the following proof tree:

$$\dfrac{j \overset{C}{\rightsquigarrow} j \quad \dfrac{b \overset{C}{\rightsquigarrow} i'' \quad R(C, i'', K)}{R(C, b, K)}}{R(C, j, S_1 \cdot K)}$$

□

Remark 16 *We have already noticed that an* Imp *program has* 3 *possible behaviors: (1) it returns a (unique) result, (2) it is stuck in an erroneous situation, (3) it diverges. Proposition 13 guarantees that the compiler preserves behaviors of type (1). Using the small-step reduction rules (proposition 15), we can also conclude that if the source program diverges then the compiled code diverges too. On the other hand, when the source program is stuck in an erroneous situation the compiled code is allowed to have an arbitrary behavior. The following example justifies this choice. Suppose at source level we have an error due to the addition of an integer and a boolean. Then this error does not need to be reflected at the implementation level where the same data type may well be used to represent both integers and booleans.*

Exercise 17 (stack height) *The* Vm *code coming from the compilation of* Imp *programs has very specific properties. In particular, for every instruction of the compiled code it is possible to predict statically,* i.e.*, at compile time, the* height of the stack *whenever the instruction is executed. We say that a sequence of instructions C is well formed if there is a function $h : \{0, \ldots, |C|\} \to \mathbf{N}$ which satisfies the conditions listed in Table 1.7 for $0 \leq i \leq |C| - 1$. In this case we write $C : h$.*

The conditions defining the predicate $C : h$ are strong enough to entail that h correctly predicts the stack height and to guarantee the uniqueness of h up to the initial condition. Show that: (1) If $C : h$, $C \vdash (i, \sigma, s) \xrightarrow{*} (j, \sigma', s')$, and $h(i) = |\sigma|$ then $h(j) = |\sigma'|$. (2) If $C : h$, $C : h'$ and $h(0) = h'(0)$ then $h = h'$.

Next prove that the result of the compilation is a well-formed code. Namely, for any expression e, statement S, and program P the following holds. (3) For any $n \in \mathbf{N}$ there is a unique h such that $\mathcal{C}(e) : h$, $h(0) = n$, and $h(|\mathcal{C}(e)|) = h(0) + 1$. (4) For any S, there is a unique h such that $\mathcal{C}(S) : h$, $h(0) = 0$, and $h(|\mathcal{C}(e)|) = 0$. (5) There is a unique h such that $\mathcal{C}(P) : h$.

1.4 Summary and references

The first step in defining the operational semantics of a programming language amounts to specify the way a program computes. The following steps are the specification of the observables (of a computation) and the definition of a compositional pre-order (or equivalence) on programs.

An alternative and related approach amounts to introduce (partial) correctness assertions on programs and deem two programs equivalent if they satisfy the same assertions. Also the validity of a program's assertion can be reduced to the validity of an ordinary logical statement in a suitable theory of first order logic.

The formal analysis of compilers is a natural application target for operational semantics. Each language in the compilation chain is given a formal semantics and the behavior of the source code is related to the behavior of its representation in intermediate languages, and down to object code.

The lecture notes [Plo04] are an early (first version appeared in 1981) systematic presentation of an operational approach to the semantics of programming languages. Rules for reasoning on partial correctness assertions of simple imperative programs are presented in [Flo67] and [Hoa69] while [MP67] is an early example of mechanized verification of a simple compiler. The presented case study builds on that example and is partially based on [Ler09].

Chapter 2

Rewriting systems

In computer science, a set equipped with a binary reduction relation is an ubiquitous structure arising, *e.g.*, when formalizing the computation rules of an automaton, the generation step of a grammar, or the reduction rules of a programming language (such as the rules for the Imp language in Table 1.3).

Definition 18 *A* rewriting system *is a pair* $(A, \to)$ *where* A *is a set and* $\to \subseteq A \times A$ *is a reduction relation. We write* $a \to b$ *for* $(a, b) \in \to$.

If we regard the reduction relation as an edge relation, we can also say that a rewriting system is a (possibly infinite) directed graph.

Next we introduce some notation. If R is a binary relation we denote with R^{-1} its inverse and with R^* its reflexive and transitive closure. In particular, if $\to$ is a reduction relation we also write $\leftarrow$ for $\to^{-1}$, $\overset{*}{\to}$ for $(\to)^*$ and $\overset{*}{\leftarrow}$ for $(\leftarrow)^*$. Finally, $\overset{*}{\leftrightarrow}$ is defined as $(\to \cup \to^{-1})^*$. This is the equivalence relation induced by the rewriting system.

2.1 Basic properties

Termination and confluence are two relevant properties of rewriting systems. Let us start with *termination*, namely the fact that all reduction sequences terminate.

Definition 19 (termination) *A rewriting system* $(A, \to)$ *is* terminating *if all sequences of the shape* $a_0 \to a_1 \to a_2 \to \cdots$ *are finite.*

In this definition, we require the sequence (not the set) to be finite. In particular, a rewriting system composed of a singleton set A where $\to = A \times A$ is *not* terminating.

When the rewriting system corresponds to the reduction rules of a programming language the termination property is connected to the termination of programs. This is a fundamental property in program verification. As a matter of fact, the verification of a program is often decomposed into the proof of a partial correctness assertion (cf. section 1.2) and a proof of termination.

Example 20 *Let* A *be the set of words composed of a (possibly empty) sequence of 'function symbols'* f *and an integer* n. *Write* f^k *for* $f \cdots f$, k *times, and define a rewriting relation* $\to$ *on* A *as follows:*

$$f^{k+1}(n) \to \begin{cases} f^k(n-10) & \text{if } n > 100 \\ f^{k+2}(n+11) & \text{otherwise.} \end{cases}$$

This is known as McCarthy's function 91*. For instance:*

$$f(100) \to f(f(111)) \to f(101) \to 91 \not\to$$

Proving its termination is not trivial, but the name of the function gives a hint. For another example, let $\mathbf{N}^+$ *be the collection of non-negative natural numbers with the following rewriting relation:*

$$n \to \begin{cases} n/2 & \text{if } n > 1, n \text{ even} \\ 3n+1 & \text{if } n > 1, n \text{ odd.} \end{cases}$$

This is known as Collatz's function and its termination is a long standing open problem.

Exercise 21 *Consider the following* Imp *command (extended with integer addition and division) where* b *is an arbitrary boolean condition:*

$$\mathsf{while}\ (u > l+1)\ \mathsf{do}\ (r := (u+l)/2\ ;\ \mathsf{if}\ b\ \mathsf{then}\ u := r\ \mathsf{else}\ l := r)\ .$$

Show that the evaluation of the command starting from a state satisfying $u, l \in \mathbf{N}$ *terminates.*

Definition 22 (normalizing) *We say that* $a \in A$ *is a* normal form *if* $\not\exists b\ (a \to b)$*. We also say that the rewriting system is* normalizing *if for all* $a \in A$ *there is a finite reduction sequence leading to a normal form.*

A terminating rewriting system is normalizing, but the converse fails. For instance, consider: $A = \{a, b\}$ with $a \to a$ and $a \to b$. In some contexts (*e.g.*, proof theory), a terminating rewriting system is also called *strongly normalizing*. A second property of interest is *confluence*.

Definition 23 (confluence) *A rewriting system* $(A, \to)$ *is* confluent *if for all* $a \in A$*:*

$$\frac{\forall\, b, c\ \ (b \overset{*}{\leftarrow} a \overset{*}{\rightarrow} c)}{\exists\, d\ \ (b \overset{*}{\rightarrow} d \overset{*}{\leftarrow} c)}\ .$$

We also write $b \downarrow c$ *if* $\exists\, d\ \ (b \overset{*}{\rightarrow} d \overset{*}{\leftarrow} c)$*.*

A property related to confluence is the property called *Church-Rosser*, after the logicians who introduced the terminology in the framework of the λ-calculus (cf. chapter 7).

Definition 24 (Church-Rosser) *A rewriting system* $(A, \to)$ *is* Church-Rosser *if for all* $a, b \in A$*,* $a \overset{*}{\leftrightarrow} b$ *implies* $a \downarrow b$*.*

Proposition 25 *A rewriting system is* Church-Rosser *iff it is* confluent*.*

Proof. ($\Rightarrow$) If $a \overset{*}{\rightarrow} b$ and $a \overset{*}{\rightarrow} c$ then $b \overset{*}{\leftrightarrow} c$. Hence $\exists\, d\ \ b \overset{*}{\rightarrow} d, c \overset{*}{\rightarrow} d$.

($\Leftarrow$) If $a \overset{*}{\leftrightarrow} b$ then a and b are connected by a finite sequence of 'picks and valleys'. For instance:

$$a \overset{*}{\rightarrow} c_1 \overset{*}{\leftarrow} c_2 \overset{*}{\rightarrow} c_3 \cdots \overset{*}{\leftarrow} c_n \overset{*}{\rightarrow} b\ .$$

Using confluence, we can then find a common reduct. To show this, proceed by induction on the number of picks and valleys. □

Let us look at possible interactions of the introduced properties.

Proposition 26 *Let $(A, \to)$ be a rewriting system.*

1. *If the rewriting system is confluent then every element has* at most one *normal form.*
2. *If moreover the rewriting system is normalizing then every element has a* unique normal form*.*

PROOF. (1) If an element reduces to two distinct normal forms then we contradict confluence.

(2) By normalization, there exists a normal form and by (1) there cannot be two different ones. □

Exercise 27 *Let $(A, \to_1)$ and $(A, \to_2)$ be two rewriting systems. We say that they* commute *if $a \overset{*}{\to}_1 b$ and $a \overset{*}{\to}_2 c$ implies $\exists\, d$ $(b \overset{*}{\to}_2 d$ and $c \overset{*}{\to}_1 d)$. Show that if $\to_1$ and $\to_2$ are* confluent *and* commute *then $\to_1 \cup \to_2$ is* confluent *too.*

2.2 Termination and well-founded orders

Terminating rewriting systems and *well-founded orders* are two sides of the same coin.

Definition 28 *A* partial order *$(P, >)$ is a set P with a* transitive relation *$>$. A partial order $(P, >)$ is* well-founded *if it is not possible to define a sequence $\{x_i\}_{i \in \mathbf{N}} \subseteq P$ such that:*

$$x_0 > x_1 > x_2 > \cdots$$

Notice that in a well-founded system we cannot have an element x such that $x > x$ for otherwise we can define a sequence $x > x > x > \cdots$ (a similar remark concerned the definition of terminating rewriting system).

Exercise 29 *Let $\mathbf{N}$ be the set of natural numbers, $\mathbf{N}^k$ the cartesian product $\mathbf{N} \times \cdots \times \mathbf{N}$, k-times, and $A = \bigcup\{\mathbf{N}^k \mid k \geq 1\}$. Let $>$ be a binary relation on A such that :*

$$(x_1, \ldots, x_m) > (y_1, \ldots, y_n) \quad \textit{iff} \quad \exists\, k \ \ (k \leq min(n, m), x_1 = y_1, \ldots, x_{k-1} = y_{k-1}, x_k > y_k) \ .$$

Prove or disprove the assertion that $>$ is a well-founded order.

Clearly, every well-founded partial order is a terminating rewriting system if we regard the order $>$ as the reduction relation. Conversely, every terminating rewriting system, say $(P, \to)$, induces the well-founded partial order $(P, \overset{+}{\to})$ where $\overset{+}{\to}$ is the transitive (but *not* reflexive) closure of $\to$.

Proposition 30 (induction principle) *Let $(P, >)$ be a well-founded partial order and for $x \in P$ let $\downarrow(x) = \{y \mid x > y\}$. Then the following* induction principle *holds where $\supset$ stands for the logical implication:*

$$\frac{\forall\, x \ \ ((\downarrow(x) \subseteq B) \supset x \in B)}{B = P} \tag{2.1}$$

PROOF. If x is *minimal* then the principle requires $x \in B$. Otherwise, suppose x_0 is *not minimal* and $x_0 \notin B$. Then there must be $x_1 < x_0$ such that $x_1 \notin B$. Again x_1 is *not minimal* and we can go on to build: $x_0 > x_1 > x_2 > \cdots$ which *contradicts* the hypothesis that P is well-founded. □

Exercise 31 *Explain why the principle fails if P is a singleton set and $>$ is reflexive.*

Remark 32 *On the* natural numbers *the induction principle can be stated as:*

$$\frac{\forall n \ \ (\forall n' < n \ \ n' \in B \supset n \in B)}{\forall n \ \ n \in B} \ ,$$

which is equivalent to the usual reasoning principle:

$$\frac{0 \in B \wedge (\forall n \ \ (n \in B \supset (n+1) \in B))}{\forall n \ \ n \in B} \ .$$

We have shown that on a well-founded order the induction principle holds. The converse holds too in the following sense.

Proposition 33 *Let $(P, >)$ be a partial order for which the* induction principle *(2.1) holds. Then $(P, >)$ is* well-founded*.*

Proof. Define: $Z = \{x \in P \mid$ there is no infinite descending chain from $x\}$ and $\downarrow(x) = \{y \mid y < x\}$. The set Z satisfies the condition: $\forall x \ \ (\downarrow(x) \subseteq Z \supset x \in Z)$. Hence by the induction principle $Z = P$. Thus P is well-founded. □

2.3 Lifting well-foundation

We examine three ways to lift an order to tuples so as to preserve *well-foundation*, namely the product order, the lexicographic order, and the multi-set order.

Definition 34 *Let $(P, >)$ be a partial order and let $P^n = P \times \cdots \times P$, n times, be the cartesian product ($n \geq 2$). The* product order *on P^n is defined by $(x_1, \ldots, x_n) >_p (y_1, \ldots, y_n)$ if:*

$$x_i \geq y_i, i = 1, \ldots, n \quad \text{and} \quad \exists j \in \{1, \ldots, n\} \ \ x_j > y_j \ .$$

The lexicographic order *(from left to right) on P^n is defined by $(x_1, \ldots, x_n) >_{lex} (y_1, \ldots, y_n)$ if:*

$$\exists j \in \{1, \ldots, n\} \ \ x_1 = y_1, \ldots, x_{j-1} = y_{j-1}, x_j > y_j \ .$$

Notice that $(x_1, \ldots, x_n) >_p (y_1, \ldots, y_n)$ implies $(x_1, \ldots, x_n) >_{lex} (y_1, \ldots, y_n)$ but that the converse fails.

Proposition 35 *If $(P, >)$ is well-founded and $n \geq 2$ then $(P^n, >_p)$ and $(P^n, >_{lex})$ are well-founded.*

Proof. For the product order, suppose there is an infinite descending chain in the product order. Then one component must be strictly decreasing infinitely often which contradicts the hypothesis that $(P, >)$ is well-founded. As for the lexicographic order, we proceed by induction on n. For the induction step, notice that the first component must eventually stabilize and then apply induction on the remaining components. □

A third way to compare a finite collection of elements is to consider them as *multi-sets* which we introduce next.

Definition 36 *A* multi-set *M over a set A is a function $M : A \to \mathbf{N}$. If $M(a) = k$ then a occurs k times in the multi-set. A* finite *multi-set is a multi-set M such that $\{a \mid M(a) \neq 0\}$ is finite.*

Definition 37 *Let $\mathcal{M}_{fin}(X)$ denote the* finite multi-sets *over a set X.*

Definition 38 *Assume $(X, >)$ is a partial order and $M, N \in \mathcal{M}_{fin}(X)$. We write $M >_{1,m} N$ if N is obtained from M by* replacing *an element by a multi-set of elements which are strictly smaller.*

Example 39 *If $X = \mathbf{N}$ then $\{\!|1, 3|\!\} >_{1,m} \{\!|1, 2, 2, 1|\!\} >_{1,m} \{\!|0, 2, 2, 1|\!\} >_{1,m} \{\!|0, 1, 1, 2, 1|\!\}$.*

Exercise 40 *Find an example where the relation $>_{1,m}$ is* not *transitive.*

Definition 41 *Let $(X, >)$ be a partial order. We define the* multi-set order *$>_m$ on $\mathcal{M}_{fin}(X)$. as the* transitive closure *of $>_{1,m}$.*

We want to show that if $(X, >)$ is well-founded then $>_m$ is well-founded. First we recall a classical result known in the literature as *König's lemma.*

Proposition 42 *A finitely branching tree with an infinite number of nodes admits an infinite path.*

Proof. First let us make our statement precise. A *tree* can be seen as a subset D of $\mathbf{N}^*$ (finite words of natural numbers) satisfying the following properties.

1. If $w \in D$ and w' is a prefix of w then $w' \in D$.
2. If $wi \in D$ and $j < i$ then $wj \in D$.

Notice that this representation is quite general in that it includes trees with a countable number of nodes and even trees with nodes having a countable number of children (*e.g.*, $\mathbf{N}^*$ is a tree). We say that a tree is *finitely branching* if every node has a finite number of children (this is strictly weaker than being able to bound the number of children of every node!).

Now suppose D is a finitely branching tree with infinitely many nodes. If $\pi \in \mathbf{N}^*$ let $\uparrow(\pi)$ be the set of paths that start with π. We show that it is always possible to extend a path π such that $\uparrow(\pi) \cap D$ is infinite to a longer path $\pi \cdot i$ with the same property, *i.e.*, $\uparrow(\pi \cdot i) \cap D$ is infinite. Indeed, the hypothesis that D is finitely branching entails that there are finitely many $i_1, \ldots, i_k$ such that $\pi i_j \in D$. Since $\uparrow(\pi) \cap D$ is infinite one of these branches, say i, must be used infinitely often. So we have that $\uparrow(\pi \cdot i) \cap D$ is infinite. □

Proposition 43 *If $(P, >)$ is* well-founded *then $(\mathcal{M}_{fin}(P), >_m)$ is well-founded.*

Proof. By contradiction suppose we have an infinitely descending chain:

$$X_0 >_m X_1 >_m \cdots$$

Because $>_m$ is the transitive closure of $>_{1,m}$ this gives an infinitely descending chain:

$$Y_0 >_{1,m} Y_1 >_{1,m} \cdots$$

where $X_0 = Y_0$. By definition of $>_{1,m}$, the step from Y_i to Y_{i+1} consists in taking an element of Y_i, say y, and replacing it by a *finite* multi-set of elements $\{\!|y_1, \ldots, y_k|\!\}$ which are strictly smaller. Suppose we have drawn a tree whose leaves correspond to the elements of Y_i (if needed we may add a special root node). Then to move to Y_{i+1} we have to take a leaf of Y_i, which corresponds to the element y, and add k branches labelled with the elements $y_1, \ldots, y_k$ (if $k = 0$ we may just add one branch leading to a special 'sink node' from which no further expansion is possible). The tree we build in this way is finitely branching and is infinite. Then by König's lemma (proposition 42) there must be an infinite path in it which corresponds to an infinitely descending chain in $(P, >)$. This is a contradiction since $(P, >)$ is supposed to be well-founded. □

Exercise 44 *Does the evaluation of the following* Imp *commands terminate assuming initially a state where m, n are positive natural numbers?*

$$\mathsf{while}\ (m \neq n)\ \mathsf{do}\ (\mathsf{if}\ (m > n)\ \mathsf{then}\ m := m - n;\ \mathsf{else}\ n := n - m;)\ ,$$
$$\mathsf{while}\ (m \neq n)\ \mathsf{do}\ (\mathsf{if}\ (m > n)\ \mathsf{then}\ m := m - n;\ \mathsf{else}\ (h := m; m := n; n := h;))\ .$$

Exercise 45 *Let $(A, \to)$ be a rewriting system and let $\mathbf{N}$ be the set of natural numbers. A* monotonic embedding *is a function $\mu : A \to \mathbf{N}$ such that if $a \to b$ then $\mu(a) >_{\mathbf{N}} \mu(b)$. Define the set of* immediate successors *of $a \in A$ as: $suc(a) = \{b \mid a \to b\}$, and say that A is finitely branching if for all elements $a \in A$, $suc(a)$ is a finite set. Prove that: (1) If a rewriting system has a monotonic embedding then it terminates. (2) If a rewriting system is finitely branching and terminating then it has a monotonic embedding. (3) The following rewriting system $(\mathbf{N} \times \mathbf{N}, \to)$ where: $(i+1, j) \to (i, k)$ and $(i, j+1) \to (i, j)$, for $i, j, k \in \mathbf{N}$, is terminating,* not *finitely branching, and* does not *have a monotonic embedding.*

2.4 Termination and local confluence

In general, it is hard to prove confluence because we have to consider arbitrary long reductions. It is much simpler to reason *locally*.

Definition 46 *A rewriting system $(A, \to)$ is* locally confluent *if for all $a \in A$:*

$$\frac{\forall\, b, c \in A\ \ (b \leftarrow a \to c)}{\exists\, d \in A\ \ (b \overset{*}{\to} d \overset{*}{\leftarrow} c)}\ .$$

Proposition 47 *If a rewriting system $(A, \to)$ is* locally confluent *and* terminating *then it is* confluent.

Proof. We apply the principle of *well-founded induction* to $(A, \overset{+}{\to})$! Suppose:

$$c_1 \overset{*}{\leftarrow} b_1 \leftarrow a \to b_2 \overset{*}{\to} c_2\ .$$

By *local confluence*: $\exists\, d\ \ (b_1 \overset{*}{\to} d \overset{*}{\leftarrow} b_2)$. Also, by *induction hypothesis* on b_1 and b_2 we have:

$$\exists\, d'\ \ (c_1 \overset{*}{\to} d' \overset{*}{\leftarrow} d)\ , \qquad \exists\, d''\ \ (d' \overset{*}{\to} d'' \overset{*}{\leftarrow} c_2)\ .$$

But then $c_1 \downarrow c_2$. Thus by the *principle of well-founded induction*, the rewriting system is *confluent*. □

Example 48 *Let* $A = \mathbf{N} \cup \{a, b\}$ *and* $\to$ *such that for* $i \in \mathbf{N}$*:* $i \to i+1$*,* $2 \cdot i \to a$*, and* $2 \cdot i + 1 \to b$*. This rewriting system is locally confluent and normalizing, but* not *terminating and* not *confluent.*

Exercise 49 *Let* Σ^* *denote the set of finite words over the alphabet* $\Sigma = \{f, g_1, g_2\}$ *with generic elements* $w, w', \ldots$ *As usual,* ϵ *denotes the empty word. Let* $\to$ *denote the smallest binary relation on* Σ^* *such that for all* $w \in \Sigma^*$*:*

$$(1) \quad fg_1w \ \to \ g_1g_1ffw \ , \quad (2) \quad fg_2w \ \to \ g_2fw \ , \quad (3) \quad f\epsilon \ \to \ \epsilon \ ,$$

and such that if $w \to w'$ *and* $a \in \Sigma$ *then* $aw \to aw'$*. This is an example of* word *rewriting; a more general notion of* term *rewriting will be considered in the following section 2.5. Prove or give a counter-example to the following assertions:*

1. *If* $w \xrightarrow{*} w_1$ *and* $w \xrightarrow{*} w_2$ *then there exists* w' *such that* $w_1 \xrightarrow{*} w'$ *and* $w_2 \xrightarrow{*} w'$*.*
2. *The rewriting system* $(\Sigma^*, \to)$ *is terminating.*
3. *Replacing rule* (1) *with the rule* $fg_1w \to g_1g_1fw$*, the answers to the previous questions are unchanged.*

2.5 Term rewriting systems

When rewriting systems are defined on sets with structure, we can exploit this structure, *e.g.*, to represent in a more succinct way the reduction relation and to reason on its properties. A situation of this type arises when dealing with sets of *first-order terms* (in the sense of first-order logic). Let us fix some notation. A *signature* Σ is a finite set of function symbols $\{f_1, \ldots, f_n\}$ where each function symbol has an *arity*, $ar(f_i)$, which is a natural number indicating the number of arguments of the function. Let V denote a countable set of *variables* with generic elements $x, y, z, \ldots$ If $V' \subseteq V$ then $T_\Sigma(V')$ is the set of *first order terms* over the variables V' with generic elements $t, s, \ldots$ (respecting the arity). So $T_\Sigma(V')$ is the least set which contains the variables V' and such that if $f \in \Sigma$, $n = ar(f)$, and $t_1, \ldots, t_n \in T_\Sigma(V')$ then $f(t_1, \ldots, t_n) \in T_\Sigma(V')$. If t is a term we denote with $\mathsf{var}(t)$ the set of *variables* occurring in the term.

A natural operation we may perform on terms is to *substitute* terms for variables. Formally, a *substitution* is a function $S : V \to T_\Sigma(V)$ which is the identity almost everywhere. We represent with the notation $[t_1/x_1, \ldots, t_n/x_n]$ the substitution S such that $S(x_i) = t_i$ for $i = 1, \ldots, n$ and which is the identity elsewhere. Notice that we always assume $x_i \neq x_j$ if $i \neq j$. We use id to denote a substitution which is the identity everywhere. We extend S to $T_\Sigma(V)$ by defining, for $f \in \Sigma$:

$$S(f(t_1, \ldots, t_n)) = f(S(t_1), \ldots, S(t_n)) \qquad \text{(extension of substitution to terms).}$$

Thanks to this extension, it is possible to *compose* substitutions: $(T \circ S)$ is the substitution defined by the equation:

$$(T \circ S)(x) = T(S(x)) \qquad \text{(composition of substitutions).}$$

As expected, composition is associative and the identity substitution behaves as a left and right identity: $id \circ S = S \circ id = S$.

Example 50 *If* $t = f(x, y)$, $S = [g(y)/x]$, *and* $T = [h/y]$ *then:*

$$(T \circ S)(t) = T(S(t)) = T(f(g(y), y)) = f(g(h), h) \ .$$

Next we aim to define the reduction relation schematically exploiting the structure of first-order terms. A *context* C is a term with exactly one occurrence of a special symbol [ ] called *hole* and of arity 0. We denote with $C[t]$ the term resulting from the replacement of the hole [ ] by t in C. A *term-rewriting rule* (or rule for short) is a pair of terms (l, r) that we write $l \to r$ such that $\mathsf{var}(r) \subseteq \mathsf{var}(l)$; the variables on the right hand side of the rule must occur on the left hand-side too.

Definition 51 *A set of term rewriting rules* $R = \{l_1 \to r_1, \ldots, l_n \to r_n\}$, *where* l_i, r_i, $i = 1, \ldots, n$ *are terms over some signature* Σ, *induces a rewriting system* $(T_\Sigma(V), \to_R)$ *where* $\to_R$ *is the least binary relation such that is* $l \to r \in R$ *is a rule,* C *is a context, and* S *is a substitution then:*

$$C[Sl] \to_R C[Sr] \ .$$

Example 52 *Assume the set of rules* R *is as follows:*

$$f(x) \ \to \ g(f(s(x))) \ , \qquad i(0, y, z) \ \to \ y \ , \qquad i(1, y, z) \ \to \ z \ .$$

Then, for instance:

$$\begin{array}{llllll} f(s(y)) & \to_R & g(f(s(s(y)))) & \to_R & g(g(f(s(s(s(y)))))) & \to_R \ \cdots \\ i(0, 1, f(y)) & \to_R & 1 & & & \\ i(0, 1, f(0)) & \to_R & i(0, 1, g(f(s(0)))) & \to_R & \cdots & \end{array}$$

There is a natural interplay between equational and term rewriting systems. We illustrate this situation with a few examples.

Example 53 *Suppose we have a set of equations dealing with natural numbers:*

$$\begin{array}{ll} +(x, Z) = +(Z, x) = x, & +(S(x), y) = +(x, S(y)) = S(+(x, y)), \\ +(+(x, y), z) = +(x, +(y, z)) \ . & \end{array}$$

Here the numbers are written in unary notation with a zero Z *and a successor* S *function symbols, and the equations are supposed to capture the behavior of a binary addition symbol* $+$. *Now it is tempting to* orient the equations *so as to simplify the expression. E.g.* $+(x, Z) \to x$, *but this is* not *always obvious! For instance, what is the orientation of:*

$$+(S(x), y) = S(+(x, y)) \quad or \quad +(+(x, y), z) = +(x, +(y, z)) \ ?$$

One proposal could be:

$$\begin{array}{lll} +(x, Z) \to x, & +(Z, x) \to x, & +(S(x), y) \to S(+(x, y)), \\ +(x, S(y)) \to S(+(x, y)), & +(+(x, y), z) \to +(x, +(y, z)) \ . & \end{array}$$

Thus we have defined a term rewriting system *and some interesting and natural questions arise. Is there a reduction strategy always leading to a normal form? Does* any *reduction strategy reach a normal form? Suppose we apply different reduction strategies, is it always possible to reach a common reduct?*

In our case we are lucky. Termination (and therefore normalization) is guaranteed. Moreover the system is confluent and therefore each term has a unique normal form. These properties can be verified automatically by state of the art tools dealing with term rewriting systems. Once these properties are verified, we have a strategy to decide the equality of two terms: we reduce the terms to their normal forms and check whether they are identical.

Example 54 *In this example we look at the equations of* group theory*:*

$$*(e,x) = x, \quad *(x,e) = x, \quad *(i(x),x) = e, \quad *(x,i(x)) = e, \quad *(*(x,y),z) = *(x,*(y,z)) \ .$$

Here e is the identity, i is the inverse function, and $*$ is the binary operation of a group. If we orient the equations from left to right we obtain a term rewriting system and again automatic tools can check that the system is terminating. However the system as it stands is not *confluent. In this case, a procedure known as* completion *tries to add rewriting rules to the system which are compatible with the equations and preserve termination. A possible outcome of this analysis is to add the following rules:*

$$\begin{array}{lll} i(e) \to e, & *(i(x),*(x,y)) \to y, & i(i(x)) \to x, \\ *(x,*(i(x),y)) \to y \ , & i(*(x,y)) \to *(i(y),i(x)) \ . & \end{array}$$

The previous examples may give the impression that checking termination and confluence is a task that can be automatized. While this is true in many practical cases, the reader should keep in mind that in general these properties are undecidable. Term rewriting systems constitute a powerful computational model and it is easy to reduce, *e.g.*, the halting problem for Turing machines to a termination problem for term rewriting systems.

2.6 Summary and references

We have shown that the following concepts are 'equivalent': (1) terminating rewriting system, (2) well-founded set, and (3) partial order with well-founded induction principle. Also, whenever working in a terminating rewriting system we have shown that to prove confluence it suffices to prove local confluence. We have also introduced the notion of term rewriting system which is a way of presenting schematically a rewriting system using first-order terms. Term rewriting systems are tightly connected to equational theories and can provide procedures to decide when two expressions are equated. The book [BN99] is a standard and quite readable introduction to term rewriting. Proposition 42 is a special case of a theorem due to König [K$\ddot{2}$6] while proposition 47 is due to Newman [New42].

Chapter 3

Syntactic unification

Syntactic unification is about solving equations on terms, or equivalently on finite labelled trees. We introduce some notation and terminology. We write $t = s$ if the terms t and s are *syntactically equal*. We define a *pre-order on substitutions* as follows:

$$R \leq S \;\text{ iff }\; \exists T \;\; T \circ R = S \; .$$

Thus $R \leq S$ if S is an *instance* of R or, equivalently, if R is *more general* than S (note that $id \leq S$, for any S).

Exercise 55 *Give an example of two substitutions S, T such that: $S \neq T$, $S \leq T$, and $T \leq S$.*

A *system of equations* E is a finite set of pairs $\{t_1 = s_1, \ldots, t_n = s_n\}$. A substitution S *unifies* a system of equations E, written $S \models E$, if $St = Ss$ (here $=$ means identity on $T_\Sigma(V)$) for all $t = s \in E$. Notice that we are *abusing notation* by using $=$ both for the identity on terms (semantic level) and for a constraint relation (syntactic level).

Exercise 56 *Show that if S is a substitution unifying the system $\{s_1 = s_2, x = t\}$ then S unifies $\{[t/x]s_1 = [t/x]s_2\}$ too.*

3.1 A basic unification algorithm

A *basic algorithm* for unification is presented in table 3.1 as a rewriting system over pairs (E, S) and a special symbol $\perp$ (the symmetric rules for (vt_i), $i = 1, 2$, are omitted). This 'abstract' presentation of the algorithm is instrumental to the proof of its properties. The idea is that we *transform* the system leaving the set of its solutions unchanged till either the solution is explicit or it appears that no solution exists. This is a standard methodology for solving systems of constraints, *e.g.*, consider Gaussian elimination for solving systems of linear equations.

Example 57 *Applying the unification algorithm to the system:*

$$\{f(x) = f(f(z)), g(a, y) = g(a, x)\} \; ,$$

leads to the substitution: $S = [f(z)/y] \circ [f(z)/x]$.

Exercise 58 *Apply the unification algorithm to the systems of equations: $\{f(x, f(x, y)) = f(g(y), f(g(a), z))\}$, a constant, and $\{f(x, f(y)) = f(y, f(f(x)))\}$.*

3.2 Properties of the algorithm

We analyse formally the unification algorithm.

Proposition 59 *The following properties of the algorithm specified in table 3.1 hold:*

1. *The reduction relation $\to$ terminates.*
2. *If $(E, id) \to^* (\emptyset, S)$ then S unifies E.*
3. *If T unifies E then all reductions starting from (E, id) terminate with some $(\emptyset, S)$ such that $S \leq T$.*

Proof. (1) We define a measure on a set of equations as $\mu(E) = (m, n)$ where pairs are lexicographically ordered from left to right (cf. section 2.3), m is the number of variables in E, and n is the number of symbols in the terms in E. The measure is extended to pairs (E, S) and $\bot$ by defining $\mu(E, S) = \mu(E)$ and $\mu(\bot) = (0, 0)$. Then we check that $(E, S) \to U$ implies $\mu(E, S) > \mu(U)$.

(2) We start with a preliminary remark. In (E, S), the second component S is just used to *accumulate* the substitutions. Therefore:

$$(E, S) \to^m (\emptyset, S_n \circ \ldots \circ S_1 \circ S) \text{ iff } (E, id) \to^m (\emptyset, S_n \circ \ldots \circ S_1) ,$$

where $m \geq 1$, $n \geq 0$ and the S_i are the elementary substitutions of the shape $[t/x]$ introduced by rule (vt_1). Next we prove the assertion by induction on the length of the derivation. For instance, suppose:

$$(E \cup \{x = t\}, id) \to ([t/x]E, [t/x]) \to^* (\emptyset, S \circ [t/x])$$

Then, by the preliminary remark, the inductive hypothesis applies to $([t/x]E, id)$. Thus $S \models [t/x]E$. Which entails $S \circ [t/x] \models E$. Moreover, since $x \notin \mathsf{var}(t)$, $S \circ [t/x](x) = S(t) = S \circ [t/x](t)$.

(3) By (1), all reduction sequences terminate. We proceed by induction on the length of the reduction sequence. We observe that if E is not empty then at least one rule applies. Since $T \models E$ it is easily checked that rules (vt_2) and (f_2) do *not* apply. Now suppose, for instance, that:

$$(E \cup \{x = t\}, id) \to ([t/x]E, [t/x])$$

applying rule (vt_1). We recall (exercise 56) that if $T \models E \cup \{x = t\}$ then $T \models [t/x]E$ and $T = T \circ [t/x]$. Then, from $T \models [t/x]E$ and by inductive hypothesis we conclude that $([t/x]E, id) \to^* (\emptyset, S)$ and $S \leq T$. Hence: $S \circ [t/x] \leq T \circ [t/x] = T$. □

(v)	$(E \cup \{x = x\}, S)$	$\to (E, S)$	
(vt_1)	$(E \cup \{x = t\}, S)$	$\to ([t/x]E, [t/x] \circ S)$	if $x \notin \mathsf{var}(t)$
(vt_2)	$(E \cup \{x = t\}, S)$	$\to \bot$	if $x \neq t, x \in \mathsf{var}(t)$
(f_1)	$(E \cup \{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\}, S)$	$\to (E \cup \{t_1 = s_1, \ldots, t_n = s_n\}, S)$	
(f_2)	$(E \cup \{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)\}, S)$	$\to \bot$	if $f \neq g$

Table 3.1: Unification algorithm

Exercise 60 *Let the* size of a term *be the number of nodes in its tree representation. Consider the following unification problem:*

$$\{x_1 = f(x_0, x_0), x_2 = f(x_1, x_1), \ldots, x_n = f(x_{n-1}, x_{n-1})\} \ . \tag{3.1}$$

Compute the most general unifier S. Show that the size of $S(x_n)$ is exponential in n.

In view of exercise 60, you could expect unification algorithms to be hopelessly inefficient. However a closer look at the solution of the unification problem (3.1) reveals that the solution can be represented compactly as soon as we move from a tree representation to a directed acyclic graph (*dag*) representation. This change of perspective allows to share terms and keep the size of $S(x_n)$ linear in n. Indeed, unification algorithms based on a dag representation can be implemented to run in *quasi-linear* time.

Exercise 61 *Propose a method to transform a unification problem of the shape:*

$$E = \{t_1 = s_1, \ldots, t_n = s_n\}$$

over a signature $\Sigma = \{g_1, \ldots, g_m\}$ with $n, m \geq 1$ into a unification problem E' with the following properties:

1. *The problem E' contains exactly one equation.*
2. *The terms in E' are built over a signature Σ' containing exactly one binary symbol f.*
3. *The problem E has a solution if and only if the problem E' has a solution.*
4. *Apply the method to the system: $E = \{x = h(y), \quad g(c, x, z) = g(y, z, z)\}$, where x, y, z are variables.*

Exercise 62 *Let $t, s, \ldots$ be terms over a signature Σ. We say that t is a* filter *(or pattern) for s if there is a substitution S such that $St = s$. In this case we write: $t \leq s$. Show or give a counter-example to the following assertions:*

1. *If $t \leq s$ then t and s are unifiable.*
2. *If t and s are unifiable then $t \leq s$ and $s \leq t$.*
3. *If $t \leq s$ and $s \leq t$ then s and t are unifiable.*
4. *For all t, s one can find r such that $r \leq t$ and $r \leq s$.*
5. *For all t, s one can find r such that $r \geq t$ and $r \geq s$.*

3.3 Summary and references

We have shown that there is a simple algorithm to solve the unification problem on first-order terms. The algorithm either shows that no solution exists or computes a *most general one*. Moreover the algorithm is efficient as soon as terms are represented as directed acyclic graphs. The unification algorithm was brought to the limelight by Robinson's work on the resolution principle and its application to theorem proving [Rob65].

Chapter 4

Termination of term rewriting systems

We introduce two methods to prove *termination of TRS*. The *interpretation method*, where we regard the function symbols as certain strictly monotonic functions, and the *recursive path order* (RPO) method which provides a syntactic criterion to compare terms. We give two proofs that RPO's guarantee termination. The first relies on *reducibility candidates*, a technique imported from proof theory, and the second on the notion of *well-partial order* and a combinatorial result on the embedding of trees known as *Kruskal's theorem*. The interpretation and the RPO methods are examples of *reduction orders* which are defined as follows.

Definition 63 (reduction order) *A* reduction order $>$ *is a* well-founded order *on* $T_\Sigma(V)$ *that is* closed under context and substitution*:*

$$\frac{t > s}{C[t] > C[s] \ , \ St > Ss} \ ,$$

where C is any one hole context and S is any substitution.

The notion of reduction order is quite general.

Proposition 64 *A TRS R* terminates *iff there is a* reduction order $>$ *such that $l \to r \in R$ implies $l > r$.*

Proof. ($\Rightarrow$) If the system terminates then the transitive closure of the reduction relation provides a reduction order.

($\Leftarrow$) If we have a reduction order then well-foundedness enforces termination. □

4.1 Interpretation method

Suppose the TRS is given over a *signature* Σ. Fix a *well-founded set* $(A, >)$ and assume that for each function symbol $f \in \Sigma$, with arity n, we select a function $f^A : A^n \to A$ which is *strictly monotonic*. That is, for all $a_1, \ldots, a_n, a_i'$ if $a_i' > a_i$ then

$$f^A(a_1, \ldots, a_{i-1}, a_i', a_{i+1}, \ldots, a_n) > f^A(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n) \ .$$

Now if we fix an *assignment* $\theta : V \to A$, for every $t \in T_\Sigma(V)$ there is a unique interpretation in A which is defined as follows:

$$[\![x]\!]\theta = \theta(x) \ , \qquad [\![f(t_1, \ldots, t_n)]\!]\theta = f^A([\![t_1]\!]\theta, \ldots, [\![t_n]\!]\theta) \ .$$

Incidentally, this is the usual interpretation of terms in *first-order logic*: a term t with variables $x_1, \ldots, x_n$ induces a function $g_t : A^n \to A$ such that:

$$g_t(a_1, \ldots, a_n) = [\![t]\!][a_1/x_1, \ldots, a_n/x_n] \ .$$

In particular, a variable x is interpreted as the identity function: $g_x(a) = [\![x]\!][a/x] = a$.

Proposition 65 *Under the hypotheses described above, the interpretation induces a* reduction order $>_A$ *on* $T_\Sigma(V)$ *defined by:* $t >_A s$ *if* $\forall \theta \ [\![t]\!]\theta >_A [\![s]\!]\theta$.

Proof. First, let us show $>_A$ is well founded. Suppose by contradiction:

$$t_0 >_A t_1 >_A \cdots$$

Then by taking an arbitrary assignment θ we have: $[\![t_0]\!]\theta >_A [\![t_1]\!]\theta >_A \cdots$ But this contradicts the hypothesis that $(A, >)$ is well-founded.

Second, let us check that $>_A$ is preserved by substitution. Suppose $t >_A t'$. For any s, x we show $[s/x]t >_A [s/x]t'$ (the generalization to a substitution $[s_1/x_1, \ldots, s_n/x_n]$ is left to the reader). In other terms, we have to show that for any assignment θ:

$$[\![[s/x]t]\!]\theta >_A [\![[s/x]t']\!]\theta \ .$$

We note that: $[\![[s/x]t]\!]\theta = [\![t]\!]\theta[[\![s]\!]\theta/x]$. Thus taking $\theta' = \theta[[\![s]\!]\theta/x]$ we have:

$$[\![[s/x]t]\!]\theta = [\![t]\!]\theta' >_A [\![t']\!]\theta' = [\![[s/x]t']\!]\theta \ .$$

Third, we check that $>_A$ is preserved by contexts. To do this, we proceed by induction on the context. The case for the empty context is immediate. For the inductive step, suppose $C = f(\cdots, C', \cdots)$. By inductive hypothesis, $C'[t] >_A C'[s]$ if $t >_A s$. Then we conclude by using the fact that f^A is strictly monotonic in every argument. □

Corollary 66 *Let R be a TRS and A be an interpretation as specified above. Then the TRS terminates if for all $l \to r \in R$ we have: $l >_A r$.*

Proof. We have shown that $>_A$ is a reduction order and we have previously observed (proposition 64) that a system is terminating if all its rules are compatible with a reduction order. □

Example 67 *Polynomial interpretations are an important and popular class of interpretations. Take $A = \{n \in \mathbf{N} \mid n \geq a \geq 1\}$. With $f^n \in \Sigma$ associate a* multivariate polynomial *$p_f(x_1, \ldots, x_n)$ such that:*

1. *Coefficients range over the natural numbers. Thus there are* no negative coefficients *and the polynomials are* monotonic.
2. *$p_f(a, \ldots, a) \in A$. Thus p_f defines a function over the domain A.*

3. *Every variable appears in a monomial with a non-zero multiplicative coefficient Thus we have* strictly monotonic functions.

By extension, we associate with a term t with variables $x_1, \ldots, x_n$ a multivariate polynomial p_t with variables $x_1, \ldots, x_n$. Notice that by taking $a \geq 1$, we make sure multiplication is a strictly monotonic function.

Example 68 *Consider the following rules for* addition *and* multiplication *over* natural numbers in unary notation*:*

$$a(\mathsf{z}, y) \to y \ , \qquad a(x, \mathsf{z}) \to x \ , \qquad a(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{s}(a(x, y)) \ ,$$
$$m(\mathsf{z}, x) \to \mathsf{z} \ , \qquad m(\mathsf{s}(x), y) \to a(y, m(x, y)) \ .$$

A polynomial interpretation showing the termination of this TRS is:

$$p_{\mathsf{z}} = 1 \ , \quad p_{\mathsf{s}} = x + 2 \ , \quad p_a = 2x + y + 1 \ , \quad p_m = (x + 1)(y + 1) \ .$$

Exercise 69 *Find a polynomial interpretation showing the termination of the TRS:*

$$f(f(x, y), z) \to f(x, f(y, z)) \ , \qquad f(x, f(y, z)) \to f(y, y) \ .$$

Exercise 70 *(1) Find a polynomial interpretation for the TRS:*

$$\begin{array}{lll} x + \mathsf{0} \to x \ , & x + \mathsf{s}(y) \to \mathsf{s}(x + y) \ , & \textit{(addition)} \\ d(0) \to 0 \ , & d(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(d(x))) \ , & \textit{(double)} \\ q(\mathsf{0}) \to \mathsf{0} \ , & q(\mathsf{s}(x)) \to q(x) + \mathsf{s}(d(x)) & \textit{(square).} \end{array}$$

(2) *Consider the term $t \equiv q^{n+1}(s^2 0)$ whose size is linear in n. Show that there is a reduction:*

$$t \xrightarrow{*} q(s^{2^{2^n}}) \ ,$$

whose length is doubly exponential in n.

While polynomial interpretations are a conceptually simple method to prove termination the reader should keep in mind that they suffer of a couple of limitations. First, polynomial interpretations are hard to find. Indeed in general even checking whether a polynomial interpretation is valid is *undecidable*. This follows from the undecidability of the so-called Hilbert's 10^{th} problem. This is the problem of recognizing the multivariate polynomials with integers coefficients which have a zero. The problem was stated in 1900, and finally in 1970 Matiyasevich proved that the problem is undecidable. Second, polynomial interpretations cannot handle fast growing functions. Indeed it can be shown that the length of reductions of TRS proven terminating by a polynomial interpretation can be at most *double exponential*. Exercise 70(2) provides a lower bound, and the upper bound is not too hard to obtain. In theory one could then consider interpretations based on faster growing functions such as exponentials, towers of exponentials,... however in practice most automatic systems just look for *low degree* polynomial interpretations.

$$(R_1) \qquad \frac{s \geq_r t}{f(\ldots s \ldots) >_r t}$$

$$(R_2) \qquad \frac{f >_\Sigma g \quad f(s_1, \ldots, s_m) >_r t_i \;\; i = 1, \ldots, n}{f(s_1, \ldots, s_m) >_r g(t_1, \ldots, t_n)}$$

$$(R_3) \qquad \frac{(s_1, \ldots, s_m) >_r^{\tau(f)} (t_1, \ldots, t_m) \quad f(s_1, \ldots, s_m) >_r t_i \;\; i = 1, \ldots, m}{f(s_1, \ldots, s_m) >_r f(t_1, \ldots, t_m)}$$

Table 4.1: Recursive path-order

4.2 Recursive path order

Recursive path orders are a family of reduction orders which are defined by induction on the structure of the terms. The way to compare terms is rather simple. First we assume a strict partial order $>_\Sigma$ on the function symbols in Σ (since Σ is supposed finite, $>_\Sigma$ is well-founded). If $f >_\Sigma g$, proving that:

$$t = f(t_1, \ldots, t_n) >_r g(s_1, \ldots, s_m) = s \ ,$$

reduces to proving: $t >_r s_i$ for $i = 1, \ldots, m$. On the other hand, proving that:

$$t = f(t_1, \ldots, t_n) >_r f(s_1, \ldots, s_m) = s \ ,$$

reduces to proving that: $(t_1, \ldots, t_n) >_r (s_1, \ldots, s_m)$, according to one of the orders that preserve well-foundation we have considered in chapter 2, namely product order, lexicographic order, or multi-set order. What we have described is *almost* the official definition of recursive path order which is given in Table 4.1.

In this definition, we assume that every function symbol f is assigned a status $\tau(f)$ which determines how f's arguments are to be compared (product, lexicographic, multi-set,...) Indeed, this is necessary to guarantee termination. For instance, consider the non-terminating TRS:

$$f(a, b) \to f(b, a) \ , \quad f(b, a) \to f(a, b) \ ,$$

with $\Sigma = \{f, a, b\}$. Assume $a >_\Sigma b$. If f's arguments could be compared with a lexicographic order from left to right *or* from right to left then we could prove both $f(a, b) >_r f(b, a)$ *and* $f(b, a) >_r f(a, b)$.

Exercise 71 *Consider the TRS:*

$$(x + y) + z \quad \to \quad x + (y + z) \ , \qquad x * \mathsf{s}(y) \quad \to \quad x + (y * x) \ .$$

Find a status for the function symbols that allows to prove:

$$(x + y) + z \quad >_r \quad x + (y + z) \ , \qquad x * \mathsf{s}(y) \quad >_r \quad x + (y * x) \ .$$

Another point that deserves to be stressed is that in the rule (R_3) we also require that the term on the left is larger than all the arguments of the term on the right. To see the necessity of this condition, consider the non-terminating TRS:

$$f(a, y) \to f(b, f(a, y)) \ ,$$

where $\Sigma = \{f, a, b\}$, $a >_\Sigma b$, and the status of f is lexicographic from left to right.

Finally, we notice that there is an additional rule (rule R_1) that entails that a term is larger than all its proper subterms. This 'subterm property' is characteristic of an important class of orders known as *simplification order* that we define next.

Definition 72 *A* strict *order* $>$ *on* $T_\Sigma(V)$ *is a* simplification order *if it is* closed under context and substitution *and moreover for all functions* $f \in \Sigma$ *it satisfies:*

$$f(x_1, \ldots, x_n) > x_i \text{ for } i = 1, \ldots, n \ .$$

Exercise 73 *Show that if* $>$ *is a simplification order and* C *is a one hole context with* $C \neq [\,]$ *then* $C[t] > t$.

We prove next that the recursive path order is a simplification order. Further it will be proven in section 4.4 that every simplification order is well-founded. This proof relies on a classical combinatorial argument known as *Kruskal's theorem*. This is enough to guarantee that the recursive path order is a *reduction order* and therefore can be used to prove the termination of TRS. We will also give in section 4.3 a direct proof of the fact that the recursive path order is well-founded that avoids the detour through Kruskal's theorem by using a so called *reducibility argument* (a standard method to prove termination of typed λ-calculi introduced in chapter 10).

Proposition 74 *The* recursive path order *is a* simplification order *on* $T_\Sigma(V)$.

Proof. To fix the ideas, we consider a particular case where we always compare tuples via the *product order*. We prove the following properties: (1) $>$ is strict, (2) $s > t$ implies $\mathsf{var}(s) \supseteq \mathsf{var}(t)$, (3) transitivity, (4) subterm property, (5) closure under substitution, and (6) closure under context.

Before proceeding, we formulate in Table 4.2 a simplified definition of recursive path order for functions having *product status*. Notice that in (R_3) we drop the condition $f(s_1, \ldots, s_m) >_r t_i$ for $i = 1, \ldots, m$. It turns out that in this case the condition can be derived from the transitivity property and the fact that for $i = 1, \ldots, m$: $f(s_1, \ldots, s_m) >_r s_i \geq_r t_i$.

$>$ is strict By induction on s show that $s > s$ is impossible. Note in particular that $x > t$ and $f > f$ are impossible.

$s > t$ implies $\mathsf{var}(s) \supseteq \mathsf{var}(t)$. By induction on the proof of $s > t$.

Transitivity Suppose $s_1 > s_2$ and $s_2 > s_3$. Show $s_1 > s_3$ by induction on $|s_1| + |s_2| + |s_3|$ analyzing the last rules applied in the proof of $s_1 > s_2$ and $s_2 > s_3$ (9 cases).

Subterm property We check that $f(x_1, \ldots, x_n) > x_i$ for $i = 1, \ldots, n$.

Closure under substitution Show that $t > r$ implies $[s/x]t > [s/x]r$ by induction on $|t| + |r|$.

$$(R_1) \qquad \frac{s \geq_r t}{f(\ldots s \ldots) >_r t}$$

$$(R_2) \qquad \frac{f >_\Sigma g \quad f(s_1,\ldots,s_m) >_r t_i \;\; i = 1,\ldots,n}{f(s_1,\ldots,s_m) >_r g(t_1,\ldots,t_n)}$$

$$(R_3) \qquad \frac{s_i \geq t_i \text{ for } i \in 1..m, \text{ and } \exists\, j \in 1..m \;\; s_j > t_j}{f(s_1,\ldots,s_m) >_r f(t_1,\ldots,t_m)} \; .$$

Table 4.2: RPO, for functions with product status

Closure under context Show by induction on the structure of a one hole context that $t > s$ implies $C[t] > C[s]$. □

Exercise 75 *Consider the following TRS:*

$$\begin{array}{lcllcl} ack(\mathsf{z},n) & \to & \mathsf{s}(\mathsf{z}) \; , & ack(\mathsf{s}(\mathsf{z}),\mathsf{z}) & \to & \mathsf{s}^2(\mathsf{z}) \; , \\ ack(\mathsf{s}^2(m),\mathsf{z}) & \to & \mathsf{s}^2(m) \; , & ack(\mathsf{s}(m),\mathsf{s}(n)) & \to & ack(ack(m,\mathsf{s}(n)),n) \; . \end{array}$$

This TRS corresponds to a very fast growing function known as Ackermann's function. For instance, this function grows faster than any tower of exponentials[1] *and no polynomial interpretation can prove its termination. In practice, running* $ack(4,4)$ *will produce an* out-of-memory *exception on most computers. Prove the termination by RPO.*

Exercise 76 *The previous exercise 75 marks a point for RPO. However, sometimes the (polynomial) interpretation method beats the RPO method. Consider the TRS:*

$$b(x) \;\to\; r(\mathsf{s}(x)) \; , \qquad r(\mathsf{s}(\mathsf{s}(x))) \;\to\; b(x) \; .$$

(1) Show that the TRS terminates by polynomial interpretation. (2) Show that there is no RPO on Σ *that can prove its termination. (3) RPO is a particular type of simplification order. Is there a simplification order that shows termination of the TRS above?*

Exercise 77 *The previous exercise 76 shows that the termination of certain TRS cannot be proven by RPO. It turns out that using an arbitrary simplification order does not change this state of affairs. Consider the TRS:*

$$f(f(x)) \to f(g(f(x))) \; .$$

(1) Show that the TRS is terminating. (2) Show that there is no simplification order $>$ *that contains* $\to$*.*

We terminate with a few remarks concerning the complexity of working with RPO. Once the order on the signature and the status of the function is fixed, deciding whether $t >_r s$ can be done in *time polynomial* in the size of the terms. However, it is possible to come out with rather artificial examples where the choice of the order on the signature is not obvious. In fact it can be shown that deciding whether $t >_r s$ with respect to *some order* on the signature is an NP-*complete* problem.

[1] Technically, Ackermann showed that this function cannot be defined by *primitive recursion.*

$$\frac{s \geq_r t}{f(\ldots s \ldots) >_r t}$$

$$\frac{f >_\Sigma g \quad f(s_1, \ldots, s_m) >_r t_i \quad i = 1, \ldots, n}{f(s_1, \ldots, s_m) >_r g(t_1, \ldots, t_n)}$$

$$\frac{\begin{array}{c}(s_1, \ldots, s_m) >_r^{lex} (t_1, \ldots, t_m) \\ f(s_1, \ldots, s_m) >_r t_i \quad i = 1, \ldots, m\end{array}}{f(s_1, \ldots, s_m) >_r f(t_1, \ldots, t_m)}$$

Table 4.3: RPO for functions with lexicographic, left-to right status

4.3 Recursive path order is well-founded

We know that RPO is a *simplification order*, *i.e.*, a strict order, closed under context and substitution. We want to show that it is *well-founded* (and therefore a *reduction order*). To this end, we apply the *reducibility candidates method*: a proof technique developed first to prove termination of typed λ-calculi. To simplify the argument, we shall assume that function arguments are always compared with the lexicographic order from left to right. The corresponding specialized definition of RPO is given in Table 4.3.

Definition 78 *We work on the set of terms $T_\Sigma(V)$ and define:*

$$\begin{array}{lcl} WF & = & \{t \in T_\Sigma(V) \mid \text{ there is no infinite sequence } t = t_0 >_r t_1 >_r \cdots\} \ , \\ Red(t) & = & \{s \mid t >_r s\} \ . \end{array}$$

Exercise 79 *Show that:*

1. *$(WF, >_r)$ is a well-founded set.*
2. *If $Red(t) \subseteq WF$ then $t \in WF$.*
3. *If $s \in WF$ and $s >_r t$ then $t \in WF$.*

Let $>_r^{lex}$ be the lexicographic ordered induced by $>_r$ on vectors of n terms in WF. The *key property* follows.

Proposition 80 *If $s_1, \ldots, s_n \in WF$ and $f(s_1, \ldots, s_n) >_r t$ then $t \in WF$.*

Proof. By *induction on the triple*:

$$(f, (s_1, \ldots, s_n), |t|) \ ,$$

with the lexicographic order from left to right where:

- The first component is a function symbol ordered by $>_\Sigma$.
- The third is the size of the term with the usual order on natural numbers.

- For the second, consider the set $\bigcup_{f\in\Sigma} WF^{ar(f)}$ ordered by:

$$(s_1,\ldots,s_n) > (t_1,\ldots,t_m) \text{ iff } n = m \text{ and } (s_1,\ldots,s_n) >_r^{lex} (t_1,\ldots,t_m)\ .$$

  Notice that two vectors of different lengths are *incomparable*. Also, $(WF, >_r)$ well-founded implies $(WF^n, >_r^{lex})$ is well-founded too.

Case $f(s_1,\ldots,s_n) >_r t$ **as** $s_i = t$ **or** $s_i >_r t$**.**

- If $s_i = t$ the conclusion is immediate as $s_i \in WF$ by hypothesis.
- If $s_i >_r t$ then $t \in WF$ as $s_i \in WF$.

Case $t = g(t_1,\ldots,t_m)$**,** $f >_\Sigma g$**,** $f(s_1,\ldots,s_n) >_r t_i$ **for** $i = 1,\ldots,m$**.**

- We notice that $(f,(s_1,\ldots,s_n),|t|) > (f,(s_1,\ldots,s_n),|t_i|)$ for $i = 1,\ldots,m$. Hence, by inductive hypothesis, $t_i \in WF$.
- Suppose $g(t_1,\ldots,t_m) >_r u$. We remark $(f,(s_1\ldots,s_n),|t|) > (g,(t_1,\ldots,t_m),|u|)$. Hence, by inductive hypothesis, $u \in WF$, and by exercise 79, $g(t_1,\ldots,t_m) \in WF$.

Case $t = f(t_1,\ldots,t_n)$**,** $f(s_1,\ldots,s_n) >_r t_i$ **for** $i = 1,\ldots,n$**,** $(s_1,\ldots,s_n) >_r^{lex} (t_1,\ldots,t_n)$**.** This case is similar to the previous one.

- We remark that $(f,(s_1,\ldots,s_n),|t|) > (f,(s_1,\ldots,s_n),|t_i|)$ for $i = 1,\ldots,n$. Hence, by inductive hypothesis, $t_i \in WF$.
- Suppose $f(t_1,\ldots,t_n) >_r u$. We notice $(f,(s_1\ldots,s_n),|t|) > (f,(t_1,\ldots,t_n),|u|)$ (second component decreases!). By inductive hypothesis, $u \in WF$, and by exercise 79, $f(t_1,\ldots,t_n) \in WF$. □

Corollary 81 *All terms are in WF.*

Proof. By induction on the structure of the terms. □

4.4 Simplification orders are well-founded

We prove that all simplification orders (in particular RPO) are well-founded. As already mentioned, the proof goes through a classical combinatorial result known as *Kruskal's theorem*. This result concerns a natural binary relation on terms (or labelled trees), known as *homeomorphic embedding*, that we denote $\triangleright$; we also denote with $\trianglerighteq$ the reflexive closure of $\triangleright$. The appearance of the embedding relation is justified by the simple observation that every simplification order contains it.

Kruskal's theorem states that when considering the embedding relation on the collection of terms built out of a finite signature and a finite set of variables there is no infinite descending chain $t_0 \triangleright t_1 \triangleright \cdots$ (the order is well-founded) and moreover it is not possible to find an infinite set of terms which are all incomparable (an infinite anti-chain). Technically, one says that the collection of terms with the embedding relation is a *well partial order* (wpo).

Definition 82 *Let $\rightarrow$ be the TRS induced by the rules:*

$$f(x_1, \ldots, x_n) \rightarrow x_i \text{ for } i = 1, \ldots, n \ .$$

We write $t \trianglerighteq s$, read t embeds s, if $t \stackrel{*}{\rightarrow} s$, i.e., *if we can rewrite t in s in a finite number of steps (possibly 0).*

Example 83 *Here is an example of homeomorphic embedding:*

$$f(f(h(a), h(x)), f(h(x), a)) \trianglerighteq f(f(a, x), x) \ .$$

Exercise 84 *Here is another definition of homeomorphic embedding:*

$$\frac{}{x \trianglerighteq x} \ , \qquad \frac{s_i \trianglerighteq t_i, i = 1, \ldots, n}{f(s_1, \ldots, s_n) \trianglerighteq f(t_1, \ldots, t_n)} \ , \qquad \frac{s_i \trianglerighteq t \text{ for some } i}{f(s_1, \ldots, s_n) \trianglerighteq t} \ .$$

Check that this definition is equivalent to the previous one.

Exercise 85 *Show that if $>$ is a simplification order and $\geq$ is its reflexive closure then $t \trianglerighteq s$ implies $t \geq s$ (in other terms, if $t \trianglerighteq s$ and $t \neq s$ then $t > s$).*

Exercise 86 *We consider a relatively simple situation, known as Dickson's lemma, where we have a well-founded order and moreover all sets of incomparable elements are finite. Consider the* product order $\geq$ *on $\mathbf{N}^k$ (vectors of natural numbers):*

$$(n_1, \ldots, n_k) \geq (m_1, \ldots, m_k) \text{ if } n_i \geq m_i, i = 1, \ldots, k \ .$$

1. *Show that $>$ (the strict part of $\geq$) is well-founded.*
2. *Show by induction on k, that from every sequence $\{v_n\}_{n \in \mathbf{N}}$ in $\mathbf{N}^k$ we can extract a* growing subsequence, *namely there is an injective function $\sigma : \mathbf{N} \rightarrow \mathbf{N}$ such that:*

$$\forall n \ \ v_{\sigma(n+1)} \geq v_{\sigma(n)} \ .$$

3. *Show that every set of incomparable elements in $\mathbf{N}^k$ (an* anti-chain*) is finite.*

Definition 87 *A* well partial order *$(A, >)$ is a strict ($\forall a \ \ a \not> a$) partial order such that for any sequence $\{a_i \mid i \in \mathbf{N}\}$ in A,*

$$\exists i, j \in \mathbf{N} \ \ j > i \text{ and } a_j \geq a_i \ .$$

Such a sequence is called good. *Otherwise, we call the sequence* bad. *This means:*

$$\forall i, j \in \mathbf{N} \ \ (j > i \text{ implies } a_j \not\geq a_i) \ .$$

Note that if a sequence is bad all its subsequences are.

Remark 88 *In this chapter a partial order by default is* strict. *The reflexive closure of a well partial order is called a* well quasi-ordering *(wqo).*

Proposition 89 *Well partial orders are the* well-founded orders *that have* no infinite anti-chain.

PROOF. ($\Rightarrow$) A wpo must be well-founded for a strictly descending chain gives a bad sequence. For the same reason, a wpo cannot contain an infinite anti-chain.

($\Leftarrow$) Vice versa, take a well-founded set without infinite anti-chain. Given an infinite sequence, the set of minimal elements of the sequence must be finite. Therefore there is a minimal element such that the sequence is infinitely often above it. □

Proposition 90 *Given a sequence in a wpo, it is always possible to extract an ascending subsequence.*

PROOF. Consider a sequence $\{a_i\}_{i\in\mathbf{N}}$. We want to show that there is an *ascending subsequence*:

$$i_1 < i_2 < i_3 < \ldots \text{ and } a_{i_1} \leq a_{i_2} \leq a_{i_3} \leq \cdots$$

We notice that in a good sequence there are *finitely many* a_i such that $\forall\, j\ \ j > i$ implies $a_j \not\geq a_i$. Otherwise, the sequence composed of all such elements is *bad*. Thus starting from a certain point i_1, if $i \geq i_1$ then $\exists\, j > i\ \ a_j \geq a_i$. Now starting from i_1 we can inductively build a sequence $i_1 < i_2 < \ldots$ such that $a_{i_1} \leq a_{i_2} \leq \cdots$ □

Proposition 91 *The* product $A \times B$ *of wpo's* A, B*, ordered component-wise (product order) is a wpo.*

PROOF. Consider $\{(a_i, b_i) \mid i \in \mathbf{N}\}$ and suppose $\{a_i \mid i \in \mathbf{N}\}$ and $\{b_i \mid i \in \mathbf{N}\}$ are both infinite (otherwise it is easy). Then consider the subsequence $i_0 < i_1 < i_2 < \cdots$ such that $a_{i_0} \leq a_{i_1} \leq a_{i_2} \leq \cdots$ (cf. previous proposition 90). Then find $k > l$ such that $b_{i_k} \geq b_{i_l}$. □

Recall that $\trianglerighteq$ is the *homeomorphic embedding* and that the *strict part* of $\trianglerighteq$, say $\triangleright$, is contained in every simplification order.

Proposition 92 (Kruskal) *Suppose* Σ *and* V finite*. Then the* strict homeomorphic embedding $\triangleright$ *on* $T_\Sigma(V)$ *is a* well partial order*.*

PROOF. We pause to notice that if Σ or V are *infinite* then $(T_\Sigma(V), \triangleright)$ contains an *infinite anti-chain* and the proposition does *not* hold. The proof proceeds by *contradiction*. Suppose there is a *bad sequence* in $T_\Sigma(V)$. Extract from the bad sequence a *minimal* one with respect to the size of the terms, say $t_1, t_2, \ldots$ This means that having built the sequence $t_1, \ldots, t_i$, we pick a term t_{i+1} of minimal size among those that follow t_i. Define:

$$S_i = \left\{ \begin{array}{ll} \emptyset & \text{if } t_i \text{ variable} \\ \{s_1, \ldots, s_n\} & \text{if } t_i = f(s_1, \ldots, s_n) \end{array} \right. \Bigg| \; S = \textstyle\bigcup_{i\geq 0} S_i \ .$$

For the time being, assume $(S, \triangleright)$ is a *wpo*; this is a tricky point whose proof is postponed. Since Σ and X are *finite*, there must be a symbol that occurs infinitely often as the root of the minimal bad sequence $t_1, t_2, \ldots$ If it is a variable or a constant we derive a contradiction. Otherwise, we have $i_0 < i_1 < \ldots$ with:

$$t_{i_k} = f(s_1^{i_k}, \ldots, s_n^{i_k}) \ .$$

Now $(S, \triangleright)$ is a wpo and the product of wpo's is a wpo (proposition 91). Therefore, the sequence:

$$\{(s_1^{i_k}, \ldots, s_n^{i_k})\}_{k\geq 0}$$

is *good*. So $\exists\, p, q \;\; q > p$ and $s_l^{i_q} \trianglerighteq s_l^{i_p}, l = 1, \ldots, n$. And this entails $t_{i_q} \trianglerighteq t_{i_p}$. Contradiction!

We now come back to the tricky point. Suppose $(S, \triangleright)$ is *not* a wpo, and let $s_1, s_2, s_3 \ldots$ be a *bad* sequence. The s_i must be all distinct. Suppose $s_1 \in S_k$. This entails $t_k \triangleright s_1$. Let $S_{<k} = S_1 \cup \cdots \cup S_{k-1}$. There is an index l such that $s_i \notin S_{<k}$ for $i \geq l$. Consider:

$$t_1, \ldots, t_{k-1}, s_1, s_l, s_{l+1}, \ldots$$

Since s_1 is smaller than t_k, *by minimality* (!) this sequence must be *good*. Since $t_1, t_2, \ldots$ and $s_1, s_2, \ldots$ are *bad*, this entails $s_j \trianglerighteq t_i$ for some $i \in \{1, \ldots, k-1\}$ and $j \in \{1, l, l+1, \ldots\}$. We distinguish two cases, both leading to a contradiction.

$j = 1$ $t_k \triangleright s_j \trianglerighteq t_i$. Contradiction!

$j \geq l$ Suppose $s_j \in S_m \backslash S_{<k}$. Thus $m \geq k > i$ and $t_m \triangleright s_j \trianglerighteq t_i$. Contradiction! □

Remark 93 *The presented result is an interesting case study for logicians. First, the proof we have presented is* non-constructive *(two nested arguments by contradiction). The literature contains proposals for constructive versions of the proof. Second, the theorem is a simple example of a combinatorial statement that* cannot be proved *in Peano's Arithmetic (a standard formalization of arithmetic in first-order logic).*

Exercise 94 *The following is a special case of Kruskal's theorem on* words *known as* Higman's lemma. *Let Σ be a finite set (alphabet). Given two words $w, w' \in \Sigma^*$ we say that w' is a* subsequence *of w, and write $w > w'$, if the word w' can be obtained from the word w by erasing some (at least one) of its characters. Apply Kruskal's theorem to conclude that $>$ is a well partial order.*

Incidentally, there is also a famous generalization of Kruskal's theorem to *graphs* known as the *graph minor theorem*. An *edge contraction* of a graph consists in removing an edge while merging the two vertices. A graph G is a *minor* of the graph H if it can be obtained from H by a sequence of edge contractions. It turns out that the *minor relation* is a *well partial order*.

Next, we present two relevant applications of Kruskal's theorem to the termination problem of TRS. The first one is another proof that RPO is well-founded.

Proposition 95 *Every* simplification order *on $T_\Sigma(V)$ is* well founded. *Hence, every* simplification order *(in particular RPO) is a* reduction order.

Proof. By *contradiction*, suppose $t_1 > t_2 > \ldots$ First, we prove by contradiction that $\mathsf{var}(t_1) \supseteq \mathsf{var}(t_2) \supseteq \ldots$ Suppose $x \in \mathsf{var}(t_{i+1}) \backslash \mathsf{var}(t_i)$ and consider the *substitution* $S = [t_i/x]$. Then $St_i = t_i > St_{i+1}$. By the *subterm property*, we have $St_{i+1} \geq t_i$. Thus $t_i > t_i$ which *contradicts* the hypothesis that $>$ is strict.

Thus we can take $X = \mathsf{var}(t_1)$ which is *finite*. Then we can *apply Kruskal's theorem* to the sequence and conclude:

$$\exists\, i, j \;\; j > i \text{ and } t_j \trianglerighteq t_i \;.$$

Thus we have both $t_i > t_j$ and $t_j \geq t_i$. Hence, $t_i > t_i$, which contradicts the hypothesis that a simplification order is strict. Notice that we used the fact that $(T_\Sigma(X), \triangleright)$ is a *wpo*, not just a well-founded set. □

The second application concerns the introduction of the interpretation method over the *reals*. Take as domain $A = \{r \in \mathbb{R}^+ \mid r \geq a \geq 1\}$. This may appear as a wrong start as the set A is *not* well founded! However, suppose that we associate with every $f^n \in \Sigma$ a multivariate polynomial $p_f(x_1, \ldots, x_n)$ such that:

1. Coefficients range over the *non-negative reals*.
2. $p_f(a, \ldots, a) \in A$: thus p_f defines a function over the domain A.
3. $p_f(a_1, \ldots, a_n) > a_i$ for $i = 1, \ldots, n$ (the new condition!).

Write $s >_A t$ if over the domain A the polynomial associated with the term s is strictly larger than the one associated with the term t. It is easily checked that this is a *simplification order*, hence a *reduction order*. The fact that we move from integers to real has an interesting consequence as the first-order theory of reals is *decidable* (*e.g.*, it is decidable whether a first-order assertion in analytic geometry is valid).

A corollary of this result is that we can decide whether there is a polynomial interpretation over the reals where the polynomials have a *bounded degree*. However, we stress that this is a rather *theoretical advantage* because of the *high complexity* of the decision procedures.

4.5 Summary and references

Proving the termination of a TRS amounts to find a reduction order that is compatible with the rules of the TRS. One method consists in interpreting the function symbols as functions over the positive integers with certain strictness properties. Another method consists in applying the rules of the recursive path orders. It turns out that the recursive path orders are an instance of the simplification orders and that the latter are well-founded. Many other methods for proving termination of TRS have been proposed and their implementation is available in several tools. Recursive path orders were introduced in [Der82] and the presented termination proof is based on [vR01]. Kruskal's tree theorem is in [Kru60] with a shorter proof in [NW63] which is the one we present. Its special case for words is presented in [Hig52] and its generalization to graphs is presented in a long series of papers starting with [RS83]. Hilbert's 10^{th} problem was among 23 open problems put forward at a 1900 international conference on mathematics. The problem was eventually shown to be undecidable by Matiyasevich in 1970. The decidability of the first-order theory of real numbers was shown by Tarski around 1950.

Chapter 5

Confluence and completion of term rewriting systems

In general the confluence of a term rewriting system (TRS for short) is an *undecidable* property. However, if the TRS is *terminating and finite* then the property is *decidable*. By proposition 47, we know that checking *local confluence* is enough and it turns out that to do that it is enough to consider a *finite* number of cases known as *critical pairs*.

5.1 Confluence of terminating term rewriting systems

The definition of critical pair captures the most general way in which two term rewriting rules can superpose and thus possibly compromise the local confluence of the TRS.

Definition 96 *Let $l_i \to r_i$ for $i = 1, 2$ be two rules of the TRS (possibly equal) and assume that the variables in each rule are* renamed *so that $\mathsf{var}(l_1) \cap \mathsf{var}(l_2) = \emptyset$. Further suppose $l_1 = C[l_1']$ where l_1' is* not *a variable and let S be the* most general unifier *of l_1' and l_2 (if it exists). Then $(S(r_1), S(C[r_2]))$ is a* critical pair.

The pair $(S(r_1), S(C[r_2]))$ is *critical* because if we take the terms l_1 and $C[l_2]$ then:

$$S(r_1) \leftarrow S(l_1) = S(C)[S(l_1')] = S(C)[S(l_2)] \to S(C)[S(r_2)] = S(C[r_2]) \ .$$

Thus the two terms in the critical pair must be joinable (have a common reduct). The main insight is that this is enough to guarantee (local) confluence. Let us start with a preliminary remark. Let the *domain* of a substitution S be the set:

$$dom(S) = \{x \in V \mid S(x) \neq x\} \ .$$

Given two substitutions S_1, S_2 let us define their *union* as follows:

$$(S_1 \cup S_2)(x) = \begin{cases} S_1(x) & \text{if } x \in dom(S_1) \backslash dom(S_2) \\ S_2(x) & \text{if } x \in dom(S_2) \backslash dom(S_1) \\ x & \text{otherwise.} \end{cases}$$

Exercise 97 *Suppose $\mathsf{var}(t) \cap \mathsf{var}(s) = \emptyset$, $dom(S_1) \subseteq \mathsf{var}(t)$, and $dom(S_2) \subseteq \mathsf{var}(s)$. Then show that $S_1(t) = S_2(s)$ entails that $S_1 \cup S_2 \models t = s$. Notice that if $\mathsf{var}(t) \cap \mathsf{var}(s) \neq \emptyset$ then the assertion is false. E.g., take: $t = x$, $s = f(x)$, $S_1 = [f(x)/x]$, $S_2 = id$, $S_1 \cup S_2 = S_1$ but $S_1(t) \neq S_1(s)$.*

Proposition 98 *Suppose given a* finite *and* terminating *TRS. Then the TRS is* confluent *iff all the* critical pairs *induced by its rules are* joinable *(and the latter is a* decidable *condition).*

PROOF. The test is *necessary* as explained above. Because the TRS is terminating, it is enough to show that the test guarantees *local confluence*. To check local confluence a *finite case analysis* suffices. If $s \to t_1$ and $s \to t_2$, then we can find rules $l_1 \to r_1, l_2 \to r_2$, contexts C_1, C_2 and substitutions S_1, S_2 such that

$$s = C_1[S_1 l_1] = C_2[S_2 l_2], \quad t_1 = C_1[S_1 r_1], \quad t_2 = C_2[S_2 r_2] \ .$$

We sketch and provide concrete examples for the main cases to consider.

Case 1 The paths corresponding to the contexts C_1 and C_2 are incomparable (neither is a prefix of the other). In this case one can close the diagram in one step. For instance, assume the rules:

$$g_i(x) \to k_i(x), \quad i = 1, 2 \ ,$$

and consider $h(g_1(x), g_2(x))$.

Case 2 There is a variable x in l_1 such that $S_2 l_2$ is actually a subterm of $S_1(x)$. In this case one can always close the diagram, though it may take *several* steps. For instance, assume the rules:

$$f(x, x, x) \to h(x, x), \quad g(x) \to k(x) \ .$$

and consider $f(g(x), g(x), g(x))$.

Case 3 We can decompose l_1 in $C[l_1']$ so that:

$$l_1' \text{ is not a variable and } S_1 l_1' = S_2 l_2 \ .$$

One can show that this situation is always an instance of a *critical pair*. For instance, assume the rules:

$$f(f(x, y), z) \to f(x, f(y, z)), \quad f(i(x), x) \to e \ ,$$

and consider $f(f(i(x), x), z)$. □

Exercise 99 *Consider the TRS:*

$$f(x, g(y, z)) \ \to \ g(f(x, y), f(x, z)) \ , \qquad g(g(x, y), z) \ \to \ g(x, g(y, z)) \ .$$

Is the resulting reduction system terminating and/or confluent?

5.2 Completion of term rewriting systems

The test for local confluence is the basis for an iterative symbolic computation method known as *Knuth-Bendix completion*. Given an equational theory, the *goal* is to obtain a confluent and terminating term rewriting system for it. The main steps in Knuth-Bendix completion are as follows::

1. Orient the equations thus obtaining a TRS.

2. Check *termination* of the TRS.

3. Then check local *confluence*.

4. If a critical pair *cannot be joined*, then we add the corresponding equation and we repeat the process.

Notice that there is no guarantee that the process terminates! At various places, one may require a human intervention: orientation of the rules, well-founded order to check termination, selection of the rules to add,...

Example 100 *The following law describes so called 'central grupoids':*

$$(x * y) * (y * z) = y \ .$$

Any simplification ordering $>$ *will satisfy:*

$$(x * y) * (y * z) > y \ ,$$

so we orient the equation from left to right. A critical pair *is:*

$$((x' * y') * (y' * z')) * ((y' * z') * z) \to (y' * z'), \quad y' * ((y' * z') * z) \ .$$

Any simplification ordering *satisfies:* $y' * ((y' * z') * z) > (y' * z')$. *Another critical pair is:*

$$(x * (x' * y')) * ((x' * y') * (y' * z')) \to (x' * y'), \quad (x * (x' * y')) * y' \ .$$

Again any simplification ordering *satisfies:*

$$(x * (x' * y')) * y' > (x' * y') \ .$$

Thus we get a terminating TRS with three rules. In the next iteration all critical pairs turn out to be joinable and thus the completion terminates successfully.

Example 101 *The equations for left/right distributivity of* $*$ *over* $+$ *are:*

$$x * (y + z) \ = \ (x * y) + (x * z) \ , \qquad (u + v) * w \ = \ (u * w) + (v * w) \ .$$

Ordering from left to right, a critical pair *is:*

$$(u + v) * (y + z) \to ((u + v) * y) + ((u + v) * z), \quad (u * (y + z)) + (v * (y + z)) \ .$$

If we normalize the two terms on the left-hand-side we get:

$$((u * y) + (v * y)) + ((u * z) + (v * z), \quad ((u * y) + (u * z)) + ((v * y) + (v * z)) \ ,$$

and there is no reasonable way to order them.

Example 102 *Consider the equations:*

$$x + \mathsf{z} = x, \quad \mathsf{s}(x + y) = x + \mathsf{s}(y), \quad x + \mathsf{s}(\mathsf{z}) = \mathsf{s}(x) \ .$$

It can be easily checked that by orienting them from left to right we obtain a terminating TRS. However, there is a critical pair between the second and third rule taking:

$$\mathsf{s}(\mathsf{s}(x)) \leftarrow \mathsf{s}(x + \mathsf{s}(\mathsf{z})) \rightarrow x + \mathsf{s}(\mathsf{s}(\mathsf{z})) \ .$$

In turn this forces the rule: $x + \mathsf{s}(\mathsf{s}(\mathsf{z})) \rightarrow \mathsf{s}(\mathsf{s}(x))$. *In this case, a simple completion method* may diverge *as one has to add all the rules of the shape:*

$$x + \mathsf{s}^n(\mathsf{z}) \rightarrow \mathsf{s}^n(x) \ .$$

However, an alternative completion strategy succeeds by orienting the second rule in the opposite direction: $x + \mathsf{s}(y) \rightarrow \mathsf{s}(x + y)$.

Exercise 103 *Prove termination and confluence of the TRS considered in examples 53 and 54.*

Exercise 104 *Consider the TRS:*

$$f(f(x, y), z) \ \rightarrow \ f(x, f(y, z)), \qquad f(i(x), x) \ \rightarrow \ e \ .$$

(1) Can you show termination by RPO? (2) Can you show termination by polynomial interpretation? (3) Is the system confluent?

Now consider the TRS:

$$f(f(x)) \rightarrow g(x) \ .$$

(4) Is it confluent? (5) Add the rule $f(g(x)) \rightarrow g(f(x))$. *Is this terminating by RPO ? (6) And by polynomial interpretation? (7) Is the system confluent? (8) Same questions if we add the rule* $g(f(x)) \rightarrow f(g(x))$.

Exercise 105 *Let* R_1 *be a TRS with rules:*

$$f(a, b, x) \rightarrow f(x, x, x), \quad x \text{ variable},$$

and let R_2 *be another TRS with rules:*

$$g(x, y) \ \rightarrow \ x, \qquad g(x, y) \ \rightarrow \ y, \qquad x, y \text{ variables.}$$

(1) Show that the systems R_1 *and* R_2 *terminate. (2) Prove or give a counter-example to the confluence of the TRS* R_1 *and* R_2. *(3) Show that the TRS* $R_1 \cup R_2$ *does* not *terminate. (4) However, show that the TRS* $R_1 \cup R_2$ *is normalizing (every term has a normal form).*

5.3 Summary and references

The *critical pair test* is a practical test to check the *(local) confluence* of TRS and the basis of an iterative method known as *Knuth-Bendix completion* [KB70]. The method starts with a TRS which is typically derived from a set of equations. It then checks the TRS for termination and local confluence. If local confluence fails, then we try to orient the critical pairs and start the verification again. Many sophisticated *refinements* of the completion procedure have been proposed and implemented in a variety of tools. Also similar ideas have been developed in parallel and independently in the area of computer algebra where a technique known as *Gröbner bases* is used to solve decision problems in rings of polynomials. Examples in this chapter are based on [BN99] and the reader is invited to check them with one of the tools available online.

Chapter 6

Term rewriting systems as functional programs

We focus on term rewriting systems whose symbols can be partitioned in *constructors* and *functions* and whose term rewriting rules guarantee a deterministic evaluation. Such systems can be regarded as rudimentary *first-order* functional programs. We then consider a recursive definition mechanism known as *primitive recursion* which guarantees termination. Going beyond termination, we present conditions that guarantee *termination in polynomial time*. More precisely, we define a restricted form of primitive recursion on binary words, also known as *bounded recursion on notation*, in which one can program exactly the *functions* computable in polynomial time.

6.1 A class of term rewriting systems

We consider term rewriting systems whose signature Σ is partitioned into *constructor symbols* denoted with $\mathsf{c}, \mathsf{d}, \ldots$ and *function symbols* denoted with $f, g, \ldots$ A *value* is a term composed of constructor symbols and denoted with $v, v', \ldots$ while a *pattern* is a term composed of constructor symbols and variables and denoted with $p, p', \ldots$ To make sure the collection of values is not empty, we assume that there is at least a *constant* (a symbol with arity 0) among the constructors. We assume all term rewriting rules have the shape:

$$f(p_1, \ldots, p_n) \to e \ .$$

Moreover given two distinct rules:

$$f(p_1, \ldots, p_n) \to e \ , \qquad f(p'_1, \ldots, p'_n) \to e \ ,$$

which refer to the same function symbol f, we assume that they cannot superpose, *i.e.*, it is *not* possible to find values $v_1, \ldots, v_n$ and substitutions S and S' such that $v_i = S(p_i) = S'(p'_i)$ for $i = 1, \ldots, n$. Under these hypotheses, closed terms are evaluated according to the following rules:

$$\frac{e_j \Downarrow v_j \quad j = 1, \ldots, n}{\mathsf{c}(e_1, \ldots, e_n) \Downarrow \mathsf{c}(v_1 \ldots, v_n)} \qquad \frac{\begin{array}{c} e_j \Downarrow v_j, \ \ f(p_1, \ldots, p_n) \to e, \\ Sp_j = v_j, j = 1, \ldots, n, \quad S(e) \Downarrow v \end{array}}{f(e_1, \ldots, e_n) \Downarrow v} \ .$$

Notice that the first rule guarantees that for all values v, we have: $v \Downarrow v$. If the term is not a value, then we look for the innermost-leftmost term of the shape $f(v_1, \ldots, v_n)$ and look for a rule $f(p_1, \ldots, p_n) \to e$ which applies to it (by hypothesis, there is at most one). If no rule applies then the evaluation is stuck. If v is a value its size $|v|$ is a natural number defined by:

$$|c(v_1, \ldots, v_n)| = 1 + \Sigma_{i=1,\ldots,n}|v_i| \ .$$

By extension, if $t \Downarrow v$ then $|t| = |v|$. Thus, in this chapter, the notation $|t|$ denotes the size of the unique value to which the closed term t evaluates (if any).

Example 106 *We introduce some* constructors *along with their arity.*

$\mathsf{t}^0, \mathsf{f}^0$	*(boolean values)*
$\mathsf{z}^0, \mathsf{s}^1$	*(tally (unary) natural numbers)*
$\mathsf{nil}^0, \mathsf{c}^1$	*(lists)*
$\epsilon^0, \mathsf{0}^1, \mathsf{1}^1$	*(binary words)*

and some functions*:*

$ite(\mathsf{t}, x, y)$	$\to$	x	*(if-then-else)*
$ite(\mathsf{f}, x, y)$	$\to$	y	
$leq(\mathsf{z}, y)$	$\to$	t	*(less-equal)*
$leq(\mathsf{s}(x), \mathsf{z})$	$\to$	f	
$leq(\mathsf{s}(x), \mathsf{s}(y))$	$\to$	$leq(x, y)$	
$ins(x, \mathsf{nil})$	$\to$	$\mathsf{c}(x, \mathsf{nil})$	*(list insertion)*
$ins(x, \mathsf{c}(y, l))$	$\to$	$ite(leq(x, y), \mathsf{c}(x, \mathsf{c}(y, l)), \mathsf{c}(y, ins(x, l)))$	

Exercise 107 *Continue the previous example by defining functions to sort lists of tally natural number according to various standard algorithms such as insertion sort, quick sort,...*

Notice that it is straightforward to program the functions above in a language with pattern-matching such as ML.

6.2 Primitive recursion

We restrict further the class of term rewriting systems so that termination is guaranteed. Assume the following constructor symbols for tally natural numbers: z^0 and s^1. Also assume the following *basic function symbols* z, s, p_i^n for $i = 1, \ldots, n$ with the following rules:

$z(x)$	$\to$	z	(zero)
$s(x)$	$\to$	$\mathsf{s}(x)$	(successor)
$p_i^n(x_1, \ldots, x_n)$	$\to$	x_i	(projections).

New function symbols can be introduced according to the composition and primitive recursion rules which are described below. We shall use x^* to denote a (possibly empty) sequence of variables $x_1, \ldots, x_n$.

Composition Given g of arity k and h_i of arity n for $i = 1, \ldots, k$ introduce a new function f of arity n with the rule:

$$f(x^*) \to g(h_1(x^*), \ldots, h_k(x^*)) \ .$$

Primitive Recursion Given g of arity n and h of arity $n+2$ introduce a new function f of arity $n+1$ with the rules:

$$\begin{array}{lcl} f(\mathsf{z}, y^*) & \to & g(y^*) \\ f(\mathsf{s}(x), y^*) & \to & h(f(x,y), x, y^*) \ . \end{array}$$

Example 108 *We practice primitive recursion by defining a few arithmetic function.*

$$\begin{array}{lcll} add(\mathsf{z}, y) & \to & y & \textit{(addition)} \\ add(\mathsf{s}(x), y) & \to & \mathsf{s}(add(x,y)) & \\ mul(\mathsf{z}, y) & \to & \mathsf{z} & \textit{(multiplication)} \\ mul(\mathsf{s}(x), y) & \to & add(mul(x,y), y) & \\ exp(x, \mathsf{z}) & \to & \mathsf{s}(\mathsf{z}) & \textit{(exponentiation)} \\ exp(x, \mathsf{s}(n)) & \to & mul(exp(x,n), x) \ . & \end{array}$$

We can go on to describe towers of exponentials,... The complexity of the programmable functions is still very high!

Exercise 109 *Define primitive recursive functions to: (1)* decrement *by one (with* $0-1=0$*), (2)* subtract *(with* $x-y=0$ *if* $y>x$*), (3) compute an* if-then-else*, (4) compute the* minimum *of two numbers.*

We notice that there is a *trade-off* between primitive recursion and full recursion. Namely, in the former *termination* is for free but some *functions* cannot be represented and some *algorithms* are more difficult or impossible to represent. For instance, it can be shown by a diagonalization argument that the universal function (the interpreter) for primitive recursive functions is a total function but not a primitive recursive one. It can also be shown that the *natural* algorithm that computes the minimum of two tally natural number cannot be expressed by primitive recursion (details in the following example).

Example 110 *Primitive recursion is a bit of a straight-jacket to guarantee termination. For instance, the following rules could be used to define the minimum of two tally natural numbers.*

$$\begin{array}{lcl} min(\mathsf{z}, y) & \to & \mathsf{z} \\ min(\mathsf{s}(x), \mathsf{z}) & \to & \mathsf{z} \\ min(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(min(x,y)) \ . \end{array}$$

The rules scan the two numbers in parallel and stop as soon as they reach the end of the smallest one. However this definition of min is not primitive recursive. Worse, it can be shown that no primitive recursive definition of min produces an algorithm computing $min(v,u)$ *in time* $min(|v|,|u|)$*.*

We are soon going to address complexity issues and it is well known that in this case unary notation is rather odd. Indeed unary notation requires *too much space* and because

representation of the input is so *large* complexities of the operations can be unexpectedly *low*. For instance, in unary notation we can compute the addition in constant time: it is enough to regard numbers as lists and concatenate them. So we revise the notion of primitive recursion by working with the following constructors which correspond to binary words: $\epsilon^0, \mathsf{0}^1, \mathsf{1}^1$. The *basic functions* are now e^1, s_i^1 for $i = 0, 1$, and p_i^n for $i = 1, \ldots, n$ with the following rules:

$$\begin{array}{lcll} e(x) & \to & \epsilon & \text{(empty word)} \\ s_i(x) & \to & \mathsf{i}x & \text{(successors)} \\ p_i(x_1, \ldots, x_n) & \to & x_i & \text{(projections).} \end{array}$$

As before, we can introduce new functions according to two mechanisms.

Composition Given g of arity k and h_i of arity n for $i = 1, \ldots, k$ we introduce a new function symbol f with the rule:

$$f(x^*) \to g(h_1(x^*), \ldots, h_k(x^*)) \ .$$

Primitive recursion Given g of arity n and h_i of arity $n+2$ for $i = 0, 1$ we introduce a new function symbol f with the rules:

$$\begin{array}{lcl} f(\epsilon, y^*) & \to & g(y^*) \\ f(\mathsf{0}(x), y^*) & \to & h_0(f(x, y^*), x, y^*) \\ f(\mathsf{1}(x), y^*) & \to & h_1(f(x, y^*), x, y^*) \ . \end{array}$$

The class of functions definable in this way are the primitive recursive functions on binary notation, also known as functions defined by *recursion on notation*.

Exercise 111 *Assume binary numbers are represented as binary words where the* least significant digit is on the left*. We consider the problem of defining some standard arithmetic functions by primitive recursion on binary words.*

1. *Define a function that takes a binary word and removes all* '0' *that do not occur on the left of a* 1 *(hence* ϵ *can be taken as the* canonical representation of zero*).*

2. *Show that the functions* division by 2*,* modulo 2*,* successor*,* if-then-else*,* predecessor*,* number of digits *can be defined by primitive recursion.*

3. *Suppose a function that implements* addition *is given (known definitions of this function are quite technical). Implement multiplication by primitive recursion on binary notation.*

6.3 Functional programs computing in polynomial time

How can we compute a function defined by primitive recursion? Suppose $v_k = \mathsf{i}_k \ldots \mathsf{i}_1 \epsilon$. Here is a simple loop that computes $f(v_k, v^*)$:

$$\begin{array}{l} r = g(v^*); \\ \mathsf{for}\ (j = 1; j \leq k; j = j + 1)\{r = h_{i_j}(r, v_{j-1}, v^*);\} \\ \mathsf{return}\ r \ . \end{array}$$

Problem: suppose that h_i and g can be computed in *polynomial time*. Can we conclude that f can be computed in *polynomial time*? Well, here is what can go wrong. Consider first the function d doubling the size of its input:

$$d(\epsilon) \rightarrow 1\epsilon \ , \qquad d(\mathsf{i}(x)) \rightarrow \mathsf{i}(\mathsf{i}(d(x))) \quad \mathsf{i} = 0, 1 \ .$$

Then consider the function e:

$$e(\epsilon) \rightarrow 1\epsilon \ , \qquad e(\mathsf{i}(x)) \rightarrow d(e(x)) \quad \mathsf{i} = 0, 1 \ .$$

These functions are definable by primitive recursion on binary notation (exercise!) and $|e(v)|$ is exponential in $|v|$. Iterating $|v|$ times polynomial time operations can generate data whose size is *not* polynomial in $|v|$.

We now introduce a notion of definition by bounded recursion on notation (BRN). This is an ordinary primitive recursion on binary words:

$$\begin{array}{lcl} f(\epsilon, y^*) & \rightarrow & g(y^*) \\ f(0(x), y^*) & \rightarrow & h_0(f(x, y^*), x, y^*) \\ f(1(x), y^*) & \rightarrow & h_1(f(x, y^*), x, y^*) \ , \end{array}$$

with the additional requirement that there exists a *polynomial* S_f with non-negative coefficients such that:

$$|f(v, v^*)| \leq S_f(|v|, |v^*|) \ .$$

It turns out that the *functions* computable by an algorithm in BRN are exactly those computable in PTIME. This result decomposes in the following two propositions.

Proposition 112 *If we can define an* algorithm by BRN *then we can compute its result in* PTIME.

Proof. First we prove by induction on the definition of a function f in BRN that there is a polynomial S_f such that for all $v_1, \ldots, v_n$:

$$|f(v_1, \ldots, v_n)| \leq S_f(|v_1|, \ldots, |v_n|) \ .$$

This is clear for the *basic functions* and for *BRN*. For the *composition*, say h, of f with $(g_1, \ldots, g_k)$ we have:

$$h(v^*) \rightarrow f(g_1(v^*), \ldots, g_k(v^*)) \ .$$

Then by inductive hypothesis:

$$|g_i(v^*)| \leq S_{g_i}(|v^*|) \ .$$

Let S be a polynomial that bounds all S_{g_i}. Applying again the inductive hypothesis:

$$\begin{array}{lcl} |f(g_1(v^*), \ldots, g_k(v^*))| & \leq & S_f(|g_1(v^*)|, \ldots, |g_k(v^*)|) \\ & \leq & S_f(S_{g_1}(|v^*|), \ldots, S_{g_k}(|v^*|)) \\ & \leq & S_f(S(|v^*|), \ldots, S(|v^*|)) \ , \end{array}$$

and the composition of polynomials is a polynomial. Thus data computed by BRN has *size polynomial in the size of the input*.

Next, we prove by induction on the definition of a function f in BRN that there is a polynomial T_f such that $f(v_1, \ldots, v_n)$ can be computed in time $T_f(|v_1|, \ldots, |v_n|)$. *Recursion*

is the interesting case. Consider again the loop computing primitive recursion, where $v_j = \mathsf{i}_j \cdots \mathsf{i}_1\epsilon$, $1 \leq j \leq k$:

$$\begin{array}{l} r = g(v^*); \\ \mathsf{for}\ (j = 1; j \leq k; j = j + 1)\{r = h_{i_j}(r, v_j, v^*);\} \\ \mathsf{return}\ r\ . \end{array}$$

For all steps j, we have that: $|r| \leq S_f(k, |v^*|)$. Let T_h be a polynomial that bounds both T_{h_0} and T_{h_1}. Then the computation of the k steps is performed in at most:

$$k \cdot T_h(S_f(k, |v^*|), k, |v^*|) = |v_k| \cdot T_h(S_f(|v_k|, |v^*|), |v_k|, |v^*|)\ ,$$

which is a polynomial in $|v_k|, |v^*|$. □

Proposition 113 *If there is a* PTIME algorithm *in some Turing-equivalent formalism then we can compile it to an* algorithm in BRN that computes the same function.

Proof. Let $M = (\Sigma, Q, q_o, F, \delta)$ be a *Turing machine* (TM) with: (i) Σ alphabet, (ii) Q states, $q_o \in Q$ initial state, (iii) $F \subseteq Q$ final states, and (iv) $\delta : \Sigma \times Q \to \Sigma \times Q \times \{L, R\}$ transition function.

If x is a real number let $\lceil x \rceil$ be the least integer n such that $x \leq n$. Obviously, elements in Σ and Q can be encoded as *binary words* of length $\lceil log_2(\sharp\Sigma) \rceil$ and $\lceil log_2(\sharp Q) \rceil$, respectively. The *configuration of a TM* can be described by a tuple (q, h, l, r) where: (i) q is the current *state*, (ii) h is the *character read*, (iii) l are the characters on the *left hand side of the head*, and (iv) r are the characters on the *right hand side of the head*.

Next, we have to define a *step function* that simulates one step of a Turing machine while working on the encodings of states and characters. Informally, the function *step* is a *case analysis* corresponding to the finite table defining the transitions of the TM. *E.g.*, the rule:

$$step(\mathsf{01}\epsilon, \mathsf{1}\epsilon, \mathsf{0}l', r) \to (\mathsf{11}\epsilon, \mathsf{0}\epsilon, l', \mathsf{0}r)\ ,$$

describes the situation where being in state 01 and reading 1, we go in state 11, write 0, and move to the left. The only technical difficulty here is that *tuples* are not a primitive data structures in our formalization. However, using the arithmetic functions, we can program *pairing of natural numbers* and the *related projections*. Alternatively (and more naturally), one could extend the framework with a *pairing* constructor.

We *ignore* these problems and assume a function *step* that takes a tuple (q, h, l, r) and returns the tuple (q', h', l', r') describing the following state. Now comes a *key idea* which we present first using a simplified notation. We have an *initial configuration* v_0 and a function *step* such that:

$$|step(v)| \leq |v| + 1\ .$$

We want to *iterate step* on v_o at least $P(|v_o|)$ times where P is a polynomial of *degree* k. W.l.o.g., we may assume the TM loops after reaching the final state so that running it longer does not hurt.

We assume an *expansion function exp* such that for all k there is an m such that:

$$|exp^m(v)| \geq |v|^k\ .$$

To do this, it is enough to iterate a function that squares the size of its entry. Remember that k and therefore m are *constants*, *i.e.*, they do not depend on the size of the input. Then we define a function *it* as:

$$it(\epsilon, v) \quad \to \quad v \ , \quad it(\mathsf{i} \cdot u, v) \quad \to \quad step(it(u, v)) \quad \mathsf{i} = \mathsf{0}, \mathsf{1} \ . \tag{6.1}$$

This is a definition by *bounded recursion on notation* since assuming $S_{it}(n, m) = n + m$ (a polynomial!) we have:

$$|it(u, v)| \leq S_{it}(|u|, |v|) \ .$$

Then to iterate the *step* function at least $|v_0|^k$ times on the initial configuration we run: $it(exp^m(v_0), v_0)$.

We can now go back to TM. All we have to do is to rewrite the *it* function above (6.1) as follows:

$$it(\epsilon, q, h, l, r) \quad \to \quad (q, h, l, r) \ , \qquad it(\mathsf{i}u, q, h, l, r) \quad \to \quad step(it(u, q, h, l, r)) \quad \mathsf{i} = \mathsf{0}, \mathsf{1} \ .$$

To summarize, given a TM running in P time (P fixed polynomial), for any input v_0 we: (i) initialize a *counter* to a value u such that $|u| \geq P(|v_0|)$ and (ii) perform a BRN on the counter thus iterating the step function $|u|$ times. Notice that here the iteration works on the *length of the counter* and not on its binary representation. Otherwise, the definition would not be by BRN and termination could take exponential time! □

Remark 114 (1) *It is quite possible to program a function that takes* exponential time *and runs in* polynomial space *(never going twice through the same configuration!). For instance, take a function that* counts *from* $\mathsf{0}^n\epsilon$ *to* $\mathsf{1}^n\epsilon$*. Implicitly, proposition 112 states that as long as we stick with BRN such function* cannot *be programmed. In the counting function, the problem is* not *the size of the data (the identity function gives the bound!) but the fact that the recursion mechanism is not compatible with primitive recursion on notation.*

(2) *The proof of proposition 113 suggests that there is a* trivial way *of building PTIME algorithms. Take* any program *and* instrument it *so that it keeps a counter that stops after a number of steps which is polynomial in the size of the input (for a fixed polynomial). Of course, the problem with this 'time-out' approach is that we have no idea whether the program will produce interesting answers* before *running it.*

(3) *The reader should keep in mind that while it is possible to build a programming language (a decidable syntax) that computes exactly the* PTIME functions*, it is not possible to build one that contains exactly the* PTIME programs*,* e.g.*, the set of Turing's machines computing in PTIME is undecidable.*

Proposition 112 restricts the programmer to primitive recursion and it provides no clue on *how to find a polynomial bound* on the size. Is it possible to find a *syntactic criterion* that guarantees the existence of a polynomial bound? The *high-complexity* of programs defined by primitive recursion on binary notation depends on the fact that we have *nested recursions*, *i.e.*, the result of a primitive recursion can be used as the main argument of another primitive recursion as in:

$$\begin{array}{llllll} d(\epsilon) & \to & \mathsf{1}\epsilon \ , & d(\mathsf{i}(x)) & \to & \mathsf{i}(\mathsf{i}(d(x))) \quad \mathsf{i} = \mathsf{0}, \mathsf{1} \ , \\ e(\epsilon) & \to & \mathsf{1}\epsilon \ , & e(\mathsf{i}(x)) & \to & d(e(x)) \quad \mathsf{i} = \mathsf{0}, \mathsf{1} \quad \Leftarrow \quad \text{('Bad' recursion!)} \end{array}$$

A key insight is that if we forbid this by a *syntactic mechanism* then *data size stays polynomial* and moreover it is still possible to define all functions computable in PTIME. To this end, functions' arguments are partitioned into *two zones* (syntactically separated by a semi-colon):

$$f(x_1, \ldots, x_n; y_1, \ldots, y_m) \quad n, m \geq 0 \ .$$

The ones on the left are called *normal* and those on the right *safe*. Let us refer to the functions in this new class as SRN functions. The *invariant* one maintains on SRN functions is that there is a polynomial P_f such that:

$$|f(v_1, \ldots, v_n; u_1, \ldots, u_m)| \leq P_f(|v_1|, \ldots, |v_n|) + \mathsf{max}(|u_1|, \ldots, |u_m|) \ .$$

In particular, if f has no normal arguments then the size of its result is bound by the size of its arguments up to an *additive* constant.

Unlike in BRN, the existence of the polynomial is *guaranteed* by the way recursion and composition are restricted. Assuming g, h_0, h_1 are SRN functions, we can define a new SRN function f with the rules:

$$\begin{array}{lcl} f(\epsilon, x^*; y^*) & \to & g(x^*; y^*) \\ f(\mathsf{0}x, x^*; y^*) & \to & h_0(x, x^*; y^*, f(x, x^*; y^*)) \\ f(\mathsf{1}x, x^*; y^*) & \to & h_1(x, x^*; y^*, f(x, x^*; y^*)) \ . \end{array}$$

The main argument lies in the normal zone (on the left) while the recursive calls take place in the safe zone (on the right). The way SRN functions are composed is also restricted so that expressions plugged in the normal zone do not depend on arguments in the safe zone. Specifically, assuming, $f, g_1, \ldots, g_k, h_1, \ldots, h_l$ are SRN functions we define their safe composition as:

$$f(\ g_1(x^*;), \ldots, g_k(x^*;)\ ;\ h_1(x^*; y^*), \ldots, h_l(x^*; y^*)\ ) \ .$$

Example 115 *Here is an example and a non-example of SRN functions.*

$$\begin{array}{lcl} d(\epsilon;) & \to & \mathsf{1}(;\epsilon) \ , \\ e(\epsilon;) & \to & \mathsf{1}\epsilon \ , \end{array} \qquad \begin{array}{lcll} d(\mathsf{i}(;x);) & \to & \mathsf{i}(;\mathsf{i}(;d(x;))) & \mathsf{i} = \mathsf{0}, \mathsf{1} \ , \\ e(\mathsf{i}(;x);) & \to & d(e(x;)) & \mathsf{i} = \mathsf{0}, \mathsf{1} \end{array} \qquad \Leftarrow \textit{Not SRN!}$$

$e(x;)$ *should go to the safe zone, while* d *is waiting for an argument in the normal zone; nested recursions do* not *compose.*

6.4 Summary and references

Functions defined by primitive recursion on unary or binary notation are guaranteed to terminate; the book [Ros84] is a compact reference on hierarchies of total recursive functions. If moreover, we restrict the size of the computed values to be polynomial in the size of the input then we can program exactly the functions computable in *polynomial time*. This is an early result in complexity theory [Cob64]. The fact that the size bounds can be obtained through a syntactic discipline has been observed more recently in [BC92]. The reader is warned that this syntactic discipline is quite restrictive and hardly practical.

Chapter 7

λ-calculus

The λ-calculus is a compact notation to represent (higher-order) functions. It turns out that this notation embodies directly many concepts arising in programming languages such as: (higher-order) functions, recursive definitions, scoping rules, and evaluation strategies. Moreover, it is sufficiently expressive to describe a number of programming features such as: control flow operators, side-effects, records, and objects which will be discussed in the following chapters. When enriched with types, the terms of the λ-calculus can be regarded as proofs in a (constructive) logic. This connection sheds light on the design of type systems for programming languages and explains the role of the λ-calculus in (higher-order) proof assistants.

In this chapter, we start the technical development by introducing an equational theory on λ-terms known as β-conversion and we prove the confluence of the related reduction rule. We also prove similar results for a stronger theory known as $\beta\eta$-conversion. Next, we show that the λ-calculus is sufficiently expressive to represent partial recursive functions (the λ-calculus is Turing equivalent). Finally we introduce a term rewriting system known as *combinatory logic* which simulates, to some extent, the λ-calculus.

7.1 Syntax

The (type-free) λ-calculus is composed of the λ-terms defined by the following grammar:

$$M ::= id \mid (\lambda id.M) \mid (MM) \qquad (\lambda\text{-terms})$$

where $id ::= x \mid y \mid \ldots$ This is a minimal language where the only operations allowed are *abstraction* $\lambda x.M$ and *application* MN. In a language such as ML, one would write $\lambda x.M$ as `function x -> M`.

It is important to notice that the abstraction $\lambda x.M$ *binds* the variable x in the λ-term M just as the quantified first-order formula $\forall x.A$ binds x in A. Consequently, in the λ-calculus a variable can occur free or bound. We denote with $\mathsf{fv}(M)$ the set of variables occurring *free* in the λ-term M.

When writing λ-terms we shall take some freedom. First, we may write $\lambda x_1, \ldots, x_n.M$ for $\lambda x_1 \ldots \lambda x_n.M$. Second, we assume application associates to the left, and therefore write $M_1 M_2 \ldots M_n$ for $(\cdots(M_1 M_2)\cdots M_n)$. Third, we suppose application binds more than λ-abstraction and write $\lambda x.MN$ for $\lambda x.(MN)$.

A number of programming operations can be introduced as *syntactic sugar*. For instance, the operation $\mathsf{let}\ x = M\ \ \mathsf{in}\ N$ that binds the λ-term M to the variable x and runs N can be represented as $(\lambda x.N)M$.

λ-terms, like first-order logic formulae or integrals, are always manipulated up to the *renaming of bound variables*. For instance, we identify the λ-terms $\lambda x.x$ and $\lambda y.y$, just as we would identify the formulae $\forall x\ \ x = x$ and $\forall y\ \ y = y$, or the integrals $\int x\ \mathrm{d}x$ and $\int y\ \mathrm{d}y$.

We remark that renaming involves a substitution of variables for variables. On the other hand, the operation of substitution is really defined up to renaming. For instance, to define $[y/x](\lambda y.xy)$ we start by renaming the abstraction as $\lambda z.[z/y](xy) = \lambda z.xz$, where z is a fresh variable, and then we apply the substitution $[y/x]$ under the abstraction to obtain $\lambda z.yz$. More generally, to define a substitution $[N/x](\lambda y.M)$ we have first to rename the bound variable y as a (fresh) variable z which does not occur free either in N or in $\lambda y.M$ and then we can define the substitution as $\lambda z.[N/x][z/y]M$. As such, the substitution is *not* a function since countably many (equivalent) choices of the fresh variable z are possible. However, we can make it into a function by assuming an enumeration of the variables and picking up, for instance, the first fresh variable that appears in the enumeration.

So we proceed as follows: first we define a substitution function on λ-terms, second we define the relation of α-conversion, and third we assume that λ-terms are handled up to α-conversion. In particular, in the proofs we shall distribute a substitution under a λ-abstraction by silently assuming that an appropriate renaming has been carried on.

Definition 116 (size) *If M is a λ-term then its size $|M|$ is a natural number defined as follows:*

$$|x| = 1, \quad |\lambda x.M| = 1 + |M|, \quad |MN| = 1 + |M| + |N| \ .$$

Definition 117 (substitution) *The substitution of a λ-term N for a variable x in the λ-term M is defined as follows:*

$$[N/x]y = \begin{cases} N & \text{if } x = y \\ y & \text{otherwise} \end{cases}$$

$$[N/x](M_1M_2) = [N/x]M_1[N/x]M_2$$

$$[N/x](\lambda y.M) = \begin{cases} \lambda y.M & \text{if } x \notin \mathsf{fv}(\lambda y.M) \\ \lambda y.[N/x]M & \text{o.w., and } y \notin \mathsf{fv}(N) \\ \lambda z.[N/x][z/y]M & \text{o.w., and } z \text{ first variable s.t. } z \notin \mathsf{fv}(MN). \end{cases}$$

To show that this definition makes sense consider first the definition restricted to the case where N is a variable and check that the substitution of a variable for a variable in a λ-term leaves the size of the λ-term unchanged.

Definition 118 *A* (one-hole) context *C is defined by:*

$$C ::= [\ ] \mid \lambda id.C \mid CM \mid MC \ .$$

We write $C[N]$ for the λ-term obtained by replacing the hole $[\ ]$ *with the λ-term N without paying attention to the potential capture of variables. Formally:*

$$[N] = N, \quad (\lambda x.C)[N] = \lambda x.C[N], \quad (CM)[N] = C[N]M, \quad (MC)[N] = MC[N] \ .$$

We are now ready to define the relation of renaming which is called α-conversion in the λ-calculus. Henceforth λ-terms are considered up to α-conversion.

Definition 119 (α-conversion) *α-conversion is the least equivalence relation $\equiv$ on λ-terms such that for any context C, λ-term M, and variables x, y such that $y \notin \mathsf{fv}(M)$ we have:*

$$C[\lambda x.M] \equiv C[\lambda y.[y/x]M] \ .$$

Remark 120 *As already mentioned, the replacement operation does not pay attention to the bound variables. For instance, if $C = \lambda x.[\ ]$ and $N = x$ then $C[N] = \lambda x.x$. For this reason, contexts, unlike λ-terms, should not be considered up to renaming.*

Definition 121 (β-reduction) *The* β-rule *is the following reduction relation between λ-terms:*

$$(\beta) \quad C[(\lambda x.M)N] \to C[[N/x]M] \ ,$$

where C is a context, M, N are λ-terms, and x is a variable.

The subterm $(\lambda x.M)N$ which is transformed by the β-rule is called the *redex* (or β-redex). We may also refer to the λ-term resulting from the application of the rule as the *reduced* λ-term. Notice that definition 121 is schematic but does *not* quite define a TRS since λ-terms are not quite first-order terms. The equivalence induced by β-reduction is called β-conversion and it is defined as follows.

Definition 122 (β-conversion) *We denote with $=_\beta$ the equivalence relation $\overset{*}{\leftrightarrow}_\beta$.*

Example 123 *Here are some* λ-terms *which are used often enough to deserve a specific name:*

$$I \equiv \lambda x.x, \quad K \equiv \lambda x, y.x, \quad S \equiv \lambda x, y, z.xz(yz), \quad \Delta \equiv \lambda x.xx, \quad \Delta_f \equiv \lambda x.f(xx) \ .$$

And here are some examples of β-reduction *(up to α-conversion!):*

$$II \to I, \quad KMN \to M, \quad SKK \to I, \quad \Delta\Delta \to \Delta\Delta, \quad \Delta_f\Delta_f \to f(\Delta_f\Delta_f) \ .$$

Exercise 124 (β-normal forms) *Let NF be the smallest set of λ-terms such that:*

$$\frac{M_i \in NF \quad i = 1, \ldots, k \quad k \geq 0}{\lambda x_1 \ldots x_n.xM_1 \ldots M_k \in NF} \ .$$

Show that NF is exactly the set of λ-terms in β-normal form.

Exercise 125 (Curry fixed point) *Let $Y \equiv \lambda f.\Delta_f\Delta_f$ where $\Delta_f \equiv \lambda x.f(xx)$. Show that:*

$$YM =_\beta M(YM) \ .$$

This is known as Curry's fixed point combinator.

Exercise 126 (Turing fixed point) Turing's fixed point combinator *is defined by:*

$$Y_T \equiv (\lambda x, y.y(xxy))(\lambda x, y.y(xxy)) \ .$$

Show that $Y_T f$ is not only convertible to, but reduces to*: $f(Y_T f)$.*

7.2 Confluence

Clearly, there are many possible ways of reducing a λ-term. Are they confluent? Let us first examine the case for *local confluence*.

Proposition 127 (local confluence) *Let M be a λ-term. Then the following holds:*

1. *If $M \to M'$ then $[M/x]N \xrightarrow{*} [M'/x]N$.*
2. *If $N \to N'$ then $[M/x]N \to [M/x]N'$.*
3. *β-reduction is* locally confluent*, that is:*

$$\frac{\forall\, M, N, P \;\; (M \to N, \qquad M \to P)}{\exists\, Q \;\; (N \xrightarrow{*} Q, \qquad P \xrightarrow{*} Q)} \; .$$

Proof. (1) By induction on N.

(2) Suppose $N = C[(\lambda y.N_1)N_2]$. We notice:

$$[M/x]((\lambda y.N_1)N_2) \equiv (\lambda y.[M/x]N_1)[M/x]N_2 \to [[M/x]N_2/y]([M/x]N_1) \equiv [M/x]([N_2/y]N_1) \; .$$

(3) The interesting case arises if one redex is contained in the other. Suppose Δ is a β-redex. If $M \equiv C[(\lambda x.M')C'[\Delta]]$ apply (1), and if $M \equiv C[(\lambda x.C'[\Delta])M']$ apply (2). □

Let us notice that β reduction may both *erase* a redex as in $(\lambda x.I)(II) \to I$ and *duplicate* it as in $\Delta(II) \to (II)(II)$. It turns out that it is possible to define a notion of *parallel reduction* $\Rightarrow$ with the following properties.

- $\to \subset \Rightarrow \subset \xrightarrow{*}$.
- A *strong confluence property* holds for $\Rightarrow$: if $M \Rightarrow N$ and $M \Rightarrow N'$ then there is P such that $N \Rightarrow P$ and $N' \Rightarrow P$.
- The relation $\Rightarrow$ is simple enough to be analyzed.

The idea is that in a parallel reduction we are allowed to reduce at once the redexes that are in the λ-term but not those which are created by the reductions. For instance, we have: $(II)(II) \Rightarrow II$ but $(II)(II) \not\Rightarrow I$.

Definition 128 (parallel β-reduction) *Parallel β-reduction is defined as follows:*

$$\frac{}{M \Rightarrow M} \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N'}{(\lambda x.M)N \Rightarrow [N'/x]M'}$$

$$\frac{M \Rightarrow M' \quad N \Rightarrow N'}{MN \Rightarrow M'N'} \qquad \frac{M \Rightarrow M'}{\lambda x.M \Rightarrow \lambda x.M'} \; .$$

Exercise 129 *Let $M \equiv (\lambda x.Ix)(II)$ where $I \equiv \lambda z.z$. What is the minimum number of parallel reductions needed to reduce M to I?*

First we notice the following structural and substitution properties of parallel reduction.

Proposition 130 *Parallel reduction enjoys the following* structural properties*:*

$$\frac{\lambda x.M \Rightarrow N}{N \equiv \lambda x.M', \qquad M \Rightarrow M'}$$

$$\frac{MN \Rightarrow L}{\begin{array}{c}(L \equiv M'N', \qquad M \Rightarrow M', \qquad N \Rightarrow N') \quad or \\ (M \equiv \lambda x.P, \quad P \Rightarrow P', \quad N \Rightarrow N', \quad L \equiv [N'/x]P')\end{array}}$$

Proof. By case analysis on the definition of parallel reduction. □

Proposition 131 *Parallel reduction enjoys the following* substitution property*:*

$$\frac{M \Rightarrow M' \qquad N \Rightarrow N'}{[N/x]M \Rightarrow [N'/x]M'} \ .$$

Proof. By induction on the definition of $M \Rightarrow M'$. For the base case we also need an induction on the structure of M. □

We are then ready to prove *strong confluence* of parallel reduction.

Proposition 132 *Parallel reduction enjoys the following* strong confluence *property:*

$$\frac{\forall\, M, N_1, N_2 \ \ (N_1 \Leftarrow M \Rightarrow N_2)}{\exists\, P \ \ (N_1 \Rightarrow P \Leftarrow N_2)} \ .$$

Proof. One can proceed by induction on $M \Rightarrow N_1$ and case analysis on $M \Rightarrow N_2$ to close the diagram. □

Corollary 133 (confluence, β) *β-reduction is confluent.*

Proof. We have:

$$\rightarrow_\beta \subset \Rightarrow \subset \stackrel{*}{\rightarrow}_\beta \ .$$

If $M \rightarrow_\beta \cdots \rightarrow_\beta N_i$, $i = 1, 2$ then $M \Rightarrow \cdots \Rightarrow N_i$, $i = 1, 2$. Now apply strong confluence to close the diagram and build P such that $N_i \Rightarrow \cdots \Rightarrow P$, $i = 1, 2$. This implies $N_i \stackrel{*}{\rightarrow}_\beta \cdots \stackrel{*}{\rightarrow}_\beta P$, $i = 1, 2$ and, by transitivity of $\stackrel{*}{\rightarrow}_\beta$, we conclude that $N_i \stackrel{*}{\rightarrow}_\beta P$, $i = 1, 2$. □

The β-rule is the basic rule of the λ-calculus. The second most popular rule is the η-rule.

Definition 134 (η-reduction) *The η-rule is defined by:*

$$(\eta) \quad C[\lambda x.Mx] \rightarrow C[M] \qquad \text{if } x \notin \mathsf{fv}(M) \ ,$$

for C context, M λ-term, and x variable.

The η-rule is a kind of *extensionality rule*. If we read it backwards, it asserts that 'every λ-term is a function'. This intuition can actually be made precise in the model theory of λ-calculus.

Proposition 135 (confluence $\beta\eta$) *The following properties hold:*

1. *η reduction is strongly* confluent *in the following sense:*

$$\frac{M \to_\eta N_i \quad i=1,2 \quad N_1 \not\equiv N_2}{\exists P \ (N_i \to_\eta P, \quad i=1,2)} .$$

2. *The β and η reductions* commute *in the following sense:*

$$\frac{M \to_\beta N_1 \quad M \to_\eta N_2 \quad N_1 \not\equiv N_2}{\exists P \ (N_1 (\to_\eta)^* P, \quad N_2 \to_\beta P)} .$$

3. *The $(\to_\beta)^*$ and $(\to_\eta)^*$ reductions commute in the following sense:*

$$\frac{M (\to_\beta)^* N_1 \quad M (\to_\eta)^* N_2}{\exists P \ (N_1 (\to_\eta)^* P, \quad N_2 (\to_\beta)^* P)} .$$

4. *$\beta\eta$ reduction is* confluent.

Proof. (1) Two redexes that superpose have the shape: $\lambda x.C[\lambda y.My]x$. Analyze what can happen.

(2) If the β redex contains the η redex we can have the following situations:

- $(\lambda x.Mx)N$: the reduced are identical.
- $(\lambda x.C[\lambda y.My])N$: close the diagram in one step.
- $(\lambda x.M)C[\lambda y.Ny]$: it may take 0, 1 or more η steps to close the diagram.

On the other hand, if the η redex contains the β redex we can have:

- $\lambda x.(\lambda y.M)x$: the reduced are identical.
- $\lambda x.C[(\lambda y.M_1)M_2]x$: close in one step.

(3) First show commutation of $(\to_\eta)^*$ with respect to $\to_\beta \cup Id$. Then proceed by induction on the number of β reductions.

(4) Consider the number of alternations of $(\to_\beta)^*$ and $(\to_\eta)^*$. □

Example 136 *Here is an extension of the λ-calculus that* does not preserve confluence *(we refer to [Bar84] for a proof). We add to the language a constant D and the rule:*

$$Dxx \to x .$$

This rule may seem artificial, but it is actually a simplification of a natural rule called surjective paring *(an extensionality rule for pairs) which also leads to a non-confluent system:*

$$D(Fx)(Sx) \to x .$$

Here D, F, S are constants where intuitively D is the pairing while F and S are the first and second projection. We stress that here the property that fails is just confluence (not local confluence). Indeed a surjective pairing rule is introduced in terminating *typed λ-calculi. By proposition 47, surjective pairing in these calculi* is *confluent.*

7.3 Programming

All *partial recursive functions* can be represented in the (type free) λ-calculus. Thus the λ-calculus, regarded as a computational model, is *Turing equivalent*. Proving this result is a matter of programming in the λ-calculus. The proof we outline below relies on the following *definition* of the partial recursive functions.

Definition 137 (minimalisation) *Given a* total function $f : \mathbf{N}^{k+1} \to \mathbf{N}$ *a* partial function $\mu(f) : \mathbf{N}^k \rightharpoonup \mathbf{N}$ *is defined by* minimization *as follows:*

$$\mu(f)(x_1, \ldots, x_k) = \begin{cases} x_0 & \text{if } x_0 = min\{x \in \mathbf{N} \mid f(x, x_1, \ldots, x_k) = 0\} \\ \uparrow & \text{if } \forall\, x \;\; f(x, x_1, \ldots, x_k) > 0 \end{cases}$$

where $\uparrow$ *means that the function is undefined.*

Definition 138 *The set of partial recursive functions is the smallest set of functions on (vectors of) natural numbers which contains the* basic functions *(zero, successor, projections) and is closed under* function composition*,* primitive recursion *(see chapter 6.2), and* minimization*.*

We discuss next the representation of partial recursive functions in the λ-calculus.

Definition 139 (Church numerals) *A natural number n is represented by the following λ-term* $\underline{n}$ *known as Church numeral:*

$$\underline{n} \equiv \lambda f.\lambda x.(f \cdots (fx) \cdots) \qquad \textit{(Church numerals)} \tag{7.1}$$

where f is applied n times.

In a sense this is similar to the *tally natural numbers* considered in chapter 6.2. We shall see in chapter 13 that the inductive definition of natural numbers actually *suggests* their representation in the λ-calculus as Church numerals.

We also have to fix a class of λ-terms that represent a diverging computation. A natural choice is to consider the λ-terms that do *not* have a *head normal form.*

Definition 140 (head normal form) *A λ-term is (has) a head normal if it has the shape (it reduces to a λ-term of the shape):*

$$\lambda x_1, \ldots, x_n.xM_1 \cdots M_m \qquad n, m \geq 0.$$

Definition 141 (function representation) *A λ-term F* represents *a partial function* $f : \mathbf{N}^k \to \mathbf{N}$ *if for all* $n_1, \ldots, n_k \in \mathbf{N}$*:*

$$\begin{array}{lll} f(n_1, \ldots, n_k) = m & \textit{iff} & F\underline{n_1} \cdots \underline{n_k} =_\beta \underline{m} \\ f(n_1, \ldots, n_k) \uparrow & \textit{iff} & F\underline{n_1} \cdots \underline{n_k} \textit{ has no head normal form.} \end{array}$$

We can represent the arithmetic functions *addition, successor, and multiplication* with the following λ-terms:

$$\begin{array}{lll} A & \equiv \lambda n.\lambda m.\lambda f.\lambda x.(n\ f)(m\ f\ x) & \text{(addition)} \\ S & \equiv \lambda n.A\ n\ \underline{1} & \text{(successor)} \\ M & \equiv \lambda n.\lambda m.\lambda f.n(m\ f) & \text{(multiplication).} \end{array}$$

To represent *boolean values* we introduce the following λ-terms:

$$T \equiv \lambda x.\lambda y.x \quad \text{(true)} , \qquad F \equiv \lambda x.\lambda y.y \quad \text{(false)}.$$

Then an *if-then-else* λ-term can be defined as follows:

$$C \quad \equiv \lambda x.\lambda y.\lambda z.x\ y\ z \quad \text{(if-then-else)}.$$

The reader may check that: $CTxy \stackrel{*}{\to}_\beta x$ and $CFxy \stackrel{*}{\to}_\beta y$. A *test-for-zero* λ-term on Church numerals can be defined as follows:

$$Z \quad \equiv \lambda n.n(\lambda x.F)T \quad \text{(test-for-zero)}.$$

We can also introduce λ-terms to *build pairs* and to *project pairs* as follows:

$$\begin{array}{lll} P & \equiv \lambda x.\lambda y.\lambda z.z\ x\ y & \text{(pairing)} \\ P_1 & \equiv \lambda p.p(\lambda x, y.x) & \text{(first projection)} \\ P_2 & \equiv \lambda p.p(\lambda x, y.y) & \text{(second projection)}. \end{array}$$

Again, the reader may check that $P_i(PM_1M_n) \stackrel{*}{\to}_\beta M_i$ for $i = 1, 2$.

Exercise 142 *Check that the λ-term:*

$$Pd \ \equiv \ \lambda n, f, x.n(\lambda g, h.h(gf))(\lambda y.x)(\lambda z.z)$$

represents the predecessor function where it is assumed that the predecessor of 0 *is* 0 *(chapter 13 provides a rational reconstruction of this complicated λ-term). Define λ-terms to represent the subtraction function, where* $m - n = 0$ *if* $n > m$*, and the exponential function* n^m*.*

Let us now consider the 3 *composition mechanisms*, namely: function composition, primitive recursion, and minimization. It should be clear that *function composition* can be directly represented in the λ-calculus. Primitive recursion can be regarded as a particular case of recursive function definition. In turn, a *recursive function definition* such as:

$$\mathsf{letrec}\ g(x) = M\ \mathsf{in}\ N\ ,$$

where g may appear in M and N is coded in the λ-calculus as:

$$(\lambda g.N)(Y(\lambda g.\lambda x.M))\ ,$$

where Y is the fixed point combinator of exercise 125 or 126. Moreover, recursive definitions provide a direct mechanism to mimick definitions by minimization. Given a function f, consider the following recursive definition of the function g:

$$g(x_0, x_1, \ldots, x_k) = \mathsf{if}\ (f(x_0, x_1, \ldots, x_k) = 0)\ \mathsf{then}\ x_0\ \mathsf{else}\ g(x_0 + 1, x_1, \ldots, x_k)\ .$$

Then $\mu(f)(x_1, \ldots, x_k) = g(0, x_1, \ldots, x_k)$. Putting all together, we have the following result.

Proposition 143 *For all partial recursive functions* $f : \mathbf{N}^k \to \mathbf{N}$ *there is a closed λ-term* F *which represents* f *in the sense of definition 141.*

7.4 Combinatory logic

Combinatory logic is a relative of the λ-calculus which can be presented as a term rewriting system.

Definition 144 *We consider a binary application operation @ and two constants K and S. As in the λ-calculus, we write MN for $@(M,N)$ and let application associate to the left. The system comes with two term rewriting rules:*

$$K\ x\ y \to x\ , \qquad S\ x\ y\ z \to x\ z(y\ z)\ .$$

It turns out that in combinatory logic there is a way to *simulate* λ-abstraction.

Definition 145 *We define a function λ that takes a variable and a term of combinatory logic and produces a term of combinatory logic. We abbreviate SKK as I.*

$$\begin{array}{lll} \lambda(x,x) & = I & \\ \lambda(x,M) & = KM & \text{if } x \notin \mathsf{var}(M) \\ \lambda(x,MN) & = S(\lambda(x,M))(\lambda(x,N))\ . & \end{array}$$

The fact that we called the function above 'λ' is justified by the following proposition.

Proposition 146 *If M, N are terms of combinatory logic and x is a variable then:*

$$\lambda(x,M)N \xrightarrow{*} [N/x]M\ .$$

Proof. By induction on M following the definition of the translation. □

Exercise 147 *Using the fact that combinatory logic (CL) is a TRS prove local confluence of CL. Then adapt the method of parallel reduction presented in section 7.2 to prove the confluence of CL.*

Combinatory logic seems mathematically simpler than the λ-calculus. Why is it not used? One reason is that terms written in combinatory logic tend to be unreadable. Another deeper reason is that the notion of conversion induced by the rules S and K is weaker than the one induced by the β rule. For instance, the reader may check that the translations in CL of the λ-terms $\lambda z.(\lambda x.x)z$ and $\lambda z.z$ do not have a common reduct. An alternative approach goes through the notion of *closure* (see following chapter 8). This is quite appropriate for discussing implementation techniques, but as in combinatory logic, the notation tends to become less manageable.

7.5 Summary and references

The λ-calculus is a minimal notation to represent higher-order functions. The λ-terms are transformed according to one basic rewriting rule: the β-rule. The λ-calculus with the β-rule is a confluent rewriting system and it is sufficiently expressive to represent all partial recursive functions. A second rule, the η-rule, can be added to the system while preserving confluence. The λ-calculus is *not* a term rewriting system but there are term rewriting systems such as *combinatory logic* which can mimick to some extent the behavior of λ-terms.

The λ-calculus was introduced by Church as part of an investigation in the formal foundations of mathematics and logic [Chu40]. At the time, the λ-calculus provided one of the concurrent formalizations of *partial recursive functions*, *i.e.*, computable functions, along with, *e.g.*, Turing machines. The foundational character of the language is even stronger when it is enriched with *types*. We shall start addressing this point in chapter 10. The related system of combinatory logic is based on work by Schönfinkel and Curry. The book [Bar84] is the basic reference for the type-free λ-calculus. It is enough to skim the first introductory chapters to have an idea of the great variety of results connected to the formalism.

Chapter 8

Weak reduction strategies, closures, and abstract machines

Full $\beta(\eta)$-reduction is the basis for the symbolic manipulation of λ-terms, *e.g.*, in proof assistants, in program transformations, and in higher-order unification and pattern-matching. However, when the λ-calculus is regarded as the core of a programming language it is sensible to consider *weaker* reduction strategies. This chapter focuses on these weaker reduction strategies and their implementation.

8.1 Weak reduction strategies

A *weak* reduction strategy is a strategy to reduce λ-terms that does *not* reduce under functional abstractions. Thus in a weak reduction strategy all λ-terms of the form $\lambda x.M$ are normal forms.

Definition 148 (weak reduction) *We define the weak β-reduction relation $\rightarrow_w$ as the least binary relation on λ-terms such that:*

$$\frac{}{(\lambda x.M)N \rightarrow_w [N/x]M} \qquad \frac{M \rightarrow_w M'}{MN \rightarrow_w M'N} \qquad \frac{N \rightarrow_w N'}{MN \rightarrow_w MN'} \ .$$

As such weak reduction is *not* confluent. For instance, we have:

$$K(II) \rightarrow_w KI \ , \qquad K(II) \rightarrow_w \lambda x.II \ ,$$

and KI and $\lambda x.II$ have no common reduct. The problem here is that the redex II is under a λ and cannot be reduced. When considering the λ-calculus as the core of a programming language, the usual approach is to fix a particular *deterministic* weak reduction strategy. Two popular ones we discuss next are known as *call-by-name* and *call-by-value*. The definition of these strategies relies on a notion of *value*.

Definition 149 (value) *A* value *V is a closed λ-term of the shape $\lambda x.M$ (a λ-abstraction).*

In the following, the call-by-name and call-by-value reduction strategies are defined on *closed* λ-terms. We actually define the reduction strategies in 3 different ways which turn out to be equivalent.

Definition 150 (call-by-name) *We define the call-by-name reduction relation $\to_n$ as the least binary relation on closed λ-terms such that:*

$$\frac{}{(\lambda x.M)N \to_n [N/x]M} \qquad \frac{M \to_n M'}{MN \to_n M'N} \ .$$

Definition 151 (call-by-value) *We define the call-by-value reduction relation $\to_v$ as the least binary relation on closed λ-terms such that:*

$$\frac{}{(\lambda x.M)V \to_v [V/x]M} \qquad \frac{M \to_v M'}{MN \to_v M'N} \qquad \frac{N \to_v N'}{VN \to_v VN'} \ .$$

Remark 152 *The basic difference between call-by-name and call-by-value is that in the latter we insist that the term passed to the function is a value. Also notice that in the definitions above, we have taken the convention that the function is reduced before the argument. Of course, an alternative definition where the argument is reduced before the function is possible. This choice only matters if the language has side-effects (cf. chapter 17).*

The definitions 150 and 151 give a strategy to look for a subterm which is a redex of the right shape. The one-hole context which sourrounds the redex is called *evaluation context*.

Definition 153 (evaluation contexts) *Call-by-name and call-by-value evaluation contexts are denoted with $E, E', \ldots$ and are defined as follows:*

$$\begin{array}{ll} E ::= [\ ] \mid EM & \textit{(call-by-name evaluation context)} \\ E ::= [\ ] \mid EM \mid VE & \textit{(call-by-value evaluation context).} \end{array}$$

Proposition 154 (decomposition) *Let M be a closed λ-term. Then either M is a value or there is a unique call-by-name (call-by-value) evaluation context E such that:*

$$M \equiv E[(\lambda x.M_1)M_2] \qquad (M \equiv E[(\lambda x.M_1)V]) \ .$$

Proof. By induction on the structure of M. M cannot be a variable because it is closed. If M is a λ-abstraction then it is a value. Suppose, $M \equiv M'M''$. We consider the case for *call-by-name*. If M' is a value then it must be a λ-abstraction and $E \equiv [\ ]$. Otherwise, by inductive hypothesis $M' \equiv E'[\Delta]$ where Δ is a β-redex and we take $E \equiv E'M''$. □

We can rely on evaluation contexts to provide alternative and equivalent definitions of call-by-name and call-by-value.

Definition 155 *Let $\to_{en}$ be the least binary reduction relation on closed λ-terms such that:*

$M \to_{en} N$ *if* $M \equiv E[(\lambda x.M_1)M_2]$ *and* $N \equiv E[[M_2/x]M_1], E$ *call-by-name evaluation context.*

Let $\to_{ev}$ be the least binary reduction relation on closed λ-terms such that:

$M \to_{ev} N$ *if* $M \equiv E[(\lambda x.M_1)V]$ *and* $N \equiv E[[V/x]M_1], E$ *call-by-value evaluation context.*

Proposition 156 *The call-by-name reduction relation $\to_n$ coincides with the relation $\to_{en}$ and the call-by-value reduction relation $\to_v$ coincides with the relation $\to_{ev}$.*

PROOF. We consider the proof for call-by-name. To show that $\to_n \subseteq \to_{en}$, we proceed by induction of the proof height of $M \to_n N$. For the base case take $E = [\,]$. For the inductive case, suppose $MN \to_n M'N$ because $M \to_n M'$. Then by inductive hypothesis, there are E' and $\Delta \equiv (\lambda x.M_1)M_2$ such that $M \equiv E'[\Delta]$ and $M' \equiv E'[[M_2/x]M_1]$. Then take $E = E'N$, $MN \equiv E[\Delta]$, $M'N \equiv E[\Delta']$, and $\Delta' \equiv [M_2/x]M_1$.

In the other direction, suppose $E[\Delta] \to_{en} E[\Delta']$ where $\Delta \equiv (\lambda x.M_1)M_2$ and $\Delta' \equiv [M_2/x]M_1$. We proceed by induction on the structure of the evaluation context E. If $E = [\,]$ then $\Delta \to_n \Delta'$. If $E = E'N$ then by induction hypothesis, $E'[\Delta] \to_n E'[\Delta']$ and therefore $E[\Delta] \to_n E[\Delta']$. □

Yet another presentation of call-by-name and call-by-value consists in defining a big-step (cf. section 1.1) evaluation relation $\Downarrow$.

Definition 157 (call-by-name evaluation) *The call-by-name evaluation relation $\Downarrow_n$ is the least binary relation on closed λ-terms such that:*

$$\frac{}{V \Downarrow_n V} \qquad \frac{M \Downarrow_n \lambda x.M' \qquad [N/x]M \Downarrow_n V}{MN \Downarrow_n V}\ .$$

Definition 158 (call-by-value evaluation) *The call-by-value evaluation relation $\Downarrow_v$ is the least binary relation on closed λ-terms such that:*

$$\frac{}{V \Downarrow_v V} \qquad \frac{M \Downarrow_v \lambda x.M' \qquad N \Downarrow_v V' \qquad [V'/x]M \Downarrow V}{MN \Downarrow_v V}\ .$$

Proposition 159 *Let M be a closed λ-term. Then:*

1. *If $M \Downarrow_n V$ then $M(\to_n)^*V$.*
2. *If $M \to_n M'$ and $M' \Downarrow_n V$ then $M \Downarrow_n V$.*
3. *If $M(\to_n)^*M' \not\to_n$ then $M \Downarrow_n M'$.*

The same properties hold if we replace $\Downarrow_n$ with $\Downarrow_v$ and $\to_n$ with $\to_v$, respectively.

PROOF. (1) By induction on the proof height of the judgment $M \Downarrow_n V$. The base case follows by reflexivity of $(\to_n)^*$. For the inductive step, suppose $MN \Downarrow_n V$ because $M \Downarrow_n \lambda x.M_1$ and $[N/x]M_1 \Downarrow_n V$. By inductive hypothesis, $M(\to_n)^*\lambda x.M_1$. Then:

$$MN(\to_n)^*(\lambda x.M_1)N \to_n [N/x]M_1\ ,$$

and by inductive hypothesis $[N/x]M_1(\to_n)^*V$.

(2) If $M \to_n M'$ then $M \equiv (\lambda x.M_1)M_2N_1 \cdots N_k$ and $M' \equiv [M_2/x]M_1N_1 \cdots N_k$. If $M' \Downarrow_n V$ we must have:

$$[M_2/x]M_1 \Downarrow_n \lambda x_1.P_1, \quad [N_1/x_1]P_1 \Downarrow_n \lambda x_2.P_2, \quad \cdots \quad [N_k/x_k]P_k \Downarrow_n V\ .$$

Then to prove $M \Downarrow V$ it suffices to extend the proof for M' with an additional step $\lambda x.M_1 \Downarrow_n \lambda x.M_1$.

(3) By proposition 154, if M' does not reduce then it is a value and we have $M' \Downarrow_n M'$. If M reduces to M' in k steps then we apply property (2) k times starting from M'. □

$$\frac{}{v[\eta] \Downarrow v[\eta]} \qquad \frac{\eta(x)[\eta] \Downarrow v[\eta']}{x[\eta] \Downarrow v[\eta']} \qquad \frac{e[\eta] \Downarrow \mathsf{quote}(e')[\eta'] \quad e'[\eta'] \Downarrow v[\eta'']}{\mathsf{unquote}(e)[\eta] \Downarrow v[\eta'']}$$

By-name:
$$\frac{e[\eta[e'/x]] \Downarrow v[\eta'],}{(\mathsf{let}\ x = e'\ \mathsf{in}\ e)[\eta]) \Downarrow v[\eta']}$$

By-value:
$$\frac{e'[\eta] \Downarrow v'[\eta'] \quad e[\eta[v'/x]] \Downarrow v[\eta]}{(\mathsf{let}\ x = e'\ \mathsf{in}\ e)[\eta] \Downarrow v[\eta]}$$

Table 8.1: Dynamic binding with by-name and by-value evaluation

8.2 Static vs. dynamic binding

The implementation of the reduction of a β-redex such as $(\lambda x.M)N$ is usually decomposed in two steps.

- The formal parameter x is bound to the *argument* N. The collection of bindings is called an *environment*.
- When the formal parameter x is *used* in the body of the function M, the argument bound to it is retrieved from the environment.

This high-level description leaves many design choices unspecified. One basic issue is what exactly constitutes an 'argument'. Indeed, in the programming languages jargon, one speaks of *static* vs. *dynamic* binding. This issue already arises in a very simple language of expressions whose syntax is as follows, where as usual $id ::= x \mid y \mid \cdots$:

$$e ::= \bot \mid n \mid id \mid \mathsf{let}\ id = e\ \mathsf{in}\ e \mid \mathsf{quote}(e) \mid \mathsf{unquote}(e) \qquad \text{(expressions)}.$$

Here $\bot$ represents a computation that *diverges*, n is an *integer*, quote allows to *freeze* the evaluation of an expression and unquote to *unfreeze* it. We denote with *Exp* the set of expressions in this language. The collection of *values* is defined by:

$$v ::= n \mid \mathsf{quote}(e) \qquad \text{(values)}.$$

It is possible to encode this simple language in the λ-calculus and reproduce the same phenomena we describe next.

Definition 160 (dynamic environment) *A* dynamic *environment is partial function* $\eta : Id \rightharpoonup Exp$ *with finite domain mapping identifiers to expressions.*

Table 8.1 introduces two evaluation relations for this language of expressions with dynamic binding following either a *by-name* or a *by-value* strategy. The basic assertion $e[\eta] \Downarrow v[\eta']$ states that an expression e in an environment η evaluates to a value v in an environment η'. The first four rules defining the assertion are shared while two distinct rules, one for by-name and the other for by-value, cover expressions of the shape $\mathsf{let}\ x = e'\ \mathsf{in}\ e$.

In *dynamic binding*, an environment binds an expression with an identifier. However, in turn the expression may contain identifiers and their binding with other expressions may be lost. In *static binding*, we introduce a more complex object which is called a *closure*. This is an expression along with an *environment* that associates identifiers with *closures*. This looks like a circular definition of closure and environment but things can be well-defined in an inductive style as follows.

$$\frac{}{v[\eta] \Downarrow v[\eta]} \qquad \frac{\eta(x) \Downarrow v[\eta']}{x[\eta] \Downarrow v[\eta']} \qquad \frac{e[\eta] \Downarrow \mathsf{quote}(e')[\eta'] \quad e'[\eta'] \Downarrow v[\eta'']}{\mathsf{unquote}(e)[\eta] \Downarrow v[\eta'']}$$

$$\text{By-name:}\ \frac{e[\eta[e'[\eta]/x] \Downarrow v[\eta']}{(\mathsf{let}\ x = e'\ \mathsf{in}\ e)[\eta] \Downarrow v[\eta']} \qquad \text{By-value:}\ \frac{e'[\eta] \Downarrow u[\eta''] \quad e[\eta[u[\eta'']/x]) \Downarrow v[\eta']}{(\mathsf{let}\ x = e'\ \mathsf{in}\ e)[\eta] \Downarrow v[\eta']}$$

Table 8.2: Static binding with by-name and by-value evaluation

Definition 161 (static environment) *The set of environments Env is the smallest set of partial functions on Id such that if* $e_i \in Exp$, $\eta_i \in Env$ *and* $\mathsf{fv}(e_i) \subseteq dom(\eta_i)$ *for* $i = 1, \ldots, n$ *(*$n \geq 0$*) then*

$$[e_1[\eta_1]/x_1, \ldots, e_n[\eta_n]/x_n] \in Env\ .$$

We denote with $\emptyset$ *the empty environment.*

Definition 162 (closure) *A* closure *is a pair* $e[\eta]$ *composed of an expression* $e \in Exp$ *and an environment* $\eta \in Env$ *such that* $\mathsf{fv}(e) \subseteq dom(\eta)$*.*

Table 8.2 defines an evaluation relation whose basic assertion is $e[\eta] \Downarrow v[\eta]$.

We have presented four possible semantics of our language of expressions: dynamic by-name, dynamic by-value, static by-name, and static by-value. We can deem that two of them are different if we can a find a closed expression where one produces a value and the other another value or no value at all.

Proposition 163 *The four presented semantics are different.*

Proof. The expression $e \equiv \mathsf{let}\ x = \bot\ \mathsf{in}\ 3$ distinguishes evaluation *by-name* and *by-value* in both static and dynamic binding. Indeed, the evaluation *by-name* returns a value and the one *by-value* does not. Next consider the following expressions and evaluations:

$$e_1 \equiv \mathsf{let}\ x = 3\ \mathsf{in}\ e_2, \quad e_2 \equiv \mathsf{let}\ y = x\ \mathsf{in}\ e_3, \quad e_3 \equiv \mathsf{let}\ x = 5\ \mathsf{in}\ y\ .$$

$$\begin{array}{llll} e_1[\emptyset] \Downarrow 5[\emptyset] & \text{(dynamic, by-name)} & e_1[\emptyset] \Downarrow 3[\emptyset] & \text{(dynamic, by-value)} \\ e_1[\emptyset] \Downarrow 3[\emptyset] & \text{(static, by-name)} & e_1[\emptyset] \Downarrow 3[\emptyset] & \text{(static, by-value).} \end{array}$$

Thus it remains to distinguish dynamic and static binding with a by-value evaluation. To do this, we rely on the quote, unquote operations and modify the expressions above as follows:

$$e_1 \equiv \mathsf{let}\ x = 3\ \mathsf{in}\ e_2, \quad e_2 \equiv \mathsf{let}\ y = \mathsf{quote}(x)\ \mathsf{in}\ e_3, \quad e_3 \equiv \mathsf{let}\ x = 5\ \mathsf{in}\ \mathsf{unquote}(y)\ .$$

Now we have: $e_1[\emptyset] \Downarrow 5[\emptyset]$ with dynamic binding, by-value and $e_1[\emptyset] \Downarrow 3[\emptyset]$ with static binding, by-value. □

The examples in the previous proof show that the correct implementation of λ-calculus relies on static binding; henceforth dynamic binding will be ignored.

8.3 Environments and closures

We adapt to the call-by-name and call-by-value λ-calculus the notions of environment and closure we have discussed in the previous section 8.2. To this end, we reuse the notion of environment modulo the replacement of the expressions (denoted e) by the λ-terms (denoted M). A *closure*, denoted with $c, c', \ldots$, is now a pair composed of a λ-term and an environment that we shall write as $M[\eta]$. A (closure) *value*, denoted with $v, v', \ldots$, is a closure whose term is a λ-abstraction. Table 8.3 describes the evaluation rules for closures according to a call-by-name and a call-by-value strategy. The first two rules are shared by both strategies. Notice that the β-rule is now decomposed in a rule where the argument is bound as a closure to the formal parameter in the environment and a rule where the closure associated with the formal parameter is retrieved from the environment.

In the presentation of the evaluation relations, at each reduction step, we have to traverse the evaluation context in order to reach the redex to be reduced. A more efficient approach consists in storing the traversed evaluation context in a *stack* and then to push and pop elements on the stack as needed (cf. small-step reduction rules for Imp in chapter 1). The form of the stack depends on the reduction strategy. In call-by-name, the evaluation context is the composition of *elementary contexts* of the shape $[\,]N$, where N is a λ-term. Then a stack representation of the evaluation context is just a list of closures (arguments with their environment):

$$s = c_1 : \ldots : c_n \qquad \text{(stack for call-by-name)}.$$

The reduction relation presented in table 8.4 now operates on pairs $(M[\eta], s)$ composed of a closure and a stack. Initially λ-terms are supposed closed and the stack is supposed empty.

A similar approach works for call-by-value. This time an evaluation context can be regarded as the composition of *elementary contexts* of the shape: $[\,]N$ or $V[\,]$. We code these elementary contexts as a list as follows:

$$[\,]c \;\equiv\; r : c \quad (r \text{ for right}), \quad v[\,] \;\equiv\; l : v \quad (l \text{ for left}).$$

Then the stack s has the shape:

$$s = m_1 : c_1 : \ldots m_n : c_n \qquad \text{where } m \in \{l, r\} \qquad \text{(stack for call-by-value)}.$$

The reduction rules are described in table 8.5.

$$\frac{}{v \Downarrow v} \qquad \frac{\eta(x) \Downarrow v}{x[\eta] \Downarrow v}$$

$$\text{By-name:}\quad \frac{M[\eta] \Downarrow_n \lambda x.M_1[\eta'] \quad M_1[\eta'[M'[\eta]/x]] \Downarrow_n v}{(MM')[\eta] \Downarrow_n v} \qquad \text{By-value:}\quad \frac{M[\eta] \Downarrow_v \lambda x.M_1[\eta'] \quad M'[\eta] \Downarrow_v v' \quad M_1[\eta'[v'/x]] \Downarrow_v v}{(MM')[\eta] \Downarrow_v v}$$

Table 8.3: Evaluation of closures: call-by-name and call-by-value

$$\begin{array}{lcl}(x[\eta], s) & \to & (\eta(x), s)\\ ((MM')[\eta], s) & \to & (M[\eta], M'[\eta] : s)\\ ((\lambda x.M)[\eta], c : s) & \to & (M[\eta[c/x]], s)\end{array}$$

Table 8.4: Abstract machine for call-by-name

$$\begin{array}{lcl}(x[\eta], s) & \to & (\eta(x), s)\\ ((MM')[\eta], s) & \to & (M[\eta], r : M'[\eta] : s)\\ (v, r : c : s) & \to & (c, l : v : s)\\ (v, l : (\lambda x.M)[\eta] : s) & \to & (M[\eta[v/x]], s)\end{array}$$

Table 8.5: Abstract machine for call-by-value

Exercise 164 *Suppose we add to the λ-calculus with call-by-value a certain number of operators $op_1, \ldots, op_m$ with arity $n_1, \ldots, n_m$, $n_j \geq 0$. (1) What are the new evaluation contexts? (2) How is the abstract machine to be modified?*

The rules in the abstract machines described in tables 8.4 and 8.5 form the basis for an implementation. As usual in the implementation of term rewriting rules, one can avoid the costly duplication of terms by using pointers. Specifically, in the rule for application one just needs to duplicate the pointer to the environment rather than the whole environment. In a machine implementation, variables can be replaced by de Brujin indexes which express the number of λ's that one has to traverse in the syntax tree to go from the variable to the binder. For instance, the λ-term $\lambda x.x(\lambda y.xy)$ is represented by $\lambda.0(\lambda.10)$. We can rely on this notation for closures too. In this case, we regard the environment as a list and let a de Brujin index express the number of λ's and elements in the environment that one has to traverse to go from the variable to the term bound to the variable. For instance, in $(\lambda.20)[c; c']$ the variable 2 refers to the closure c' while the variable 0 refers to the λ. Using this notation, *e.g.*, the last rule of Table 8.4 can be written as:

$$((\lambda.M)[\eta], c : s) \ \to \ (M[c : \eta], s) \ .$$

It is interesting to notice that during the (abstract) machine computation (de Brujin) indexes are *never* modified. This simple remark makes manifest that the number of closures in an environment is bounded by the largest index (plus 1) of the initial λ-term to be reduced. In practice, the inputs of a functional program have indexes of bounded size and therefore, in this case, the selection of an element in an environment can be done in constant time. More generally, assuming that lists have bounded length and that duplicated environments are shared each step of computation described by the rules in Tables 8.4 and 8.5 can be implemented in costant time, thus justfying the 'abstract machine' terminology. Notice however, that this analysis ignores the hidden cost of garbage collection.

Exercise 165 *Implement the abstract machines in Tables 8.4 and 8.5 using De Brujin notation.*

Exercise 166 *Suppose we add to the call-by-name λ-calculus two monadic operators: $\mathcal{C}$ for* control *and $\mathcal{A}$ for* abort*. If M is a term then $\mathcal{C}M$ and $\mathcal{A}M$ are λ-terms. An evaluation context E is always defined as: $E ::= [\,] \mid EM$, and the reduction of the control and abort operators is governed by the following rules:*

$$E[\mathcal{C}M] \;\rightarrow\; M(\lambda x.\mathcal{A}E[x]) \;, \qquad E[\mathcal{A}M] \;\rightarrow\; M \;.$$

Adapting the rules in Table 8.4, design an abstract machine to execute the terms in this extended language (a similar exercise can be carried on for call-by-value). Hint: *assume an operator ret which takes a* whole stack *and* retracts *it into a closure; then,* e.g.*, the rule for the control operator can be formulated as: $((\mathcal{C}M)[\eta], s) \;\rightarrow\; (M[\eta], ret(s))$.*

8.4 Summary and references

We have focused on two popular weak reduction strategies: call-by-name and call-by-value and considered alternative and equivalent presentations via *evaluation contexts* and via (big-step) *evaluation relations*. We have highlighted the distinction between static and dynamic binding and shown that the implementation of the former relies on the notions of *closure* and *environment*. Static binding leads to a correct implementation of the call-by-name and call-by-value λ-calculus. Further we have shown that *evaluation contexts* can be implemented as stacks and that this leads to abstract machines for call-by-name and call-by-value. The notion of call-by-name and call-by-value evaluation strategy in the λ-calculus is studied in [Plo75] and an early notion of abstract machine is presented in [Lan64]. de Brujin notation for λ-terms is introduced in [dB72]. The implementation techniques studied for weak reduction strategies can be extended to the (non-weak) β-reduction presented in chapter 7 (see, *e.g.*, [CHL96]).

Chapter 9

Contextual equivalence and simulation

We look for a notion of *pre-order* (and a derived *equivalence*) among program expressions (not necessarily full programs). It should be *natural* and *usable*. To this end, we introduce first a notion of *contextual pre-order* which is *natural* and then we show that it can be characterized as a certain *simulation* which is easier to reason about. The notion of simulation is an example of *co-inductively defined relation*. We take the opportunity to put on solid grounds some basic notions on fixed points of monotonic functions and (co-)inductive definitions.

9.1 Observation pre-order and equivalence

We focus on a (deterministic) *call-by-name* λ*-calculus* as presented in chapter 8. However, the approach applies to programming languages in general (including non-deterministic ones). We work with the evaluation relation for call-by-name in definition 157 and simply write $\Downarrow$ rather than $\Downarrow_n$ since no confusion with call-by-value can arise. Also, in this chapter, all terms are λ-terms.

We write $M \Downarrow$ and say that M converges if $\exists V \;\; M \Downarrow V$. Note that for every closed term M either there is a unique value V to which M evaluates or the evaluation diverges (the derivation tree is infinite). The situation were the evaluation is stuck cannot arise.

In order to define a pre-order (or an equivalence) among two terms we have to *decide* (cf. chapter 1) in which *contexts* the terms can be placed and which *observations* can be performed on the terms once they are placed in the contexts. Our *hypotheses* are as follows.

- All contexts C such that $C[M]$ and $C[N]$ are *closed terms*. We insist on closing contexts because reduction is defined on closed terms.
- We observe the *termination* of the term placed in a closing context. Observing natural numbers or booleans would not change the state of affairs.

Definition 167 (contextual pre-order and equivalence) *We define the contextual* pre-order *on terms as:*

$$M \leq_C N \text{ if for all closing } C \; (C[M] \Downarrow \text{ implies } C[N] \Downarrow) \; .$$

Contextual equivalence *is derived by defining:*

$$M \approx_C N \text{ if } M \leq_C N \text{ and } N \leq_C M \ .$$

Thus two terms are deemed 'equivalent' if from the point of view of the admitted observation they are indistinguishable in any closing context.

Exercise 168 *Prove the following properties:*

1. $\leq_C$ *is a pre-order (reflexive and transitive).*
2. *If* $M \leq_C N$ *then for all contexts* C *(not necessarily closing)* $C[M] \leq_C C[N]$.
3. $\lambda x.\lambda y.x \not\leq_C \lambda x.\lambda y.y$.
4. *If* $\underline{n}, \underline{m}$ *are Church numerals (cf. definition 139) with* $n \neq m$ *then* $\underline{n} \not\leq_C \underline{m}$.
5. *Find a pair of terms* M, N *such that* $M \neq_\beta N$ *and you expect* $M \approx_C N$.

To prove that $M \not\leq_C N$ it suffices to find a context such that $C[M] \Downarrow$ and $C[N] \not\Downarrow$. On the other hand, to prove that $M \leq_C N$ we have to consider *all closing contexts*. For instance, proving $(\lambda x.M)N \leq_C [N/x]M$ is not so easy! This motivates the quest for a more practical proof method based on the notion of simulation which will be discussed in section 9.4.

Exercise 169 *Let* $\leq_{IO}$ *be a relation on closed terms defined by:*

$$M \leq_{IO} N \text{ if } \forall\, P \text{ closed } MP \Downarrow \text{ implies } NP \Downarrow$$

Show that $\leq_{IO}$ *is a pre-order and that it is* not *preserved by contexts.*

9.2 Fixed points

In this and the following section, we make a pause to state and prove some *general facts* on partial orders, monotonic/continuous functions, fixed points, and (co-)inductive definitions. These facts are *used all the time* when manipulating programming languages, formal languages, logics,... The reader would be well-advised to become *acquainted* with these concepts. We start by recalling some standard definitions on partial orders (notice that, unlike in chapter 2, partial orders are supposed to be reflexive).

Definition 170 (partial order) *A* partial order $(L, \leq)$ *is a set* L *equipped with a binary relation* $\leq$ *which is reflexive, anti-symmetric, and transitive.*

Definition 171 (upper/lower bounds) *Suppose* $(L, \leq)$ *is a partial order and let* $X \subseteq L$ *(possibly empty). An element* $y \in L$ *is an* upper bound *for* X *if* $\forall\, x \in X \ x \leq y$. *An element* $y \in L$ *is the* supremum *(sup) of* X *if it is the least upper bound. The notions of* lower bound *and* infimum *(inf) are defined by duality.*

Definition 172 (lattice) *A* lattice *is a partial order* $(L, \leq)$ *such that every pair of elements of* L *has a sup and an inf. A* complete lattice *is a partial order* $(L, \leq)$ *such that every subset of* L *has a sup (the existence of the inf follows).*

Exercise 173 *Show that: (1) The* subsets of a set *with the inclusion relation as partial order form a complete lattice. (2) Every subset of a complete lattice has an* inf. *(3) Every* finite lattice *is* complete.

Next we introduce the notion of monotonic, *i.e.*, order-preserving function and consider the structure of its fixed points in a complete lattice.

Definition 174 (monotonic function) *A* monotonic function f *on a partial order L is a function respecting the order:*

$$\forall\, x, y \;\; (\; x \leq y \text{ implies } f(x) \leq f(y) \;) \; .$$

We say that x is a fixed point *of f if $f(x) = x$.*

Proposition 175 (Tarski) *Let $f : L \to L$ be a monotonic function on a complete lattice. Then f has a* greatest *and a* least fixed point *expressed by:*

$$sup\{x \mid x \leq f(x)\} \qquad and \qquad inf\{x \mid f(x) \leq x\} \; .$$

Proof. Set $z = sup\{x \mid x \leq f(x)\}$. If $f(y) = y$ then $y \leq z$. Hence it remains to show that z is a fixed point. First, we show:

$$z \leq sup\{f(x) \mid x \leq f(x)\} \leq f(z) \; .$$

Then by monotonicity: $f(z) \leq f(f(z))$. And by definition of z, we derive $f(z) \leq z$. □

Exercise 176 *Let $(\mathbf{N} \cup \{\infty\}, \leq)$ be the set of natural numbers with an added maximum element ∞, $0 < 1 < 2 < \ldots < \infty$. Show that every monotonic function f on this order has a fixed point.*

The following exercises consider two situations which often arise in practice.

Exercise 177 (fixed points on finite lattices) *Let $(L, \leq)$ be a* finite *lattice and $f : L \to L$ be a monotonic function. Let $\bot$ ($\top$) be the* least *(*greatest*) element of L. If $x \in L$ then let $f^n(x)$ be the n-*time iteration *of f on x, where $f^0(x) = x$.*

1. *Show that there is an $n \geq 0$ such that the* least fixed point *of f equals $f^n(\bot)$.*
2. *State and prove a dual property for the* greatest fixed point.
3. *Show that these properties fail to hold if one removes the hypothesis that the lattice is* finite.

Exercise 178 (fixed points of continuous functions) *A subset X of a partial order is* directed *if*

$$\forall\, x, y \in X \;\; \exists\, z \in \;\; (x \leq z) \text{ and } (y \leq z) \; .$$

A function on a complete lattice is continuous *if it preserves the sup of* directed sets*:*

$$f(sup(X)) = sup(f(X)) \qquad (if\ X\ directed).$$

1. *Show that a* continuous *function is* monotonic*.*
2. *Give an example of a function on a complete lattice which is continuous but does* not *preserve the sup of a (non-directed) set.*
3. *Show that the* least fixed point *of a continuous function f is expressed by:*

$$sup\{f^n(\bot) \mid n \geq 0\} \ .$$

We can summarize the exercises 177 and 178 as follows. If the the lattice is finite, to compute the least (greatest) fixed point it suffices to iterate a finite number of times the monotonic function starting from the least (greatest) element. Otherwise, if the function is continuous (preserves directed sets), then the *least fixed point* is the *sup* of the (countable) iteration of the function starting from the least element. Similar remarks apply to the greatest fixed point modulo suitable definitions of the notions of co-directed set and co-continuous function.

In general, it is possible to build the least or greatest fixed point of a monotonic function on a complete lattice as an *iterative* process provided one accepts a *transfinite number of iterations*. To do this, we can rely on the notion of *ordinal* in set theory. Intuitively, ordinals are obtained by iterating the operations of successor and supremum; the former are called *successor ordinals* and the latter *limit ordinals*:

$$0, 1, 2, \ldots, \omega, \omega+1, \omega+2, \ldots, \omega+\omega, \ldots \qquad \text{(first few ordinals)}.$$

Formally, in *set theory*, the set X is an ordinal if:

- $Z \in Y \in X$ implies $Z \in X$.
- All sequences such that $X_0 \ni X_1 \ni X_2 \ni \cdots$ are finite.

Notice that if we read $X \in Y$ as $X < Y$ the first property corresponds to *transitivity* and the second to *well-foundation*. So an ordinal is a set which is transitive and well-founded with respect to the $\in$-relation. In set theory, the role of 0 is played by the empty set, the successor of an ordinal set κ is the set $\{\kappa\} \cup \kappa$, and the limit of a sequence of ordinals is their union. Now given a complete lattice L and a monotonic function $f : L \to L$, we can define the (transfinite) sequence:

$$f_0 = \bot \ , \quad f_{\kappa+1} = f(f_\kappa) \ , \quad f_\kappa = \bigvee_{\kappa' < \kappa} f_{\kappa'} \qquad \kappa \text{ limit ordinal.}$$

This defines an increasing sequence which must reach the least fixed point when the cardinality of the ordinal κ is greater than the cardinality of the complete lattice L; for otherwise, we would have a subset of L whose cardinality is greater than L. A dual argument shows that we can approximate the greatest fixed point starting from the top element of the lattice.

9.3 (Co-)Inductive definitions

We discuss examples of inductive and co-inductive set definitions. Behind these definitions there is a complete lattice and a monotonic function, and the (co-)inductive set which is defined is nothing but the least (the greatest) fixed point of the monotonic function.

Example 179 (an inductive definition) *Let* $\mathbf{Z}$ *be the set of integer numbers and suc and* $+$ *the standard successor and addition operations, respectively. We could define:*

> *The least subset of* $\mathbf{Z}$ *which contains* $\{0, 2\}$ *and is closed under the addition operation.*

It is not so obvious that we are indeed defining a set. One has to make sure that the 'least set' does exist. To do this, we explicit a function $f : 2^{\mathbf{Z}} \to 2^{\mathbf{Z}}$,

$$f(X) = \{0, 2\} \cup \{x + y \mid x, y \in X\} \ ,$$

such that "X *contains the set* $\{0, 2\}$ *and* X *is closed under the addition operation" iff* $f(X) \subseteq X$. *Then one remarks that* $2^{\mathbf{Z}}$ *is a complete lattice and* f *is monotonic (indeed continuous, see exercise 178). Hence the least fixed point exists and is expressed by:*

$$\bigcap \{X \mid f(X) \subseteq X\} = \bigcup_{n \geq 0} f^n(\emptyset) \ .$$

Example 180 (another inductive definition) *Let* R *be a* binary relation *on a set* D. *The* reflexive and transitive closure R^* *is the least relation that contains the identity relation, the relation* R *and such that if* $(x, y), (y, z) \in R^*$ *then* $(x, z) \in R^*$. *Let us show that we can regard* R^* *as a least fixed point. As complete lattice, we take the binary relations on the set* D *ordered by inclusion. As monotonic function* f, *we define:*

$$f(S) = Id_D \cup R \cup S \circ S$$

where Id_D *is the identity relation on* D *and* $S \circ S$ *is the (relational) composition of* S *with itself.*

Example 181 (buggy inductive definition) *Monotonicity is a key property. As usual, let us write* $(x \equiv y)$ *mod* 2 *if the integers* x, y *have the property that* $(x - y)$ *is a multiple of* 2. *Suppose we 'define'* X *as the least set of integers such that: (1)* $0 \in X$ *and (2) if* $x \in X$ *then* $\forall\, y \in X \ \ (\ (x \equiv (y + 1))$ *mod* 2 $)$. *Unfortunately such a set does* not *exist. We should have a set of integers* X *such that* $0 \in X$ *and* $(0 \equiv 1)$ *mod* 2.

The notion of *co-inductive definition* is obtained by *dualization*. Rather than looking for the *least* set such that..., we now look for the *greatest* set such that...

Example 182 (co-inductive definition) *A typical example of co-inductive definition arises in the theory of finite automata. Let* $M = (Q, \Sigma, q_0, F, \delta)$ *be a* finite deterministic automaton *with* Q *set of states,* Σ *input alphabet,* q_0 *initial state,* F *set of accepting states, and* $\delta : \Sigma \times Q \to Q$ *transition function. Consider the function* $f : 2^{(Q \times Q)} \to 2^{(Q \times Q)}$ *defined by* $(q, q') \in f(R)$ *if*

1. $q \in F$ *iff* $q' \in F$.
2. $\forall\, a \in \Sigma \ \ (\delta(a, q), \delta(a, q')) \in R$.

It is easy to check that the function f is monotonic on the set of binary relations on Q ordered by inclusion. The least R such that $R = f(R)$ is simply the empty relation which is not very interesting. However, the greatest R such that $f(R) = R$ is the relation that corresponds to state equivalence in finite automata. Indeed, since $2^{Q\times Q}$ is a finite lattice, the definition gives a way to compute the equivalence on states (see exercise 177). Start with the full relation $Q \times Q$ and iterate the function f till you reach a fixed point.

Co-inductive definitions are quite useful in defining various notions of diverging computation. We illustrate this point in the following example.

Example 183 (another example of co-inductive definition) *Let $(S, \to)$ be a set of* states *and $\to \subseteq S \times S$ a* transition relation. *Define D as the* greatest *subset of S such that if $s \in D$ then: $\exists\, s'\ s \to s'$ and $s' \in D$. We take as complete lattice the parts of S ordered by inclusion. The monotonic function f associated with the definition is for $X \subseteq S$:*

$$f(X) = \{s \in X \mid \exists\, s'\ \ s \to s'\}\ .$$

To see the definition at work, suppose $S = \{1, 2, 3, 4\}$ with transitions: $1 \to 2, 3, 4$, $3 \to 1$, $4 \to 4$. The greatest fixed point of f is $\{1, 3, 4\}$ on the other hand the least fixed point is just the empty set. Intuitively, the greatest fixed point of f is the collection of elements starting from which there is an infinite reduction sequence.

To summarize a (co-)inductive definition is well-defined if the associated function is *monotonic*. In this case, the defined set corresponds to a least (greatest) *fixed point* of the associated function.

Exercise 184 *Modify example 183 so as to define the collection of elements which are* not *normalizing,* i.e.*, there is* no *reduction sequence leading to an element in normal form (cf. definition 22).*

9.4 Simulation

Simulation is a standard example of co-inductive definition of a binary relation which can be used to compare programs' behaviors.

Definition 185 (simulation) *We say that a binary relation on* closed terms *S is a* simulation *if whenever $(M, N) \in S$ we have: (1) if $M \Downarrow$ then $N \Downarrow$ and (2) for all P closed $(MP, NP) \in S$. We shall also use the infix notation $M\ S\ N$ for $(M, N) \in S$. We define $\leq_S$ as the largest simulation.*

Exercise 186 *Show that $\leq_S$ is the largest fixed point of the following function on binary relations:*

$$f(S)\ \ = \{(M, N) \mid\ \ M \Downarrow\ \text{implies}\ N \Downarrow\ ,\quad \forall\, P\ \text{closed}\ (MP, NP) \in S\}\ .$$

Definition 187 *We extend $\leq_S$ to* open terms *by defining:*

$$M \leq_S N\ \text{if for all closing substitutions}\ \sigma\ (\sigma M \leq_S \sigma N)\ .$$

We also write $M =_S N$ if $M \leq_S N$ and $N \leq_S M$.

To prove that $M \leq_S N$ (M, N closed) it suffices to find a relation S which is a simulation and such that $M\ S\ N$. The following proof contains several examples of this technique.

Proposition 188 *The following properties of simulation hold:*

1. $\leq_S$ *is a pre-order (on open terms).*
2. *If* $M \leq_S N$ *then for any substitution* σ *(not necessarily closed)* $\sigma M \leq_S \sigma N$.
3. *If* $M \Downarrow V$ *and* $N \Downarrow V$, M, N *closed, then* $M =_S N$.
4. $(\lambda x.M)N =_S [N/x]M$ *(M, N can be open).*
5. *If* $M \leq_C N$ *then* $M \leq_S N$ *(M, N can be open).*

Proof. (1) On closed terms $\leq_S$ is reflexive and transitive. If M is an open term then $M \leq_S M$ because for every closing substitutions $\sigma M \leq_S \sigma M$. Suppose $M \leq_S N$ and $N \leq_S P$. Given a closing substitution σ for M, P we can always extend it to a closing substitution σ' for N. Then we have:

$$\sigma M \equiv \sigma' M \leq_S \sigma' N \leq_S \sigma' P \equiv \sigma P\ .$$

(2) Suppose $M \leq_S N$ and σ is a substitution. To prove $\sigma M \leq_S \sigma N$, we have to check that for all closing substitution σ', $\sigma'(\sigma M) \leq_S \sigma'(\sigma N)$. And this holds because $(\sigma' \circ \sigma)$ is a closing substitution for M, N.

(3) We check that the following relation on closed terms is a simulation:

$$S = \{(MP_1 \cdots P_n, NP_1 \cdots P_n) \mid M \Downarrow V, N \Downarrow V, n \geq 0\}\ .$$

If $MP_1 \cdots P_n \Downarrow$ then we must have $VP_1 \cdots P_n \Downarrow$ and therefore $NP_1 \cdots P_n \Downarrow$.
Also if $(MP_1 \cdots P_n, NP_1 \cdots P_n) \in S$ then for all P closed, $(MP_1 \cdots P_n P, NP_1 \cdots P_n P) \in S$.
(4) Both the following relation and its inverse are simulations:

$$S = \{((\lambda x.M)NP_1 \cdots P_n, [N/x]MP_1 \cdots P_n) \mid n \geq 0\}\ .$$

Thus $(\lambda x.M)N =_S [N/x]M$ holds if $(\lambda x.M)N$ is closed. If $(\lambda x.M)N$ is open and σ is a closing substitution then we observe:

$$\sigma(\lambda x.M)N \equiv (\lambda x.\sigma M)\sigma N =_S [\sigma N/x]\sigma M \equiv \sigma([N/x]M)\ .$$

(5) First check that $\leq_C$ on closed terms is a simulation. Thus $\leq_C \subseteq \leq_S$ on closed terms. For open terms, suppose $M \leq_C N$, let x^* be the list of variables free in M, N, and let σ be any closing substitution. Then take the closed terms: $M' \equiv (\lambda x^*.M)\sigma(x^*)$ and $N' \equiv (\lambda x^*.N)\sigma(x^*)$. We have $M' \leq_C N'$ and therefore $M' \leq_S N'$. Moreover $\sigma M =_S M'$ and $\sigma N =_S N'$. Therefore: $\sigma M \leq_S \sigma N$. □

Exercise 189 *Prove that:*

1. *If* $M \not\Downarrow$ *and* $N \not\Downarrow$ *then* $M =_S N$.
2. *Let* $\Omega_n \equiv \lambda x_1 \ldots . \lambda x_n . \Omega$. *Then* $\Omega_n <_S \Omega_{n+1}$ *(strictly) and, for all* M, $\Omega_0 \leq_S M$.

3. Let $K^\infty \equiv YK$. Then for all M, $M \leq_S K^\infty$.

4. $\lambda x.x \not\leq_S \lambda x, y.xy$ (thus η-conversion is unsound).

So it seems *easier proving* $M \leq_S N$ than proving $M \leq_C N$. However, we still need to prove that $\leq_S$ is *preserved by contexts*. If this property holds, then it is easy to conclude that the largest simulation coincides with the contextual pre-order.

Proposition 190 *Let x be a variable and M, N, P be terms. If $M \leq_S N$ then: (1) $MP \leq_S NP$ and (2) $\lambda x.M \leq_S \lambda x.N$.*

(1) Because $\leq_S$ is a simulation.

(2) Suppose $M \leq_S N$ and let σ be a closing substitution for $\lambda x.M, \lambda x.N$. As usual, suppose σ commutes with λ up to renaming. Now for all closed P, define σ' as the substitution that extends σ so that $\sigma'(x) = P$. Then, by hypothesis:

$$[P/x]\sigma M = \sigma' M \leq_S \sigma' N = [P/x]\sigma N \ .$$

Then we have:

$$(\lambda x.\sigma M)P =_S [P/x]\sigma M \leq_S [P/x]\sigma N = (\lambda x.\sigma N)P \ .$$

Clearly $(\lambda x.\sigma M) \Downarrow$ implies $(\lambda x.\sigma N) \Downarrow$. With reference to the function f defined in exercise 186, we have shown $(\lambda x.\sigma M)\ f(\leq_S)\ (\lambda x.\sigma N)$, and we know $f(\leq_S) = \leq_S$. □

Exercise 191 *Let us revise the pre-order considered in exercise 169 by defining a relation $\leq_{IO^*}$ on closed terms as:*

$$M \leq_{IO^*} N \text{ if for all } n \geq 0,\ P_1, \ldots, P_n \text{ closed, } MP_1 \cdots P_n \Downarrow \text{ implies } NP_1 \cdots P_n \Downarrow.$$

Prove that $\leq_{IO^*}$ coincides with $\leq_S$.

Unfortunately, it is *not* so easy to prove that $M \leq_S N$ implies $PM \leq_S PN$. The proof plan is to introduce an *auxiliary relation* $\leq_A$ on *open* terms, which includes $\leq_S$, is preserved by contexts, and, with some work, turns out to coincide with $\leq_S$.

Definition 192 (auxiliary simulation relation) *The auxiliary relation $M \leq_A N$ is defined* inductively *on M by the following rules:*

$$\frac{x \leq_S N}{x \leq_A N} \qquad \frac{M \leq_A M' \quad \lambda x.M' \leq_S N}{\lambda x.M \leq_A N} \qquad \frac{M_1 \leq_A M_1' \quad M_2 \leq_A M_2' \quad M_1'M_2' \leq_S N}{M_1M_2 \leq_A N} \ .$$

The definition of the auxiliary relation seems rather *mysterious*. To have a clue, let us look at its *properties*.

Proposition 193 *The auxiliary relation $\leq_A$ enjoys the following properties:*

1. $\leq_A$ is reflexive.

2. $\leq_A \circ \leq_S \subseteq \leq_A$.

3. $\leq_S \subseteq \leq_A$.

PROOF. (1) By induction on the structure of the term.

(2) Suppose $M \leq_A N \leq_S P$ and proceed by induction on the proof of $M \leq_A N$.

(3) By the previous property (2), using the fact that $\leq_A$ is reflexive. □

The next proposition introduces the key properties of the auxiliary relation.

Proposition 194 (key properties) *Let M, M', N, N' be terms. Then:*

1. *If $M \leq_A M'$ and $N \leq_A N'$ then $[N/x]M \leq_A [N'/x]M'$.*
2. *If $M \leq_A M'$ and $N \leq_A N'$ then $MN \leq_A M'N'$.*
3. *If $M \Downarrow V$ and $M \leq_A N$ then $V \leq_A N$.*

PROOF. (1) By induction on the proof of $M \leq_A M'$. For instance, suppose:

$$\frac{M \leq_A M'' \quad \lambda y.M'' \leq_S M'}{\lambda y.M \leq_A M'} .$$

We have to prove: $\lambda y.[N/x]M \leq_A [N'/x]M'$. By inductive hypothesis, we know: $[N/x]M \leq_A [N'/x]M''$. Also, by substitutivity of $\leq_S$ we have:

$$[N'/x](\lambda y.M'') \equiv \lambda y.[N'/x]M'' \leq_S [N'/x]M' .$$

Hence, by definition of $\leq_A$, we conclude.

(2) Consider the terms Mx and $M'x$, with x fresh. From $M \leq_A M'$ we can derive $Mx \leq_A M'x$. Then by property (1) above, we know that:

$$N \leq_A N' \text{ implies } [N/x](Mx) \equiv MN \leq_A [N'/x](M'x) \equiv M'N' .$$

(3) We proceed by induction on $M \Downarrow V$. We detail the main case. Suppose:

$$\frac{M_1 \Downarrow \lambda x.M_1'' \quad [M_2/x]M_1'' \Downarrow V}{M_1M_2 \Downarrow V} \quad \frac{M_1 \leq_A M_1' \quad M_2 \leq_A M_2' \quad M_1'M_2' \leq_S N}{M_1M_2 \leq_A N} .$$

By induction hypothesis on $M_1 \Downarrow \lambda x.M_1''$ we derive:

$$\lambda x.M_1'' \leq_A M_1' . \tag{9.1}$$

The proof of the property (9.1) above must have the following shape:

$$\frac{M_1'' \leq_A M_1''' \quad \lambda x.M_1''' \leq_S M_1'}{\lambda x.M_1'' \leq_A M_1'} . \tag{9.2}$$

By the substitutivity property (proposition 194.1), we derive:

$$[M_2/x]M_1'' \leq_A [M_2'/x]M_1''' . \tag{9.3}$$

Also by proposition 190, we know that:

$$[M_2'/x]M_1''' =_S (\lambda x.M_1''')M_2' \leq_S M_1'M_2' (\leq_S N) . \tag{9.4}$$

So we have: $[M_2/x]M_1'' \Downarrow V$, $[M_2'/x]M_1'' \leq_A N$, and by inductive hypothesis, we conclude: $V \leq_A N$. □

We can now prove the announced result: the largest simulation coincides with the contextual pre-order.

Proposition 195 *Let M, N be terms. Then:*

1. *$\leq_A$ is a simulation (and therefore $\leq_A \subseteq \leq_S$).*
2. *$M \leq_A N$ implies $M \leq_C N$.*

Proof. (1) If $M \leq_A N$ and $M \Downarrow V$ then $V \leq_A N$ by proposition 194.3. Since V has the shape $\lambda x.M'$ we must have:

$$\frac{M' \leq_A M'' \quad \lambda x.M'' \leq_S N}{\lambda x.M' \leq_A N},$$

and, by definition of simulation, $N \Downarrow$. Also by proposition 194.2, we know that $M \leq_A N$ implies $MP \leq_A NP$.

(2) Suppose $M \leq_A N$ and C one hole, closing context. Then $C[M] \leq_A C[N]$, and this implies $C[M] \leq_S C[N]$. So if $C[M] \Downarrow$ then $C[N] \Downarrow$ too. □

Exercise 196 *We define a notion of contextual pre-order $\leq_C$ for the call-by-value λ-calculus simply by taking definition 167 and considering that the predicate $\Downarrow$ corresponds to call-by-value evaluation. We also say that a (call-by-value) simulation is a binary relation S on closed λ terms such that whenever $(M, N) \in S$ we have: (1) if $M \Downarrow$ then $N \Downarrow$ and (2) for all closed values V, $(MV, NV) \in S$. Denote with $\leq_S$ the largest simulation. If M, N are terms (possibly open) say that $M \leq_S N$ if for all closing substitutions σ mapping variables to values $\sigma M \leq_S \sigma N$. Adapt the theory developed in this chapter to prove that the pre-orders $\leq_C$ and $\leq_S$ coincide.*

9.5 Summary and references

The contextual pre-order is a natural compositional way to compare terms and the simulation pre-order is an effective method to reason on this relation. Simulation is a typical example of co-inductive definition and corresponds to the greatest fixed point of a monotonic function. Dually, inductive definitions correspond to the least fixed point of a monotonic function. Such fixed points are guaranteed to exist for monotonic functions over complete lattices and in many practical situations they can be effectively computed or at least approximated. The notion of simulation (and bisimulation, see chapter 24) was introduced in [Par81] in the context of the semantics of concurrent processes where it is extensively used. The proof that simulation is preserved by contexts is based on [How96] and it can be extended to a number of other calculi, including, *e.g.*, the call-by-value λ-calculus (exercise 196).

Chapter 10

Propositional types

The reader is supposed to be familiar with the usage of types in programming languages. Then, if we regard the λ-calculus as the kernel of a programming language, it is natural to wonder what kind of types could be associated with λ-terms. For the time being, we shall focus on a collection of *propositional* types which include basic types such as integers, booleans,... and functional, product, and sum types. We may also refer to these types as *simple* types as opposed to more complex types including quantifications we shall discuss in chapters 12 and 13.

Definition 197 *We define the collection of (functional)* propositional types *as follows:*

$$A ::= b \mid tid \mid (A \to A) \ ,$$

where b is a basic type *(there can be more) and $tid ::= t \mid s \mid \ldots$ are* type variables*.*

Definition 198 *A* type context Γ *is a set of pairs $\{x_1 : A_1, \ldots, x_n : A_n\}$ where all variables $x_1, \ldots, x_n$ are* distinct*.*

We use $\Gamma, x : A$ as an *abbreviation* for $\Gamma \cup \{x : A\}$ where x does not occur in Γ. Also we abbreviate type context to context whenever no confusion may arise with *term* contexts. Table 10.1 presents a first system to assign types to λ-terms. In this formulation, the variable of a λ-abstraction is decorated with a type as in $\lambda x : A.M$. As in the usual programming practice, the type A specifies the type of the parameter of the function. The presented system is composed of a rule (*asmp*) to discharge an *assumption* from the context, a rule $(\to_I)$ which *introduces* a functional type, and a rule $(\to_E)$ which *eliminates* a functional type. This presentation style where the rules associated with the type operators are split into introduction and elimination rules comes from logic where it is called *natural deduction*. The following exercises are a first illustration of the connection between type systems and logic.

Exercise 199 *Show that if $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ is derivable then $(A_1 \to \cdots (A_n \to B) \cdots)$ is a* tautology *of propositional logic where we interpret $\to$ as implication and atomic types as propositional variables. Conclude that there are types A which are* not inhabited*,* i.e.*, there is no (closed) λ-term M such that $\emptyset \vdash M : A$.*

Exercise 200 *Show that there is no λ-term M such that: $\emptyset \vdash M : (b \to b) \to b$. Write $A \to b$ as $\neg A$. Show that there are λ-terms N_1 and N_2 such that:*

$$\emptyset \vdash N_1 : A \to (\neg\neg A) \ , \quad \emptyset \vdash N_2 : (\neg\neg\neg A) \to (\neg A) \ .$$

On the other hand, there are tautologies which are not inhabited! For instance, consider: $A \equiv ((t \to s) \to t) \to t$. *Show that there is no* λ*-term* M *in normal form such that* $\emptyset \vdash M : A$ *is derivable. This is enough because later we shall show that all typable* λ*-terms normalize to a* λ*-term of the same type. For another example, show that there is no* λ*-term* M *in normal form such that* $\emptyset \vdash M : \neg\neg t \to t$ *is derivable (the intuitionistic/constructive negation is not involutive!).*

Next we review a few alternative presentations of the type system. In Table 10.1, λ-abstractions are decorated with types. However, we can also consider a presentation where types are assigned to *pure*, *i.e.*, type-less, λ-terms. Then one speaks of a presentation in *Curry-style*, as opposed to the previous one which is in *Church-style*. In our case, the only difference between the two is that the rule $(\to_I)$ in Curry-style becomes:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ .$$

An important consequence of this change is that now a λ-term may have more than one type. This makes the *type inference problem* (see chapters 11 and 12) more interesting and so this problem is often studied for systems in Curry-style.

Yet another presentation of the type system is possible by *labeling every variable* with its type and by *dropping the context*. This gives the system presented in Table 10.2. Finally, we may decide to *label every* λ*-term* (not just the variables) with its type and in this case we can *drop the type* since the type of a λ-term is just the outermost label. The resulting system is given in Table 10.3.

The previous exercises 199 and 200 suggest that the functional type constructor can be regarded as a logical implication. It turns out that one may push this connection further by regarding the product (sum) type constructor as a logical conjunction (disjunction). The resulting type system in natural deduction style is presented in Table 10.4. Later in chapters 12 and 13, we shall see that this connection can be extended even further to types with universal and existential quantifications.

Not all term constructors found in a programming language have a logical interpretation. For instance, Table 10.5 introduces typing rules for a constant zero Z, a successor function S, and a fixed point combinator Y. While the fixed point combinator allows to define general recursive functions (see chapter 7) its logical interpretation is problematic. Indeed, with the rule (Y), every type A is *inhabited* by the closed λ-term $Y(\lambda x : A.x)$. Thus the typing rule for Y is definitely incompatible with logic as it leads to inconsistency!

$$(asmp)\quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$(\to_I)\quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad (\to_E)\quad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Table 10.1: Assignment of propositional types to λ-terms

$$\begin{array}{lll} id & ::= x \mid y \mid \ldots \\ M & ::= id^A \mid \lambda id^A.M \mid MM \end{array}$$

$$\frac{}{x^A : A} \qquad \frac{M : A \to B \quad N : A}{MN : B} \qquad \frac{M : B}{\lambda x^A.M : A \to B}$$

Table 10.2: System with type labelled variables

$$\begin{array}{lll} id & ::= x \mid y \mid \ldots \\ M & ::= id^A \mid (\lambda id^A.M)^A \mid (MM)^A \end{array}$$

$$\frac{}{x^A} \qquad \frac{M^{A\to B} \quad N^A}{(M^{A\to B}\ N^A)^B} \qquad \frac{M^B}{(\lambda x^A.M^B)^{(A\to B)}}$$

Table 10.3: System with type-labelled λ-terms

$$(\times_I) \quad \frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1, M_2 \rangle : A_1 \times A_2}$$

$$(\times_{E,1}) \quad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_1(M) : A_1} \qquad (\times_{E,2}) \quad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_2(M) : A_2}$$

$$(+_{I,1}) \quad \frac{\Gamma \vdash M : A_1}{\Gamma \vdash \mathsf{in}_1{}^{A_1+A_2}(M) : A_1 + A_2} \qquad (+_{I,2}) \quad \frac{\Gamma \vdash M : A_2}{\Gamma \vdash \mathsf{in}_2{}^{A_1+A_2}(M) : A_1 + A_2}$$

$$(+_E) \quad \frac{\Gamma \vdash M : (A_1 + A_2) \quad \Gamma \vdash N_i : A_i \to B \quad i = 1, 2}{\Gamma \vdash \mathsf{case}(M, N_1, N_2) : B}$$

Table 10.4: Typing rules for product and sum

$$(\mathsf{Z}) \quad \frac{}{\Gamma \vdash \mathsf{Z} : nat} \qquad (\mathsf{S}) \quad \frac{\Gamma \vdash M : nat}{\Gamma \vdash \mathsf{S}M : nat} \qquad (\mathsf{Y}) \quad \frac{\Gamma \vdash M : (A \to A)}{\Gamma \vdash \mathsf{Y}M : A}$$

Table 10.5: Non-logical extension of the type system

10.1 Subject reduction

If a λ-term is *well-typed*, then by inspection of the rules we see, *e.g.*, that the λ-term cannot contain the application of a natural number to a function. However, to get *static guarantees* we must make sure that *typing is invariant under reduction*, *i.e.*, if a λ-term is well-typed and we reduce it then we still get a well-typed λ-term. To establish this property we note the following property.

Proposition 201 (substitution) *If* $\Gamma, x : A \vdash M : B$ *and* $\Gamma \vdash N : A$ *then* $\Gamma \vdash [N/x]M : B$.

Proof. By induction on the *height of the proof* of $\Gamma, x : A \vdash M : B$. For instance, suppose the *root of the proof* has the shape:

$$\frac{\Gamma, x : A, y : B' \vdash M : B''}{\Gamma, x : A \vdash \lambda y.M : (B' \to B'')} \ ,$$

with $x \neq y$. Then by inductive hypothesis, $\Gamma, y : B' \vdash [N/x]M : B''$ and conclude by $(\to_I)$. □

We can now state the invariance of typing under reduction as follows. Historically, this property is called *subject reduction*.

Proposition 202 (subject reduction) *If* $\Gamma \vdash M : A$ *and* $M \to_\beta N$ *then* $\Gamma \vdash N : A$.

Proof. Recall, that $M \to_\beta N$ means:

$$M \equiv C[(\lambda x.M_1)M_2] \quad N \equiv C[[M_2/x]M_1] \ .$$

To prove subject reduction we proceed by induction on the structure of C. The *basic case* follows directly from the substitution lemma. For the *inductive case* consider in turn the cases where: (1) $C = \lambda y.C'$, (2) $C = C'P$, and (3) $C = PC'$. □

In the 'pure' λ-calculus, we identify the normal forms with the results of the computation. In applications, however, one can distinguish two kinds of normal/irreducible forms: those that correspond to a *value* and those that correspond to an *erroneous configuration* like dividing by zero, or applying an integer to another integer. Thus a program, *i.e.*, a closed λ-term, has *three possible outcomes*: (1) it *returns a value*, (2) it reaches an *erroneous configuration*, and (3) it *diverges* (cf. chapter 1). Besides being invariant by reduction, a desirable property for a type system is that: *well-typed programs cannot go wrong*, or at least that they go wrong in some expected way (*e.g.*, division by zero). This property is often called *progress*, because in its simple form it requires that if a program is not a value then it can reduce (progress). The following exercise elaborates on this point.

Exercise 203 (on progress) *Suppose we reconsider the* non-logical extension *of the simply typed λ-calculus with a* basic type *nat,* constants Z, S, Y, *and with the following fixed-point rule:*

$$C[\mathsf{Y}M] \to C[M(\mathsf{Y}M)] \ .$$

Let a program *be a closed typable λ-term of type nat and let a* value *be a λ-term of the shape* $(\mathsf{S}\cdots(\mathsf{SZ})\cdots)$. *Show that if P is a program in* normal form *(cannot reduce) then P is a* value.

10.2 A normalizing strategy for the simply typed λ-calculus

We describe a *normalizing* strategy for the simply typed λ-calculus. To this end we introduce some measures of the complexity of a type and a λ-term.

Definition 204 (type degree) *The* degree of a type *is defined as follows:*

$$\delta(t) = 1 \ , \qquad \delta(A \to B) = 1 + max(\delta(A), \delta(B)) \ .$$

Definition 205 (redex degree) *Let $R \equiv (\lambda x : A.M)N$ be a redex. The* degree of the redex*, written $\delta_r(R)$ is the degree of the type associated with the λ-term $(\lambda x : A.M)$.*

Definition 206 (term degree) *The degree of a λ-term, written $\delta_t(M)$, is 0 if M is in normal form and the maximum of the degrees of the redexes contained in M otherwise.*

Remark 207 *A redex R is also a λ-term and we have $\delta_r(R) \leq \delta_t(R)$.*

Proposition 208 (degree and substitution) *If x is of type A then*

$$\delta_t([N/x]M) \leq max(\delta(A), \delta_t(M), \delta_t(N)) \ .$$

Proof. The redexes in $[N/x]M$ fall in the following categories. (1) The redexes already in M. (2) The redexes already in N. (3) New redexes arising by the substitution if $N \equiv \lambda y.N'$ and $M = C[xM']$. These redexes have degree $\delta(A)$. □

Proposition 209 (degrees and reduction) *If $M \to N$ then $\delta_t(N) \leq \delta_t(M)$.*

Proof. We apply the previous analysis. □

We now define a reduction strategy that reduces first an innermost redex of maximal degree.

Definition 210 (innermost maximal degree strategy) *Let M be a λ-term which is* not in normal form*. The innermost maximal degree strategy selects a redex R of maximal degree ($\delta_r(R) = \delta_t(M)$) and such that all redexes contained in R have* lower degree*.*

Proposition 211 *The innermost maximal degree strategy is normalizing.*

Proof. Notice that by reducing an innermost maximal degree redex we guarantee that the reduced λ-term contains *strictly less redexes of maximal degree*. Then we prove normalization by taking as *measure*:

$$\mu(M) = (n, m) \ ,$$

with the *lexicographic order* (from left to right), where $n = \delta_t(M)$ and m is the number of redexes of maximal degree. If $m = 0$ then M is in normal form. If $m = 1$ then the reduced term has lower degree (first component decreases). If $m > 1$ then the first component does not increase (proposition 209) and the second decreases. □

10.3 Termination of the simply typed λ-calculus

A λ-term M is called *strongly normalizable* if all β-reductions starting from M terminate (thus strong-normalization is just a synonymous for termination!).

Definition 212 *Let SN be the set of* strongly normalizable λ-terms.

This set plays a role similar to the set WF for RPO termination (section 4.3). The notion of size of a λ-term follows definition 116.

Definition 213 (maximal length) *If $M \in SN$ then the maximal length of a derivation starting from M is called the* reduction depth *of M, and is denoted $depth(M)$.*

Remark 214 *The maximal length is well-defined because the reduction tree of a λ-term is finitely branching (cf. proposition 42).*

In order to show that all simply typed λ-terms are $\rightarrow_\beta$-strongly normalizable, the *key idea* is to interpret types as subsets of the set SN of strongly normalizing λ-terms.

Definition 215 (type interpretation) *The interpretation of a propositional type A is defined as follows:*

$$\begin{array}{lcl} [\![b]\!] = [\![t]\!] & = & SN \\ [\![A \to B]\!] & = & \{M \mid \forall\, N \in [\![A]\!]\ \ (MN \in [\![B]\!])\}\ . \end{array}$$

Proposition 216 *For any type A, the following properties hold:*

1. $[\![A]\!] \subseteq SN$.
2. *If $N_i \in SN$ for $i = 1, \ldots, k$ then $xN_1 \cdots N_k \in [\![A]\!]$.*
3. *If $[N/x]MM_1 \cdots M_k \in [\![A]\!]$ and $N \in SN$ then $(\lambda x.M)NM_1 \cdots M_k \in [\![A]\!]$.*

Proof. By induction on A.

Atomic types. (1) By definition. (2) The reductions of $xN_1 \ldots N_k$ are just an interleaving of the reductions of $N_1, \ldots N_k$. (3) We have:

$$depth((\lambda x.M)NM_1 \ldots M_k) \leq depth(N) + depth([N/x]MM_1 \ldots M_k) + 1\ .$$

Functional types $A \to B$. Suppose $M \in [\![A \to B]\!]$.

(1) By inductive hypothesis, $x \in [\![A]\!]$. Hence $Mx \in [\![B]\!] \subseteq SN$, by inductive hypothesis. This entails $M \in SN$.

(2) Take $M = xN_1 \ldots N_k$ with $N_i \in SN$. Take $N_{k+1} \in [\![A]\!] \subseteq SN$. By inductive hypothesis, $xN_1 \ldots N_kN_{k+1} \in [\![B]\!]$.

(3) If $[N/x]MN_1 \ldots N_k \in [\![A \to B]\!]$ then by the interpretation of the functional types we have:

$$\forall\, N_{k+1} \in [\![A]\!]\ \ [N/x]MN_1 \ldots N_kN_{k+1} \in [\![B]\!]\ .$$

Then by inductive hypothesis on B:

$$\forall\, N_{k+1} \in [\![A]\!]\ \ (\lambda x.M)NN_1 \ldots N_kN_{k+1} \in [\![B]\!]\ ,$$

which is equivalent to $(\lambda x.M)NN_1 \ldots N_k \in [\![A \to B]\!]$. □

Remark 217 *These interpretations of types are called* reducibility candidates*. These are sets of of strongly normalizable λ-terms (property 1) which contain at least the variables (and more) (property 2), and are closed under head expansions (property 3).*

We can now state the soundness of the interpretation.

Proposition 218 (soundness) *If* $x_1 : A_1, \ldots, x_k : A_k \vdash M : B$ *(in Curry-style) and* $N_i \in [\![A_i]\!]$ *for* $i = 1, \ldots, k$ *then* $[N_1/x_1, \ldots, N_k/x_k]M \in [\![B]\!]$.

Proof. By *induction on the typing proof.*

(*asmp*) Immediate by *definition.*

($\to_E$) By the *interpretation* of $\to$.

($\to_I$) Here is what goes on in a simplified case. By inductive hypothesis on $x : A \vdash M : B$ we have:

$$\forall\, N \in [\![A]\!]\ \ [N/x]M \in [\![B]\!].$$

Then, by the *closure under head expansions* of the interpretations we derive:

$$\forall\, N \in [\![A]\!]\ \ (\lambda x.M)N \in [\![B]\!]\ ,$$

which is equivalent to $(\lambda x : A.M) \in [\![A \to B]\!]$. □

The strong normalization property follows as a simple corollary.

Corollary 219 (strong normalization) *If a λ-term is typable then it is strongly normalizing.*

Proof. Suppose $x_1 : A_1, \ldots, x_k : A_k \vdash M : B$. We know $x_i \in [\![A_i]\!]$. By proposition 218 (soundness), $M \in [\![B]\!]$ and we know $[\![B]\!] \subseteq SN$. □

A rational reconstruction of the proof could go as follows. (1) We decide to interpret types as sets of strongly normalizing λ-terms and show that $\vdash M : A$ implies $M \in [\![A]\!]$. (2) Then the definition 215 of the type interpretation is natural and the properties 1 and 2 of proposition 216 amount to check that indeed a type interpretation is composed of strongly normalizing λ-terms and it is not empty. (3) Finally, the need for property 3 of proposition 216 (closure under head expansion) appears in the proof of proposition 218 (soundness, case ($\to_I$)).

Exercise 220 (recursive types) *Assume a* recursively defined type t *satisfying the equation* $t = t \to b$ *and suppose we add a rule for typing up to type equality:*

$$\frac{\Gamma \vdash M : A \quad A = B}{\Gamma \vdash M : B}\ .$$

*Show that in this case the following λ-term (Curry's fixed point combinator) is typable (*e.g.*, in Curry-style):*

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))\ .$$

Are the λ-terms typable in this system terminating?

10.4 Summary and references

A minimal property required for a type system is that it is invariant under reduction. Sometimes, it is possible to connect type systems to logic. This is the so called Curry-Howard correspondence which goes as follows:

λ-calculus	proof system
type	proposition
λ-term	proof
reduction	proof normalization

In a natural deduction presentation, an opportunity for a proof normalization arises when the introduction of an operator is followed by an elimination. For instance, λ-abstraction is followed by an application, a pairing is followed by a projection, and an injection is followed by a case selection. The book [GLT89] is a good introduction to the connections between proof theory and type theory including alternative presentations of the logical systems.

Chapter 11

Type inference for propositional types

Given a (pure) λ-term M and a context Γ, the *type inference problem* is the problem of checking whether there is a type A such that $\Gamma \vdash M : A$. Given a (pure) λ-term M, a *variant of the problem* is to look for a type A *and* a context Γ such that $\Gamma \vdash M : A$. Connected to the type inference problem is the problem of actually producing an *informative output*. Typically, if a λ-term M is typable, we are interested in a *synthetic representation of its types*, and if it is not, we look for an *informative error message*.

11.1 Reduction of type-inference to unification

We present a polynomial time reduction of the *type inference problem* for the propositional type system in Curry style (chapter 10) to the *syntactic unification problem* (chapter 3). The existence of a *most general unifier* for the unification problem leads to the existence of a *most general type* for the type inference problem.

Definition 221 *A* goal *is a finite set G of triples (Γ, M, A) where Γ is a* context*, M a* λ-term*, and A a* propositional type.

We assume that all bound variables in M are *distinct* and different from the free ones, that all free variables occur in the context Γ, and that for every variable x we have a type variable t_x. We define a *reduction relation* on pairs (G, E). Assuming $G = \{g\} \cup G'$ and $g \equiv (\Gamma, M, A) \notin G'$, all the rules produce a pair $(G' \cup G_g, E \cup E_g)$ where G_g and E_g are defined in Table 11.1.

Proposition 222 *The reduction specified in Table 11.1 terminates.*

g	G_g	E_g	
(Γ, x, A)	$\emptyset$	$\{t_x = A\}$	
(Γ, M_1M_2, A)	$\{(\Gamma, M_1, t_1 \to A), (\Gamma, M_2, t_1)\}$	$\emptyset$	(t_1 fresh)
$(\Gamma, \lambda x.M_1, A)$	$\{(\Gamma, x : t_x, M_1, t)\}$	$\{A = t_x \to t\}$	(t fresh)

Table 11.1: Reduction of type inference to unification

PROOF. It is enough to notice that every reduction step replaces a triple (Γ, M, A) by a finite number of triples (Γ', M', A') where M' is structurally smaller than M. □

We introduce some notation. In the following, we consider substitutions S that act on the first-order terms built over the signature $\Sigma = \{b^0, \to^2\}$. We define:

$$\begin{array}{ll} S \models E & \text{if } S \text{ unifies } E\ , \\ S \models (\Gamma, M, A) & \text{if } S\Gamma \vdash M : S(A) \text{ is derivable,} \\ S \models G & \text{if } \forall\, g \in G\ \ S \models g\ , \\ S \models (G, E) & \text{if } S \models G \text{ and } S \models E\ . \end{array}$$

Given a λ-term M_0 with free variables $x_1, \ldots, x_n$, we set the initial pair to $(G_0, \emptyset)$, with $G_0 = \{(\Gamma_0, M_0, t_0)\}$, t_0 fresh, and $\Gamma_0 = x_1 : t_{x_1}, \ldots, x_n : t_{x_n}$. Next, we state the main properties of the reduction.

Proposition 223 *If* $(G_0, \emptyset) \stackrel{*}{\to} (G, E)$ *then:*

1. *If* $S \models (G, E)$ *then* $S\Gamma_0 \vdash M_0 : St_0$.
2. *If* $\Gamma \vdash M_0 : A$ *then* $\exists S(S \models (G, E), S\Gamma_0 \subseteq \Gamma,$ *and* $A = St_0)$.

PROOF. For both properties we proceed by induction on the length of the reduction.

(1) For instance, suppose (1) true for: $(G \cup \{(\Gamma, MN, A)\}, E)$. The rule for application produces the pair (G', E) with $G' = G \cup \{(\Gamma, M, t_1 \to A), (\Gamma, N, t_1)\}$. Suppose $S \models (G', E)$. This means $S \models (G, E)$, $S\Gamma \vdash M : S(t_1 \to A)$, and $S\Gamma \vdash N : St_1$. By $(\to_E)$, we conclude $S\Gamma \vdash MN : SA$. Thus $S \models (G \cup \{(\Gamma, MN, A)\}, E)$, and by hypothesis $S\Gamma_0 \vdash M_0 : St_0$.

(2) For instance, suppose: $\Gamma \vdash M_0 : A$, $S \models (G \cup \{(\Gamma', \lambda x.M, A')\}, E)$, $S\Gamma_0 \subseteq \Gamma$, and $A = St_0$. This implies: $S\Gamma' \vdash \lambda x.M : S(A')$, which entails: $S\Gamma', x : A_1 \vdash M : A_2$, $SA' = A_1 \to A_2$, for some A_1, A_2. Suppose we reduce to the pair:

$$(G \cup \{(\Gamma', x : t_x, M, t)\}, E \cup \{A' = t_x \to t\})\ .$$

Then take $S' = S[A_1/t_x, A_2/t]$. □

Remark 224 *Property (1) entails the* soundness *of the method. Indeed, suppose from the initial goal we derive a set of equations E and a substitution S such that $S \models E$ (a* unifier*). Then we derive a* correct typing *$S\Gamma_0 \vdash M_0 : St_0$. On the other hand, property (2) entails the* completeness *of the method. Suppose $\Gamma \vdash M_0 : A$ is a* valid typing*. Then we can reduce (Γ_0, M_0, t_0) to $(\emptyset, E)$ and find a* unifier *S for E such that $S\Gamma_0$ is contained in Γ and $St_0 = A$. In particular, if we take the most general unifier S of E and we apply it to t_0 we obtain the* most general type*: every other type is an instance of St_0.*

Example 225 *The most general type of the λ-term $\lambda f.\lambda x.f(f(x))$ is $(t \to t) \to (t \to t)$. Note that strictly speaking the most general type is* not unique*. For instance, $(s \to s) \to (s \to s)$ is also a most general type of the λ-term considered.*

Remark 226 (graphical presentation) *It is possible to give an equivalent 'graphical' presentation of the unification method. (1) Rename bound variables so that they are all distinct and different from the free ones. (2) Draw the tree associated with the λ-term. (3) Associate*

a distinct type variable with every internal node of the tree. (4) Associate a type variable t_x with a leaf node corresponding to the variable x. (5) For every abstraction node $(\lambda x.M^{t'})^t$ generate the equation $t = t_x \to t'$. (6) For every application node $(M^{t'}N^{t''})^t$ generate the equation $t' = t'' \to t$.

Exercise 227 *Compute, if they exist,* the most general types *of the following λ-terms:*

$$\lambda x.\lambda y.\lambda z.xz(yz), \quad \lambda x.\lambda y.x(yx), \quad \lambda k.(k(\lambda x.\lambda h.hx)) \ .$$

11.2 Reduction of unification to type inference

We discuss a method to reduce any unification problem to a type-inference problem. We also show that the principal types are exactly the types inhabited by a closed λ-term. We suppose as usual that $K \equiv \lambda x.\lambda y.x$.

Proposition 228 *The principal type of the λ-term E below is $t \to (t \to (s \to s))$.*

$$E \equiv \lambda x.\lambda y.\lambda w.Kw(\lambda f.\lambda p.p(fx)(fy)) \ .$$

Proof. The fact that f is applied to both x and y forces the equality of the types of x and y. On the other hand, since the principal type of K is $t \to s \to t$, the type of w must be equal to the type of the result. □

Proposition 229 *Let M_1 and M_2 be λ-terms with principal types A_1 and A_2, respectively. Then the principal type of the λ-term $\lambda f.E(fM_1)M_2$, for $f \notin \mathsf{fv}(M_1M_2)$, is $(A_1 \to A_2) \to (s \to s)$.*

Proof. The function f must apply to the λ-term M_1 and based on proposition 228, the type of the result of f must be equal to the type of M_2. □

Proposition 230 *For every type A with (type) variables contained in $\{t_1, \ldots, t_n\}$ there is a λ-term M_A whose principal type is: $t_1 \to \cdots \to t_n \to A \to (s \to s)$, where $s \notin \{t_1, \ldots, t_n\}$.*

Proof. By induction on the structure of A. If $A = t_i$ we take:

$$\lambda x_1 \ldots x_n.\lambda y.Ex_iy : t_1 \to \cdots \to t_n \to t_i \to s \to s \ .$$

If $A = A_1 \to A_2$, by inductive hypothesis we have:

$$M_{A_i} : t_1 \to \cdots \to t_n \to A_i \to (s \to s) \quad i = 1, 2 \ .$$

We define:

$$M_{A_1 \to A_2} \equiv \lambda x_1 \ldots x_n.\lambda y.\lambda z.KzP \ .$$

This λ-term has the expected type provided we can force in the λ-term P the type of y to be $(A_1 \to A_2)$. We observe that if we write:

$$Q_i \equiv M_{A_i}x_1 \ldots x_n y_i \ .$$

we force the type of y_i to be A_i. Then we can define P as follows:

$$P \equiv \lambda y_1, y_2.\lambda p.pQ_1Q_2(E(yy_1)y_2) \ ,$$

and as expected the type of y is $A_1 \to A_2$. □

Exercise 231 *(1) Apply the method to the types t_1, t_2 and $(t_1 \to t_2)$ relatively to the set of (type) variables $\{t_1, t_2\}$. (2) Write a program that builds the equivalent of the λ-term M_A in a language of the ML-family and uses the type-inference system to compute its principal type.*

Proposition 232 *Given two types A and B there is a λ-term $U_{A,B}$ which is typable if and only if A and B are unifiable.*

Proof. We have:

$$M_A : t_1 \to \cdots \to t_n \to A \to s \to s \ , \quad M_B : t_1 \to \cdots \to t_n \to B \to s \to s \ .$$

We build:

$$\begin{array}{ll} U_{A,B} & \equiv \lambda x_1 \ldots x_n . \lambda y_1 . \lambda y_2 . \lambda p . p P_A P_B (E y_1 y_2) \ , \\ P_A & \equiv M_A x_1 \ldots x_n y_1 \ , \\ P_B & \equiv M_B x_1 \ldots x_n y_2 \ . \end{array}$$

□

Exercise 233 *Apply proposition 232 if $A = t_1$ and $B = t_2$ and if $A = t_1$ and $B = t_1 \to t_2$.*

Proposition 234 *Every unification problem can be reduced to a type-inference problem.*

Proof. We know from exercise 61 that every unification problem reduces to a unification problem composed of one equation with terms built over a signature with exactly one binary symbol. We take '$\to$' as binary symbol and using the proposition 232 above we build two types A and B which are unifiable iff the λ-term $U_{A,B}$ is typable. □

Proposition 235 *For every type A, there is a λ-term $F_{(A \to A)}$ whose principal type is $(A \to A)$.*

Proof. Let A be a type whose type variables are contained in $\{t_1, \ldots, t_n\}$. Let M_A be a λ-term with principal type: $t_1 \to \cdots \to t_n \to A \to s \to s$ (proposition 230). Then build the λ-term:

$$F_{A \to A} \equiv \lambda y . K y (\lambda x_1 \ldots x_n . (M_A x_1 \ldots x_n y)) \ .$$

This λ-term has principal type $(A \to A)$. □

Proposition 236 *Let M be a λ-term with type A (not necessarily its principal type). Then one can build a closed λ-term N whose principal type is A. Thus the inhabited types are exactly the principal types.*

Proof. By proposition 235, the principal type of $F_{A \to A}$ is $(A \to A)$. Then the λ-term $F_{A \to A} M$ has principal type A. □

11.3 Summary and references

A *type inference problem* can be (efficiently) reduced to a *syntactic unification problem*. Then the existence of a *most general unifier* is reflected back in the existence of a *most general type*. We have also shown that every unification problem reduces to a type inference problem and that for every inhabited type A it is possible to build a λ-term whose principal type is A. Notice however that knowing if a type is inhabited is a PSPACE-complete problem [Sta79]. The connection between type inference and unification was already pointed out in [Hin69]. By now, the reduction of a program analysis problem to the solution of a set of constraints has become a standard technique. For instance, the data flow analyses performed by optimizing compilers are reduced to systems of monotonic boolean equations.

Chapter 12

Predicative polymorphic types and type inference

Consider any standard sorting algorithm sort on lists. Most likely, the sorting algorithm just depends on a boolean predicate on the elements of the list while the type of the elements of the list does not really matter. One says that the sorting algorithm is *polymorphic* in that it can be applied to data of different, but related, shape. We could type sort as follows:

$$\mathsf{sort} : \forall\, t\ \ \mathsf{list}(t) \to (t \to t \to \mathsf{bool}) \to \mathsf{list}(t)\ ,$$

with the following intuitive meaning: for any type t, given a list of elements of type t, and a binary predicate on t, the function sort returns a list of elements of type t. Notice that we are assigning to sort a type which is not quite *simple*, *i.e.*, propositional, as it contains a *universal quantification* over types. In this chapter, we introduce a particular class of universally quantified types which can be used to type polymorphic functions. We then study the type inference problem for the type system extended with such types.

12.1 Predicative universal types and polymorphism

The reader is supposed to be familiar with propositional and first-order logic. In the standard interpretation of propositional logic predicates are boolean values while in first-order logic they are regarded as relations over some universe. In second order logic, we can quantify over predicates. For instance, here are some formulae in propositional, first-order, and second-order logic:

$(P \supset P) \supset (P \supset P)$	(Propositional formula)
$\forall\, x\ \ (P(x) \supset P(\mathsf{S}(x))) \supset (P(\mathsf{Z}) \supset \forall\, x\ \ P(x))$	(First-order formula)
$\forall\, P\ \ (P \supset P) \supset (P \supset P)$	(Second-order formula).

Following the types-as-formulae correspondence outlined in chapter 10, we may consider a type system where quantification over type variables is allowed. One fundamental question is whether a type with quantified types should be regarded as an ordinary type, or if it should be lifted to a superior status. In this chapter, we take the second option. In the logical jargon, this corresponds to a *predicative* approach to second-order quantification. We shall not dwell

$$(asmp)\quad \frac{x:\sigma\in\Gamma}{\Gamma\vdash x:\sigma}$$

$$(\to_I)\quad \frac{\Gamma,x:A\vdash M:B}{\Gamma\vdash \lambda x.M:A\to B}\qquad (\to_E)\quad \frac{\Gamma\vdash M:A\to B\qquad \Gamma\vdash N:A}{\Gamma\vdash MN:B}$$

$$(\forall_I)\quad \frac{\Gamma\vdash M:\sigma\quad t\notin \mathsf{ftv}(\Gamma)}{\Gamma\vdash M:\forall t.\sigma}\qquad (\forall_E)\quad \frac{\Gamma\vdash M:\forall t.\sigma}{\Gamma\vdash M:[A/t]\sigma}$$

Table 12.1: Predicative type system (Curry-style)

into foundational issues and just assume a distinction between types without quantification and types with quantification which will be called henceforth *type schema*. Thus:

$$\begin{array}{lll} A & \equiv (t\to t)\to(t\to t) & \text{is a } \textit{type},\\ \sigma & \equiv \forall\, t\ \ (t\to t)\to(t\to t) & \text{is a } \textit{type schema}. \end{array}$$

The advantage of this approach is that we stay close to propositional types and that in this way we can generalize the type inference techniques presented in chapter 11. The inconvenience is that we do not have the full power of second-order quantification. This power will be explored in chapter 13. The syntax of types, type schemas, and type contexts is specified as follows.

$$\begin{array}{ll} A ::= b \mid \mathit{tid} \mid (A\to A) & \text{(types)}\\ \sigma ::= A \mid \forall \mathit{tid}.\sigma & \text{(type schemas)}\\ \Gamma ::= \mathit{id}:\sigma,\ldots,\mathit{id}:\sigma & \text{(type contexts).} \end{array}$$

We stress that $\forall t.(t\to t)$ is *not* a type and $\forall t.t\to\forall t.t$ is *not* a type schema. Table 12.1 presents an extended type system with type schemas. Notice that types schemas can occur in type contexts and that in the rules $(\to_I)$ and $(\to_E)$, we handle *types* (not type schemas)

Next, we explore the connection between *universal (predicative) types* and *polymorphism*. Sometimes, the same code/function can be applied to different data-types. For instance, the functional that iterates twice a function $D\equiv\lambda f.\lambda x.f(fx)$, will work equally well on a function over *booleans* or over *integers*. In the context of *propositional types*, we have already seen that we can automatically infer for D the *most general type*:

$$D:(t\to t)\to(t\to t)\ .$$

The reader may be under the impression that this type is good enough to represent the fact that D will work on any argument of type $(A\to A)$. Almost but *not quite*... Suppose:

$$F_1:\ \ (\mathsf{bool}\to\mathsf{bool})\ ,\qquad F_2:\ \ (\mathsf{int}\to\mathsf{int})\ ,$$

and consider the λ-term $P\equiv \mathsf{let}\ f=D\ \mathsf{in}\ \langle fF_1,fF_2\rangle$, where as usual $\mathsf{let}\ x=M\ \mathsf{in}\ N\equiv(\lambda x.N)M$ and $\langle M,N\rangle\equiv\lambda z.zMN$. The reader may check that the λ-term P has no propositional type and that the example can be rephrased in the pure λ-calculus without appealing to the basic types bool and int. A possible *way out* is to consider that D has a *type schema*:

$$\sigma\equiv\forall t.(t\to t)\to(t\to t)\ ,$$

and then to *specialize* it just before it is applied to F_1 and F_2. This is *almost* what we can do with the predicative type system in Table 12.1. The *problem* that remains is that we cannot

$$(\forall_I)\quad \frac{\Gamma \vdash M : \sigma \quad t \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \lambda t.M : \forall t.\sigma} \qquad (\forall_E)\quad \frac{\Gamma \vdash M : \forall t.\sigma}{\Gamma \vdash MA : [A/t]\sigma}$$

Table 12.2: Rules $(\forall_I)$ and $(\forall_E)$ in Church-style

really type the λ-term $(\lambda f.\langle fF_1, fF_2\rangle)D$ as expected because $\sigma \to \cdots$ is *not* even a type schema according to our definitions. One could allow *more complex types...*, but there is a more *conservative solution* which consists in taking the let-definition as a *primitive* and giving the following typing rule for it:

$$(\mathsf{let})\quad \frac{\Gamma, x : \sigma \vdash N : A \quad \Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : A}\ .$$

This is a *first formalization* (others will follow) of a type system which captures the polymorphism available in the *ML* programming languages.

Example 237 *Consider* $M \equiv \lambda y.\mathsf{let}\ x = \lambda z.z\ \mathsf{in}\ y(xx)$ *which is not typable in the propositional type system but has type* $A \equiv ((t \to t) \to t') \to t'$ *in the ML type system. The main difference is that we assign to the variable* x *the type schema* $\sigma \equiv \forall s.(s \to s)$*. Then taking* $B \equiv (t \to t) \to t'$ *we can derive:*

$$\dfrac{\dfrac{y : B, z : s \vdash z : s}{y : B \vdash \lambda z.z : (s \to s) \quad s \notin \mathsf{ftv}(B)}}{y : B \vdash \lambda z.z : \sigma}$$

On the other hand, one can derive: $y : B, x : \sigma \vdash (xx) : (t \to t)$*.*

The rules $(\forall_I)$ and $(\forall_E)$ in Table 12.1 are *not syntax-directed*. When do we apply them? One possibility is to *ask the programmer* to specify when this must be done. This requires an *enriched syntax* for λ-terms which includes: (i) the possibility to abstract a λ-term M with respect to a type variable t, a *type abstraction* $\lambda t.M$, and (ii) the possibility to apply a λ-term M to a type A, a *type application* MA. The resulting system in *Church-style* is composed of the rules $(asmp)$, $(\to_I)$, $(\to_E)$, and (let) we have already presented modulo the fact that: (1) the λ-abstraction is decorated with a type (cf. Table 10.1), and (2) the rules in Table 12.2 replace the homonymous rules in Table 12.1.

Due to the simplicity of the *ML* system, it is actually *possible to foresee* the points where the rules $(\forall_I)$ and $(\forall_E)$ need to be applied. This leads to a *Curry-style* and *syntax directed* type system: the shape of the λ-term determines the rule to apply.

Definition 238 (generalisation) *Given a pair composed of a context* Γ *and a type* A*, its* generalization $G(\Gamma, A)$ *is defined as the type schema that results by quantifying the type variables which occur in* A *but do not occur free in* Γ*.*

Example 239 *If* $\Gamma = x : \forall t.(s \to t)$ *and* $A = s \to (t \to r)$ *then* $G(\Gamma, A) = \forall t.\forall r.A$*.*

The idea to define the syntax-directed type system presented in Table 12.3 is to *generalize as much as possible* let-variables and then *instantiate once* the type schema in the context.

$$(asmp)\quad \frac{x : \forall t^*.A \in \Gamma}{\Gamma \vdash^{syn} x : [B^*/t^*]A} \qquad (\mathsf{let})\quad \frac{\Gamma, x : G(\Gamma, B) \vdash^{syn} N : A \quad \Gamma \vdash^{syn} M : B}{\Gamma \vdash^{syn} \mathsf{let}\ x = M\ \mathsf{in}\ N : A}$$

$$(\to_I)\quad \frac{\Gamma, x : A \vdash^{syn} M : B}{\Gamma \vdash^{syn} \lambda x.M : A \to B} \qquad (\to_E)\quad \frac{\Gamma \vdash^{syn} M : A \to B \qquad \Gamma \vdash^{syn} N : A}{\Gamma \vdash^{syn} MN : B}$$

Table 12.3: Predicative type system (Curry-style, syntax directed)

We shall use the entailment symbol $\vdash^{syn}$ when referring to judgments in this system. Unlike in the type system in Table 12.1, the syntax-directed type system in Table 12.3 can only assign *types* to λ-terms (*not* type schema).

Exercise 240 (running example, continued) *Consider again the λ-term:*

$$M \equiv \lambda y.\mathsf{let}\ x = \lambda z.z\ \mathsf{in}\ y(xx)\ ,$$

and check that we can derive: $\emptyset \vdash^{syn} M : ((t \to t) \to t') \to t'$.

12.2 A type inference algorithm

Building on the syntax directed presentation of the type system, we describe next a type inference algorithm. We rely on the following *notation*:

M, N	type free λ-terms with let-definitions,
Γ	type context with propositional types,
Θ	partial function from identifiers to pairs (Γ, A).

The (partial) function $PT(M, \Theta)$ tries to infer a principal typing judgment $\Gamma \vdash M : A$ for M. The search is *driven by M* while Θ keeps track of the *type schema assigned to let-bound variables*. We assume: (i) all bound variables are renamed so as to be distinct and different from the free variables, and (ii) in all subterms $\mathsf{let}\ x = N\ \mathsf{in}\ M$ we have $x \in \mathsf{fv}(M)$. Given two typing judgments $J_i \equiv \Gamma_i \vdash M_i : A_i$, $i = 1, 2$, we denote by $UnifyApl(J_1, J_2)$ a triple (S, t, J_2') obtained as follows:

1. obtain $J_2' \equiv \Gamma_2' \vdash M_2 : A_2'$ by *renaming* the type variables of J_2 so that they are *disjoint* from those in J_1,
2. select a *fresh* type variable t,
3. build the *system of equations*:
$$E = \{A_1 = A_2' \to t\} \cup \{A = A' \mid x : A \in \Gamma_1, x : A' \in \Gamma_2'\},$$
4. compute (if it exists) a *most general unifier* S of E.

With this notation, the type inference algorithm is presented in Table 12.4.

Exercise 241 (running example, continued) *Consider again the λ-term: $M \equiv \lambda y.\mathsf{let}\ x = \lambda z.z\ \mathsf{in}\ y(xx)$ and check that $PT(M, \emptyset) = \emptyset \vdash M : ((t \to t) \to t') \to t'$.*

$PT(M,\Theta) = \mathsf{case}\ M$
$x:$ $\mathsf{case}\ \Theta(x)$ $\quad(\Gamma, A) \quad : \Gamma \vdash x : A$ $\quad\_ \quad : x : t_x \vdash x : t_x$
$\lambda x.M:$ $\mathsf{let}\ (\Gamma \vdash M : A) = PT(M,\Theta)\ \mathsf{in}$ $\quad\mathsf{case} \quad x : A' \in \Gamma$ $\quad\mathsf{true} \quad : \Gamma \backslash (x : A') \vdash \lambda x.M : A' \to A$ $\quad\_ \quad : \Gamma \vdash \lambda x.M : t \to A, t\ \text{fresh}$
$M_1 M_2:$ $\mathsf{let}\ J_i \equiv (\Gamma_i \vdash M_i : A_i) = PT(M_i,\Theta)\ i = 1,2\ \mathsf{in}$ $\mathsf{let}\ (S, t, \Gamma_2' \vdash M_2 : A_2') = \mathit{UnifyApl}(J_1, J_2)\ \mathsf{in}$ $S(\Gamma_1 \cup \Gamma_2' \vdash M_1 M_2 : t)$
$\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2:$ $\mathsf{let}\ (\Gamma_1 \vdash M_1 : A_1) = PT(M_1,\Theta)\ \mathsf{in}$ $\mathsf{let}\ \Theta' = \Theta[(\Gamma_1, A_1)/x]\ \mathsf{in}$ $\mathsf{let}\ (\Gamma_2 \vdash M_2 : A_2) = PT(M_2,\Theta')\ \mathsf{in}$ $\Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : A_2$

Table 12.4: Type-inference algorithm

$$(asmp)\quad \frac{x : A \in \Gamma}{\Gamma \vdash^{let} x : A} \qquad (\mathsf{let})\quad \frac{\Gamma \vdash^{let} [M/x]N : A \quad \Gamma \vdash^{let} M : B}{\Gamma \vdash^{let} \mathsf{let}\ x = M\ \mathsf{in}\ N : A}$$

$$(\rightarrow_I)\quad \frac{\Gamma, x : A \vdash^{let} M : B}{\Gamma \vdash^{let} \lambda x.M : A \rightarrow B} \qquad (\rightarrow_E)\quad \frac{\Gamma \vdash^{let} M : A \rightarrow B \qquad \Gamma \vdash^{let} N : A}{\Gamma \vdash^{let} MN : B}$$

Table 12.5: Propositional typing with let-expansion

12.3 Reduction of stratified polymorphic typing to propositional typing

We may consider a type system where to type $\mathsf{let}\ x = M\ \mathsf{in}\ N$ we actually type M and $[N/x]M$, where by typing we mean *propositional typing*. This way of proceeding is *not particularly efficient* because the let-expansion might take exponential time (see following exercise 243). However, the interesting point is that the λ-terms typable in this way are exactly those typable in the original *ML* system. Therefore we have the following *intuitive characterization*:

ML typing = Propositional typing + let-expansion.

The type inference algorithm presented in Table 12.4 is a way to *keep implicit the let-expansion* (but type renaming still forces a type expansion as we shall see shortly!). Table 12.5 describes a type system based on let-expansion. We use the entailment symbol $\vdash^{let}$ to distinguish this system from the previous ones. In the (let) rule, we just check that the substituted term M is typable with some type B; this check is necessary if $x \notin \mathsf{fv}(M)$.

Exercise 242 (running example) *Consider again the λ-term:*

$$M \equiv \lambda y.\mathsf{let}\ x = \lambda z.z\ \mathsf{in}\ y(xx)\ ,$$

and check that we can derive: $\emptyset \vdash^{let} M : ((t \rightarrow t) \rightarrow t') \rightarrow t'$.

Exercise 243 (let-expansion) *Consider a λ-calculus extended with* let *definitions of the shape $\mathsf{let}\ x = M\ \mathsf{in}\ N$. Let C denote a context with a hole (cf. definition 118) and define the reduction relation $\rightarrow_{\mathsf{let}}$ as follows:*

$$\rightarrow_{\mathsf{let}} = \{(\ C[\mathsf{let}\ x = N\ \mathsf{in}\ M]\ ,\ C[[N/x]M]\ )\mid C\ context, M, N\ \lambda\text{-}terms\ , x\ variable\}\ .$$

We extend the definition of size of a λ-term $|M|$ (cf. definition 116) with: $|\mathsf{let}\ x = M\ \mathsf{in}\ N| = 1 + |M| + |N|$. We also define the depth *$d(M)$ of a λ-term as follows:*

$$\begin{array}{ll} d(x) = 1, & d(MN) = max(d(M), d(N)), \\ d(\lambda x.M) = d(M), & d(\mathsf{let}\ x = M\ \mathsf{in}\ N) = d(M) + d(N)\ . \end{array}$$

1. *Show that there is a strategy to reduce a λ-term M to a normal form N such that:*

$$|N| \leq |M|^{d(M)}\ .$$

2. *Show that the reduction relation $\rightarrow_{\mathsf{let}}$ is locally confluent.*

How hard is it to *decide* if a λ-term is typable in the *ML* system? Well, in *theory* it is hard but in *practice* it is easy! The characterization via propositional typing with let expansion shows that the problem can be solved in *exponential time*: (1) let-expand the λ-term (exponential penalty), (2) reduce the propositional type-inference problem to a unification problem (efficient), and (3) solve the unification problem (efficient).

In fact one can show that *any decision problem* that runs in exponential time can be coded as an *ML* type inference problem. Hence any algorithm (including the symbolic one) that solves the problem will run in *at least exponential time*. The good news are that the complexity is exponential in the *let-depth* (example next) of the λ-term and that deeply nested chains of let-definitions do not seem to appear in practice.

Example 244 *Here is a way to blow up ML type inference.*

$$\begin{array}{lll} P & \equiv \lambda x,y,z.zxy & : t_1 \to t_2 \to (t_1 \to t_2 \to t_3) \to t_3 \\ M_1 & \equiv \lambda y.Pyy & : t_1 \to (t_1 \to t_1 \to t_2) \to t_2 \\ M_2 & \equiv \lambda y.M_1(M_1 y) & : t_1 \to (((t_1 \to t_1 \to t_2) \to t_2) \to \\ & & \quad ((t_1 \to t_1 \to t_2) \to t_2) \to t_3) \to t_3 \\ M_3 & \equiv \lambda y.M_2(M_2 y) & : \cdots \end{array}$$

The number of distinct type variables and the size of the principal type (roughly) doubles at each step so that inferring the principal type of M_6 is already problematic.

12.4 Summary and references

Universally quantified types are the types of *polymorphic* λ-terms. In particular we have considered a *predicative/stratified* form of universal quantification (as used in *ML*). It turns out that the type inference techniques developed in chapter 11 can be *extended* to predicative polymorphism. The complexity of type inference is then *exponential* in the number of nested let-definitions. Still the approach works well because these complex definitions do not seem to arise in practice. The design of a polymorphic type system for *ML* language is due to [Mil78, LM82]. The complexity of the type inference problem is characterized in [KTU90, Mai90]. The book [Mit96] contains a detailed analysis of the type inference algorithm described in Table 12.4.

Chapter 13

Impredicative polymorphic types

In chapter 12, we have introduced universally quantified types and observed that these types can be regarded as the types of polymorphic functions. In that context, a universally quantified type lives in a higher universe of *type schemas*. In this chapter, we consider an alternative approach where a universally quantified type is still an ordinary type. Then one speaks of *impredicative* types as opposed to the *predicative* types introduced in chapter 12. In order to formalize impredicative types we introduce an extension of the propositional type system presented in chapter 10 known as *system F*.

A strong point of system F is its expressive power. In particular, we show that the addition of impredicative universal quantification suffices to represent product, sum, and existential types. The reader is supposed to be familiar with the usage of product and sum types in programming. As for existential types, we shall see that they arise naturally when hiding the representation details of a data type.

We also provide an encoding of inductively defined data structures such as natural numbers, lists, and trees, and of the iterative functions definable on them (iterative functions are related to the primitive recursive functions introduced in chapter 6).

While being quite expressive, system F can still be regarded as a logical system. In particular, λ-terms typable in system F are strongly normalizing. This is a difficult result that relies on a generalization of the reducibility candidates technique introduced in chapter 10.

13.1 System F

System F is a *logical system* obtained from the propositional intuitionistic system (propositional types as far as we are concerned) by introducing second order quantification. At the type level, we can quantify over type variables:

$$A \equiv \forall\, t\ \ (t \to t)\ .$$

At the term level, we can abstract with respect to a type and apply a λ-term to a type. For instance, we can define a a 'polymorphic' identity $pid \equiv \lambda t.\lambda x : t.x$ with the type A above. By applying pid to the basic type nat, we obtain an identity $pid\ nat$ of type $nat \to nat$. However, we may also apply pid to the type A itself to obtain an identity $pid\ A$ of type $(A \to A)$. In System F, the type quantification in the type A quantifies on *all types* including A itself. One

says that the type system is *impredicative*, as opposed to the *predicative/stratified* system we have considered in chapter 12.

Table 13.1 defines the syntax of types and λ-terms where we denote with $\mathsf{ftv}(\Gamma)$ the collection of type variables that occur free in types occurring in the (type) context Γ.

Table 13.2 introduces the typing rules in Church-style and the reduction rules of system F. The novelties with respect to the system for propositional types (Table 10.1) are represented by the typing rules $(\forall_I)$ and $(\forall_E)$ and the (β_t)-rule for reducing the application of a type abstraction to a type. We stress that in this chapter the (β) and (β_t) rules, as well as the following (η) and (η_t) rules, can be applied in any context.

Exercise 245 *Show that without the side condition '$t \notin \mathsf{ftv}(\Gamma)$' in rule $(\forall_I)$, one can build a closed λ-term of type A, for any type A. In other terms, without the side condition the system is logically inconsistent!*

As usual, we can add *extensional* rules. The (η) and (η_t) reduction rules (applicable in any context) are the following:

$$\begin{array}{lll} (\lambda x : A.Mx) & \to M \quad \text{if } x \notin \mathsf{fv}(M) & (\eta) \\ (\lambda t.Mt) & \to M \quad \text{if } t \notin \mathsf{ftv}(M) & (\eta_t)\ . \end{array}$$

We leave it to the reader the check that in system F with the β and β_t-rules (and possibly with the η and η_t rules): (1) typing is preserved by reduction, and (2) reduction is locally confluent. In section 13.3, we shall prove that typable λ-terms are *strongly normalizable*; thus confluence will follow from local confluence.

As a first example of the expressivity of second order quantification, we consider the representation of product, sum, and existential types in system F. The typing rules and the reduction rules are introduced in Table 13.3. The reader should be familiar with the rules for product and sum which have already been introduced in Table 10.4. On the other hand, the rules for existential types are new and deserve some comments. A λ-term of existential type $\exists t.A$ is (up to conversion) a pair composed of a type B and a λ-term of type $[B/t]A$. Existential types can be used to hide the details of the implementation of a data type and as such they can be regarded as 'abstract data types'. For instance, suppose we want to represent sets of numbers with operations to create the empty set, test membership, insert a number in the set, and remove a number from the set. Assuming, $\mathbf{1}$ is the unit type, $\mathbf{N}$ is the type for natural numbers, and $\mathbf{B}$ the type for booleans, we could specify the signature of a *set* data type as:

$$A \equiv \exists t.((1 \to t) \times (\mathbf{N} \to t \to \mathbf{B}) \times (\mathbf{N} \to t \to t) \times (\mathbf{N} \to t \to t))\ . \tag{13.1}$$

tid	$::= t \mid s \mid \ldots$	(type variables)
A	$::= tid \mid A \to A \mid \forall tid.A$	(types)
id	$::= x \mid y \mid \ldots$	(variables)
M	$::= id \mid \lambda id : A.M \mid MM \mid \lambda tid.M \mid MA$	(λ-terms)
Γ	$::= id : A, \ldots, id : A$	(contexts)

Table 13.1: Syntax of system F: types and λ-terms (Church style)

Typing rules

$$(asmp)\quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$(\to_I)\quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad (\to_E)\quad \frac{\Gamma \vdash M : A \to B \;\; \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$(\forall_I)\quad \frac{\Gamma \vdash M : A \;\; t \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \lambda t.M : \forall t.A} \qquad (\forall_E)\quad \frac{\Gamma \vdash M : \forall t.A}{\Gamma \vdash MB : [B/t]A}$$

Reduction rules (in any context)

$$\begin{array}{lll} (\lambda x : A.M)N & \to [N/x]M & (\beta) \\ (\lambda t.M)A & \to [A/t]M & (\beta_t) \end{array}$$

Table 13.2: Typing (Church-style) and reduction rules in system F

We could then produce a concrete implementation of the data type by instantiating the type t, say, with the type of the lists of natural numbers along with the implementations of the operations mentioned above. We stress that the type A above just describes the *signature* of a *set* data type but *not* its expected *behavior*. For instance, there is no guarantee that inserting a number in a set and then removing it produces a set which equals the original one.

Table 13.4 describes an encoding of product, sum, and existential types and λ-terms in system F. This encoding is quite good, as shown by the following proposition, and it can be used, *e.g.*, to reduce the strong normalization of the extended system to the strong normalization of system F.

Proposition 246 *Suppose $\Gamma \vdash M : A$ in the system F extended with product, sum, and existential types (table 13.3). Then, the encoding described in Table 13.4 preserves typing and reduction. Namely, (1) $\underline{\Gamma} \vdash \underline{M} : \underline{A}$ and (2) if $M \to N$ then $\underline{M} \xrightarrow{*} \underline{N}$.*

Proof. (1) First check that the type encoding commutes with substitution. Then proceed by induction on the proof of $\Gamma \vdash M : A$.

(2) First check that the term encoding commutes with substitution. Then proceed by case analysis on the redex. □

Exercise 247 *Show that for every type context Γ and λ-term M there is at most one type A such that $\Gamma \vdash M : A$ is derivable according to the rules in Tables 13.2 and 13.3, and that in this case the derivation is unique. What happens if we remove the type labels attached to the operators in_1, in_2, and pack?*

13.2 Inductive types and iterative functions

Iterative functions are defined on the ground (with no variables), first-order terms over a signature Σ. The basic idea is to define a function *by induction on the structure of a ground term*, hence we have as many cases as function symbols in the signature Σ. Let us consider

Typing rules

$$\frac{\Gamma \vdash M_i : A_i \quad i = 1,2}{\Gamma \vdash \langle M_1, M_2 \rangle : A_1 \times A_2} \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_1(M) : A_1} \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_2(M) : A_2}$$

$$\frac{\Gamma \vdash M : A_1}{\Gamma \vdash \mathsf{in}_1^{\mathsf{A}_1+\mathsf{A}_2}(M) : A_1 + A_2} \qquad \frac{\Gamma \vdash M : A_2}{\Gamma \vdash \mathsf{in}_2^{\mathsf{A}_1+\mathsf{A}_2}(M) : A_1 + A_2} \qquad \frac{\Gamma \vdash M : A_1 + A_2 \quad \Gamma \vdash N_i : A_i \to C \quad i = 1,2}{\Gamma \vdash \mathsf{case}(M, N_1, N_2) : C}$$

$$\frac{\Gamma \vdash M : [B/t]A}{\Gamma \vdash \mathsf{pack}^{\exists t.A}(B, M) : \exists t.A} \qquad \frac{\Gamma \vdash M : \exists t.A \quad \Gamma \vdash N : \forall t.(A \to C) \quad t \notin \mathsf{ftv}(C)}{\Gamma \vdash \mathsf{unpack}(M, N) : C}$$

Reduction rules (in any context)

$$\begin{array}{lcll} \pi_i\langle M_1, M_2\rangle & \to & M_i & i = 1,2 \\ \mathsf{case}(\mathsf{in}_\mathsf{i}^{\mathsf{A}+\mathsf{B}} M) N_1 N_2 & \to & N_i M & i = 1,2 \\ \mathsf{unpack}(\mathsf{pack}^{\exists t.A}(B, M), N) & \to & NBM & \end{array}$$

Table 13.3: Product, sum, and existential types

Type encoding

$$\begin{array}{ll} \underline{t} & = t \\ \underline{A \to B} & = \underline{A} \to \underline{B} \\ \underline{\forall t.A} & = \forall t.\underline{A} \\ \underline{A_1 \times A_2} & = \forall s.(\underline{A_1} \to \underline{A_2} \to s) \to s \\ \underline{A_1 + A_2} & = \forall s.(\underline{A_1} \to s) \to (\underline{A_2} \to s) \to s \\ \underline{\exists t.A} & = \forall s.(\forall t.(\underline{A} \to s)) \to s \end{array}$$

Term encoding

$$\begin{array}{lll} \langle , \rangle & = \lambda x_1 : \underline{A_1}, x_2 : \underline{A_2}.\lambda s.\lambda p : \underline{A_1} \to (\underline{A_2} \to s).p x_1 x_2 & \\ \pi_i & = \lambda p : \underline{A_1 \times A_2}.p\underline{A_i}(\lambda x_1 : \underline{A_1}, x_2 : \underline{A_2}.x_i) & (i = 1,2) \\ \mathsf{in}_\mathsf{i}^{\mathsf{A}_1+\mathsf{A}_2} & = \lambda x : \underline{A_i}.\lambda s.\lambda y_1 : \underline{A_1} \to s.\lambda y_2 : \underline{A_2} \to s.y_i x & (i = 1,2) \\ \mathsf{case} & = \lambda x : \underline{A_1 + A_2}.\lambda s.\lambda y_1 : \underline{A_1} \to s, y_2 : \underline{A_2} \to s.x s y_1 y_2 & \\ \mathsf{pack}^{\exists t.A} & = \lambda t.\lambda x : \underline{A}.\lambda s.\lambda y : \forall t.\underline{A} \to s.y\ t\ x & \\ \mathsf{unpack} & = \lambda x : \underline{\exists t.A}.\lambda y : \forall t.\underline{A} \to C.x C y & \end{array}$$

Table 13.4: Representation of product, sum, and existential types in system F

the signature of *tally natural numbers* $\Sigma = \{\mathsf{S}^1, \mathsf{Z}^0\}$ and let $T = T_\Sigma(\emptyset)$ be the set of ground terms. Given $g : T^n \to T$ and $h : T^{n+1} \to T$ the function $f : T^{n+1} \to T$ is defined by *iteration* by the following term rewriting rules:

$$f(\mathsf{Z}, y^*) \;\; \to \;\; g(y^*) \; , \qquad f(\mathsf{S}(x), y^*) \;\; \to \;\; h(f(x, y^*), y^*) \; .$$

At first sight this is less powerful than primitive recursive definitions because the function h *does not* depend directly on x.

However, one can first define *pairing* and *projections* and then show that a function f defined by *primitive recursion* such as:

$$f(\mathsf{Z}, y^*) \;\; \to \;\; g(y^*) \; , \qquad f(\mathsf{S}(x), y^*) \;\; \to \;\; h(f(x, y^*), x, y^*) \; ,$$

can also be defined by *iteration* as follows:

$$\begin{array}{lcl} f'(\mathsf{Z}, y^*) & \to & \langle g(y^*), \mathsf{Z} \rangle \; , \\ f'(\mathsf{S}(x), y^*) & \to & h'(f'(x, y^*), y^*) \; , \\ h'(x, y^*) & \to & \langle h(\pi_1(x), \pi_2(x), y^*), \mathsf{S}(\pi_2(x)) \rangle \; . \end{array}$$

One checks by induction on $x \in T$ that: $f'(x, y^*) = \langle f(x, y^*), x \rangle$, and from f' one obtains f by projection.

Exercise 248 (predecessor, equality) *Give a primitive recursive definition of the predecessor function p where $p(0) = 0$. Then transform the definition into an iterative definition and derive a λ-term, typable in system F, to compute the predecessor function on Church numerals (cf. exercise 142). Further, derive a λ-term that checks the equality of two Church numerals.*

Definition 249 (iterative functions) *Let Σ be a signature with function symbols (constructors) c_i, where $ar(c_i) = n_i$, for $i = 1, \ldots, k$. Let $T = T_\Sigma(\emptyset)$ be the closed first-order terms over the signature. The collection of* iterative functions *is the smallest set such that:*

- *The functions induced by the constructors and the projection functions are iterative functions.*
- *The set is closed under* composition*, namely if $g : T^n \to T$ and $h_i : T^m \to T$, for $i = 1, \ldots, n$, are iterative functions then $g(h_1, \ldots, h_n)$ is an iterative function.*
- *The set is closed under* iteration*, namely if $h_i : T^{n_i + m} \to T$, for $i = 1, \ldots, k$, are iterative functions then the function $f : T^{m+1} \to T$ such that:*

  $$f(c_i(x_1, \ldots, x_{n_i}), y^*) = h_i(f(x_1, y^*), \ldots, f(x_{n_i}, y^*), y^*) \qquad (\textit{for } i = 1, \ldots, k) \; ,$$

  is an iterative function.

Table 13.5 explains how to associate: (1) with a *signature* Σ a *type* $\underline{\Sigma}$ of system F, (2) with a *constructor* of the signature Σ a *closed λ-term* of system F of the appropriate type, and (3) with a *ground term* a over the signature Σ a λ-term of system F $\underline{a}$ of type $\underline{\Sigma}$.

$$
\begin{array}{lll}
\Sigma & = \{c_i^{n_i} : \underbrace{T \times \cdots \times T}_{n_i \ times} \to T \mid i = 1, \ldots, k\} & \\
\underline{\Sigma} & \equiv \forall t.A_1 \to \cdots \to A_k \to t, \quad \text{where: } A_i \equiv \underbrace{\underline{t} \to \cdots \to \underline{t}}_{n_i \ times} \to t \ , & \\
\underline{c_i^{n_i}} & \equiv \lambda y_1 : \underline{\Sigma} \ldots \lambda y_{n_i} : \underline{\Sigma}. \quad \lambda t.\lambda x_1 : A_1 \ldots \lambda x_k : A_k. & \\
 & \quad x_i(y_1 t x_1 \cdots x_k) \cdots (y_{n_i} t x_1 \cdots x_k) : \underbrace{\underline{\Sigma} \to \cdots \to \underline{\Sigma}}_{n_i \ times} \to \underline{\Sigma} & \\
\underline{a} & \equiv \lambda t.\lambda x_1 : A_1 \ldots \lambda x_k : A_k.[\![a]\!] \text{ , with: } [\![c_i^{n_i}(a_1, \ldots, a_{n_i})]\!] \equiv x_i [\![a_1]\!] \cdots [\![a_{n_i}]\!] \ . &
\end{array}
$$

Table 13.5: Encoding of signatures, constructors, and ground terms in system F

Example 250 (tally natural numbers) *If we apply the coding method to the signature* $\Sigma = \{\mathsf{S}^1, \mathsf{Z}^0\}$ *of* tally natural numbers *we obtain the type:*

$$\underline{\Sigma} \equiv \forall t.(t \to t) \to (t \to t) \ .$$

Then we represent the constructors in the signature with the λ*-terms:*

$$
\begin{array}{lll}
\underline{S} & \equiv \lambda y : \underline{\Sigma}.\lambda t.\lambda x_1 : t \to t.\lambda x_2 : t.x_1(y t x_1 x_2) & : \underline{\Sigma} \to \underline{\Sigma} \\
\underline{Z} & \equiv \lambda t.\lambda x_1 : t \to t.\lambda x_2 : t.x_2 & : \underline{\Sigma} \ .
\end{array}
$$

The term $n \equiv \mathsf{S}^n\mathsf{Z}$*,* $n \geq 0$*, is represented (up to conversion) by the* λ*-term:*

$$\underline{n} \equiv \lambda t.\lambda x_1 : t \to t.\lambda x_2 : t.x_1^n x_2 : \underline{\Sigma} \ ,$$

which is a typed version of the Church numeral *presented in section 7.3. We notice that:*

$$\underline{S}\ \underline{n} \ \to \lambda t.\lambda x_1 : t \to t.\lambda x_2 : t.x_1(\underline{n}\ t\ x_1\ x_2) \ \stackrel{*}{\to} \lambda t.\lambda x_1 : t \to t.\lambda x_2 : t.x_1(x_1^n\ x_2) \ \equiv \underline{n+1} \ .$$

Exercise 251 *Make explicit the coding of the following signatures: (1) The signature with* no operation*. (2) The signature with* two 0-ary operations *(the 'booleans'). (3) The signature of* binary words*. (4) The signature of* binary trees*.*

Proposition 252 *There is a bijective correspondence between the* ground terms *over a signature* Σ *and the* closed λ-terms *of system F of the corresponding type* $\underline{\Sigma}$ *modulo* $\beta\eta$*-conversion.*

Proof. Let M be a closed λ-term of system F in β-normal form of type $\underline{\Sigma}$, where $\underline{\Sigma}$ is defined according to the rules in Table 13.5. The existence of the β-normal form will be proved next. M has to have the shape:

$$M \equiv \lambda t.\lambda x_1 : A_1 \ldots \lambda x_i : A_i.M' \quad i \leq k \ .$$

If $i < k$ and M' is not a λ-abstraction then M' has the shape $(\cdots(x_j M_1)\cdots M_h)$ and so we can η-expand M' without introducing a β-redex. By iterated η-expansions we arrive at a λ-term in β normal form of the shape:

$$\lambda t.\lambda x_1 : A_1 \ldots \lambda x_k : A_k.M'',$$

where M'' has type t, it is in β normal form, and may include free variables $x_1, \ldots, x_k$. We note that the types of the variables x_i do not contain second order quantifications. We claim that M'' cannot contain a λ-abstraction:

- A λ-abstraction on the left of an application would contradict the hypothesis that M is in β normal form.
- A λ-abstraction on the right of an application is incompatible with the 'first order' types of the variables A_i.

We have shown that a closed λ-term of type $\underline{\Sigma}$ is determined up to $\beta\eta$ conversion by a λ-term M'' which is a well-typed combination of the variables x_i, for $i = 1, \ldots, k$. Since each variable corresponds to a constructor of the signature we can conclude that there is a unique ground term over the signature which corresponds to M''. □

Remark 253 *The rule* (η) *is needed to have a bijection between ground terms of the signature* Σ *and closed* λ*-terms of type* $\underline{\Sigma}$*. For instance, with reference to example 250 (tally natural numbers), there are two distinct* λ*-terms in* β*-normal form corresponding to the numeral* 1*, namely* $\underline{1}$ *and* $\lambda t.\lambda x_1 : t \to t.x_1$*.*

Definition 254 *A function* $f : T^n \to T$ *over a signature* Σ *is representable (with respect to the proposed coding) if there is a closed* λ*-term* $M : \underline{\Sigma}^n \to \underline{\Sigma}$*, such that for any vector of ground terms* a^**:*

$$M\underline{a^*} =_{\beta\eta} \underline{f(a^*)} \ .$$

Proposition 255 *All* iterative functions *over a signature* Σ *are representable.*

Proof. We proceed by induction on the definition of iterative function. The interesting case is iteration. Let $h_i : T_S^{n_i+m} \to T_S$ be iterative functions for $i = 1, \ldots, k$, and the function $f : T_S^{m+1} \to T_S$ be defined by:

$$f(x^*, c_i(y^*)) = h_i(x^*, f(x^*, y_1), \ldots, f(x^*, y_{n_i})) \quad i = 1, \ldots, k \ , \tag{13.2}$$

where $x^* \equiv x_1, \ldots, x_m$.[1] We represent f with the function:

$$\underline{f} \equiv \lambda x_1 : \underline{\Sigma}. \ldots \lambda x_m : \underline{\Sigma}.\lambda x : \underline{\Sigma}.x\underline{\Sigma}(\underline{h_1}x^*) \cdots (\underline{h_k}x^*) \ ,$$

where we know inductively that $\underline{h_i}$ represents h_i. Note that iteration is already built into the representation of the data. We prove by induction on the structure of a ground term a that for any vector of ground terms b^*, $\underline{f}\ \underline{b^*}\ \underline{a} =_{\beta\eta} \underline{f(b^*, a)}$.

- If $a \equiv c_i^0$ then

$$\underline{f}\ \underline{b^*}\ \underline{c_i^0} \to^* \underline{c_i^0}\underline{\Sigma}(\underline{h_1}\underline{b^*}) \cdots (\underline{h_k}\underline{b^*}) \to^* \underline{h_i}\underline{b^*} =_{\beta\eta} \underline{h_i(b^*)} \ ,$$

where the last step holds by induction hypothesis on h_i.

- If $a \equiv c_i^n(a_1, \ldots, a_n)$ then:

$$\begin{aligned} \underline{f(b^*, c_i(a_1, \ldots, a_n))} &= \underline{h_i(b^*, f(b^*, a_1), \ldots, f(b^*, a_n))} \\ &=_{\beta\eta} \underline{h_i}\ \underline{b^*}\ \underline{f(b^*, a_1)} \ldots \underline{f(b^*, a_n)} \quad \text{(by induction hypothesis on } h_i\text{)}. \end{aligned}$$

[1] In this proof, it is convenient to write the additional parameters x^* before the main argument of the iteration.

On the other hand, we compute:

$$\begin{array}{l}\underline{f}\ \underline{b^*}\ \underline{c_i^n(a_1,\ldots,a_n)}\\ \to \underline{c_i^n(a_1,\ldots,a_n)}\underline{\Sigma}(\underline{h_1}\ \underline{b^*})\cdots(\underline{h_k}\ \underline{b^*})\\ \to (\underline{h_i}\ \underline{b^*})(\underline{a_1}\underline{\Sigma}(\underline{h_1}\ \underline{b^*})\cdots(\underline{h_k}\ \underline{b^*}))\cdots(\underline{a_n}\underline{\Sigma}(\underline{h_1}\ \underline{b^*})\cdots(\underline{h_k}\ \underline{b^*}))\ .\end{array}$$

Also, by induction hypothesis on a, we have for $i = 1,\ldots,n$:

$$\underline{f(b^*,a_i)} =_{\beta\eta} \underline{f}\underline{b^*}\underline{a_i} \xrightarrow{*} \underline{a_i}\underline{\Sigma}(\underline{h_1}\underline{b^*})\cdots(\underline{h_k}\underline{b^*}))\ .$$

Hence, by combining the computations above, we obtain:

$$\begin{array}{rl}\underline{f}\ \underline{b^*}\ \underline{c_i^n(a_1,\ldots,a_n)} & =_{\beta\eta} \underline{h_i}\ \underline{b^*}\ \underline{f(b^*,a_1)}\ldots\underline{f(b^*,a_n)}\\ & =_{\beta\eta} \underline{f(b^*,c_i(a_1,\ldots,a_n))}\ .\end{array}$$

□

Example 256 *Suppose T is the set of tally natural numbers and $g : T \to T$ and $h : T^2 \to T$. The iteration $it(h,g)$ of h and g must satisfy:*

$$\begin{array}{ll}it(h,g)(\mathsf{Z},y) & = g(y)\\ it(h,g)(\mathsf{S}(x),y) & = h(it(h,g)(x,y),y)\end{array}$$

In the pure *λ-calculus, we would define:*

$$it \ \equiv\ \lambda h.\lambda g.\lambda x.\lambda y.\ x\ (\lambda z.h\ z\ y)\ (g\ y)\ .$$

For $\underline{\Sigma} \equiv \forall t.(t \to t) \to (t \to t)$, this is typable *as follows:*

$$\begin{array}{rcl}it & \equiv & \lambda h : \underline{\Sigma} \to (\underline{\Sigma} \to \underline{\Sigma}).\lambda g : \underline{\Sigma} \to \underline{\Sigma}.\lambda x : \underline{\Sigma}.\lambda y : \underline{\Sigma}.\\ & & x\ \underline{\Sigma}\ (\lambda z : \underline{\Sigma}.h\ z\ y)\ (g\ y)\\ & : & (\underline{\Sigma} \to (\underline{\Sigma} \to \underline{\Sigma})) \to (\underline{\Sigma} \to \underline{\Sigma}) \to (\underline{\Sigma} \to (\underline{\Sigma} \to \underline{\Sigma}))\ .\end{array}$$

Notice that this would not work *with a propositional type of the shape $(B \to B) \to (B \to B)$!*

Example 257 *One can also handle the case of signatures which are defined* parametrically *with respect to a collection of data. For instance $List(D)$ is the* signature of lists *whose elements belong to the set D. This signature is equipped with the constructors:*

$$\mathsf{nil} : List(D), \qquad \mathsf{cons} : D \times List(D) \to List(D)\ .$$

One can define iterative functions *over $List(D)$ and show that these functions can be represented in system F for a suitable embedding of the closed λ-terms in system F. The sort $List(D)$ is coded by the type:*

$$\forall t.t \to (r \to t \to t) \to t\ ,$$

where r is a type variable, and generic elements in D are represented by (free) variables of type r.

13.3 Strong normalization

We now move towards a proof of the announced strong normalization result. The proof is based on a notion of *reducibility candidate* which is an *abstraction* of the notion already considered for the strong normalization of the propositionally typed λ-calculus (chapter 10) and recursive path ordering (chapter 4). In order to make *notation lighter* we shall work with untyped λ-terms obtained from the *erasure* of well-typed λ-terms.

Definition 258 (erasure) *The (type)* erasure *function er takes a typed λ-term and returns an untyped λ-term. It is defined by induction on the structure of the λ-term as follows:*

$$er(x) = x, \qquad er(\lambda x : A.M) = \lambda x.er(M), \qquad er(MN) = er(M)er(N),$$
$$er(\lambda t.M) = er(M), \qquad er(MA) = er(M) \ .$$

In system F, we distinguish *two flavors of β-reduction*: the one involving a redex $(\lambda x : A.M)N$ which we call simply (β) and the one involving a redex $(\lambda t.M)A$ which we call (β_t). Erasing type information we eliminate the reductions (β_t). However this does *not* affect the strong normalization property as shown in the following.

Proposition 259 (erasure vs. typed) *Let M be a well-typed λ-term in system F. Then:*

1. *If $M \to_\beta N$ then $er(M) \to_\beta er(N)$.*
2. *If $M \to_{\beta_t} N$ then $er(M) \equiv er(N)$.*
3. *If M may diverge then $er(M)$ may diverge.*

Proof. Properties (1) and (2) are left to the reader. For (3), we observe that sequences of β_t-reductions always terminate as the size of the λ-term shrinks. Hence we can extract an infinite reduction of $er(M)$ from an infinite reduction of M. □

We can now address the *key* issue. Suppose we want to adapt the *semantic* method already used in the propositional case. What is the interpretation of $A \equiv \forall t.(t \to t)$? We have to build first a *universe* $\mathcal{U}$ of type interpretations where each type interpretation is a set of λ-terms. Then we could require:

$$[\![A]\!] = \{M \mid \forall\, X \in \mathcal{U} \ \ \forall\, N \in X \ \ (MN \in X)\} \ .$$

Technically, the type interpretations are the so-called reducibility candidates and are defined as follows. Let SN be the collection of untyped λ-terms which are strongly normalizable with respect to the (β) rule. We shall use $P, Q, \ldots$ to denote the untyped λ-terms (as opposed to the typed ones which are denoted with $M, N, \ldots$).

Definition 260 (candidates) *A set X of λ-terms is a* reducibility candidate *if:*

1. *$X \subseteq SN$.*
2. *$Q_i \in SN$, $i = 1, \ldots, n$, $n \geq 0$ implies $xQ_1, \ldots, Q_n \in X$.*
3. *$[Q/x]PQ_1, \ldots, Q_n \in X$ and $Q \in SN$ implies $(\lambda x.P)QQ_1, \ldots, Q_n \in X$.*

We denote with RC the collection of reducibility candidates.

Remark 261 *We have made into a* definition *the* properties *stated in proposition 216 of the interpretation of propositional types.*

Proposition 262 (properties reducibility candidates) *The following properties hold.*

1. *The set SN is a reducibility candidate.*
2. *If $X \in RC$ then $X \neq \emptyset$.*
3. *The collection RC is closed under arbitrary intersections.*
4. *If $X, Y \in RC$ then the following set is a reducibility candidate:*

$$X \to Y = \{M \mid \forall\, N \in X \;\; (MN \in Y)\} \; .$$

Proof. We abbreviate $Q_1, \ldots, Q_n$ with Q^*. We recall (definition 213) that if $P \in SN$ then $depth(P)$ is the length of its longest reduction.

(1) As in the propositional case, we observe that $[Q/x]PQ^* \in SN$ and $Q \in SN$ implies $(\lambda x.P)QQ^* \in SN$. This is an induction on $depth(P)+depth(Q)+depth(Q_1)+\cdots+depth(Q_n)$.

(2) By definition, $x \in X$.

(3) Immediate.

(4) Here we see the use of the 'saturation' condition (3) in definition 260. □

Next we define a type interpretation.

Definition 263 (type interpretation) *Let Tvar be the set of type variables. Given a type environment $\eta : Tvar \to RC$ we interpret types as follows:*

$$\begin{array}{lcl} [\![t]\!]\eta & = & \eta(t) \\ [\![A \to B]\!]\eta & = & [\![A]\!]\eta \to [\![B]\!]\eta \\ [\![\forall t.A]\!]\eta & = & \bigcap_{X \in RC} [\![A]\!]\eta[X/t] \; . \end{array}$$

We remark that the interpretations of a functional type and a universal type are well-defined because of propositions 262(4) and 262(3), respectively. Strong normalization follows from the *soundness of the interpretation* which is stated as follows.

Proposition 264 (soundness) *Let η be a type environment and $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ a derivable judgment. If $P_i \in [\![A_i]\!]\eta$, for $i = 1, \ldots, n$ then*

$$[P_1/x_1, \ldots, P_n/x_n]er(M) \in [\![B]\!]\eta \; .$$

Proof. We abbreviate $[P_1/x_1, \ldots, P_n/x_n]$ with $[P^*/x^*]$. We proceed by induction on the typing proof.

$(asmp)$ follows by definition.

($\rightarrow_I$) We have to show:

$$\lambda x.[P^*/x^*]er(M) \in [\![A \rightarrow B]\!]\eta \ .$$

By inductive hypothesis, we know: $[P^*/x^*][P/x]er(M) \in [\![B]\!]\eta$, for all $P \in [\![A]\!]\eta$. We conclude by using the properties of reducibility candidates.

($\rightarrow_E$) By the definition of $\rightarrow$.

($\forall_I$) We have to show:

$$[P^*/x^*]er(M) \in \bigcap_{X \in RC} [\![B]\!]\eta[X/t] \ .$$

By the side condition on the typing rule, we know: $[\![A_i]\!]\eta = [\![A_i]\!]\eta[X/t]$, for an arbitrary $X \in RC$. By inductive hypothesis: $[P^*/x^*]er(M) \in [\![B]\!]\eta[X/t]$, for an arbitrary $X \in RC$.

($\forall_E$) We have to show:

$$[P^*/x^*]er(M) \in [\![B]\!]\eta[[\![A]\!]\eta/t] \ .$$

By inductive hypothesis: $[P^*/x^*]er(M) \in \bigcap_{X \in RC} [\![B]\!]\eta[X/t]$. Choose $X = [\![A]\!]\eta$. □

Corollary 265 (strong normalization) *If $\Gamma \vdash M : A$ in system F, then M is strongly normalizing.*

Proof. We note that $\forall\, A, \eta, x \ \ (x \in [\![A]\!]\eta)$. Then we apply proposition 264 with $P_i \equiv x_i$, and derive that: $er(M) \in [\![A]\!]\eta \subseteq SN$. By proposition 259, we conclude that M is strongly normalizing. □

Exercise 266 (neutral λ-term) *Alternative definitions of reducibility candidates can be found in the literature; one follows. Say that a λ-term is* neutral *if it does not start with a λ-abstraction. Define $Red(M) = \{M' \mid M \rightarrow_\beta M'\}$. The collection RC' is given by the sets X of strongly normalizing λ-terms satisfying the following conditions:*

1. *$M \in X$ and $M \rightarrow_\beta M'$ implies $M' \in X$.*
2. *M neutral and $Red(M) \subseteq X$ implies $M \in X$.*

Carry on the strong normalization proof using the collection RC'.

13.4 Summary and references

The introduction of second-order quantification *preserves the standard properties* of the propositionally typed calculus: subject reduction, strong normalization, confluence ... while *increasing the expressivity* in a very significant way as one can encode inductive data types and iterative functions. There is one catch however: *type inference becomes undecidable* which is one reason why *ML*-like programming languages adopt a weaker/predicative form of polymorphism. When extended with first-order quantification, system F is the backbone of a higher-order constructive logic (the so called *calculus of constructions* on which the Coq proof assistant is built [CH88]).

The system F has been introduced by Girard in [Gir71] as a tool for the study of the *cut-elimination procedure* in *second order Peano arithmetic* (PA_2). More precisely the *normalization of system F* implies the *termination of the cut-elimination procedure* in PA_2 (and thus the consistency of analysis!). By relying on this strong connection between system F and PA_2 it is proven that all functions that can be shown to be total in PA_2 are *representable* in system F. This is a *huge* collection of total recursive functions that goes well beyond the primitive recursive functions. The connections with the notion of type polymorphism (or type parametricity) arising in programming are noticed in [Rey74] and the relationship between existential types and abstract data types are pointed out in [MP88]. The results on the representation of iterative functions are based on [BB85]. The type inference problem for a Curry-style system F (cf. chapter 12) turns out to be undecidable [Wel99].

Chapter 14

Program transformations

In this chapter, we introduce four program transformations. Each transformation has its own interest. Moreover, when they are put in pipeline they provide a compilation chain from a call-by-value λ-calculus to a register transfer level (RTL) language. A RTL language can be regarded as a machine independent version of assembly code. Functions correspond to assembly level routines and the functions' bodies correspond to sequences of vectors' allocations and vectors' projections ended by a tail recursive call. The compilation chain is summarized in the following diagram:

$$\lambda \quad \stackrel{\mathcal{C}_{cps}}{\rightarrow} \quad \lambda_{cps} \quad \stackrel{\mathcal{C}_{vn}}{\rightarrow} \quad \lambda_{cps,vn} \quad \stackrel{\mathcal{C}_{cc}}{\rightarrow} \quad \lambda_{cc,vn} \quad \stackrel{\mathcal{C}_{h}}{\rightarrow} \quad \lambda_{h,vn} \tag{14.1}$$

The source language is a call-by-value, λ-calculus (cf. chapter 8). The first transformation, called *continuation-passing style* (CPS), internalizes the notion of evaluation context, the second, called *value naming*, assigns a name to every value, the third, called *closure conversion*, internalizes the notion of closure and makes sure functions are closed, *i.e.*, they do not contain free variables, and the last, called *hoisting*, transforms a collection of closed nested function definitions into a collection of possibly open, flat, *i.e.*, without nesting, function definitions.

Since we want to compose these transformations, we make sure the target language of each transformation coincides with the source of the following one. As a matter of fact, all the languages are subsets of the initial source language though their evaluation mechanism is refined along the way. In particular, one moves from an ordinary substitution to a specialized one where variables can only be replaced by other variables.

The approach to compiler correctness is similar to the one considered for the toy compiler of section 1.3. One proves that each transformation is correct in the sense that the object code simulates the source code. Then, by composition, one derives the correctness of the compilation chain.

14.1 Continuation passing style form

The origin of the CPS transformation goes back to so called *double-negation* transformations from classical to intuitionistic/constructive logic. In constructive logic, the formula $\neg\neg t \rightarrow t$ is not derivable but the formula $\neg\neg\neg t \rightarrow \neg t$ is (cf. exercise 200). Then the idea is to transform formulae in classical logic to *negated* formulae in constructive logic so that negation is involutive on the image of the transformation. For some fixed type variable s, let $\neg A =$

$(A \to s)$, and define a transformation of propositional types and type contexts as follows:

$$\underline{t} = t \ , \quad \underline{A \to B} = \underline{A} \to \neg\neg\underline{B} \ , \quad \underline{\emptyset} = \emptyset \ , \quad \underline{\Gamma, x : A} = \underline{\Gamma}, x : \underline{A} \ .$$

What the transformation shows is that for for every formula A provable in classical logic, there is a classicaly equivalent formula $\neg\neg\underline{A}$ which is provable in constructive logic. Now suppose we start with a λ-term of type A, say $\Gamma \vdash M : A$, *i.e.*, with a constructive proof of A. Can we build a λ-term $\underline{M}$ such that $\underline{\Gamma} \vdash \underline{M} : \neg\neg\underline{A}$? For variables and λ-abstractions, the typing suggests directly:

$$\underline{x} = \lambda k.kx \ , \quad \underline{\lambda x.M} = \lambda k.k(\lambda x.\underline{M}) \ .$$

The case for application is a bit more complex, but the reader may easily check that the following does the job:

$$\underline{MN} = \lambda k.\underline{M}(\lambda m.\underline{N}(\lambda n.mnk)) \ .$$

Moreover, the transformed λ-term simulates the original one as soon it is provided with an additional argument which represents the initial evaluation context. For instance, the reader may check that:

$$\underline{((\lambda x.x)y)}(\lambda z.z) \xrightarrow{*}_{\beta} y \ , \tag{14.2}$$

where $\lambda z.z$ stands for the initial evaluation context. As a matter of fact, types are useless in proving the simulation property and they will be omitted in the following formal treatment. However, as we have seen, types shed light on the CPS transformation and we shall come back to them in chapter 15. The reader may have noticed that the reduction (14.2) above performs many 'useless' β-reductions. For this reason, as well as for simplifying the proof strategy, we shall study an *optimized version* of the CPS transformation.

Table 14.1 introduces the source language: a type-free, left-to-right, call-by-value λ-calculus. Notice that for technical reasons we include the variables among the values. Also notice that the calculus is richer than the one studied in chapter 8 in that it includes *let-definitions*, *polyadic abstraction*, and *tupling*, with the related application and projection operators. Polyadic abstraction grants a function the right to take several arguments at once while tupling allows to build vectors of terms. For the sake of readibility, we shall denote *explicitly* the polyadic application with the symbol @. We stress that polyadic abstraction can be simulated by iterated λ-abstraction and tupling can be simulated by iterated pairing. Still, it is worth to take them as primitive in order to simplify the analysis of the following program transformations.

Working with polyadic abstraction and tuples, we need a compact notation to represent sequences of symbols. We shall write X^+ (resp. X^*) for a non-empty (possibly empty) finite sequence $X_1, \ldots, X_n$ of symbols. By extension, $\lambda x^+.M$ stands for $\lambda x_1 \ldots x_n.M$, $[V^+/x^+]M$ stands for $[V_1/x_1, \ldots, V_n/x_n]M$, and $\mathsf{let}\ (x = V)^+\ \mathsf{in}\ M$ stands for $\mathsf{let}\ x_1 = V_1\ \mathsf{in}\ \cdots \mathsf{let}\ x_n = V_n\ \mathsf{in}\ M$. By default, a *term* is a λ-term in the enriched λ-calculus under consideration.

Table 14.2 introduces a fragment of the λ-calculus described in Table 14.1 and a related CPS transformation. An evaluation context E can be represented as a term $\lambda x.E[x]$; in a CPS transformation each function takes its evaluation context, represented as a term, as a fresh additional parameter. The *initial* evaluation context is defined relatively to a fresh variable named '*halt*'.

The syntax of CPS terms is such that in an application all terms are values and this property is preserved by reduction. A corollary of this syntactic restriction is that the redex

SYNTAX

$$
\begin{array}{lll}
V & ::= id \mid \lambda id^{+}.M \mid (V^{*}) & \text{(values)} \\
M & ::= V \mid @(M, M^{+}) \mid \mathsf{let}\ id = M\ \mathsf{in}\ M \mid (M^{*}) \mid \pi_i(M) \mid & \text{(terms)} \\
E & ::= [\,] \mid @(V^{*}, E, M^{*}) \mid \mathsf{let}\ id = E\ \mathsf{in}\ M \mid (V^{*}, E, M^{*}) \mid \pi_i(E) & \text{(evaluation contexts)}
\end{array}
$$

REDUCTION RULES

$$
\begin{array}{lcll}
E[@(\lambda x_1 \ldots x_n.M, V_1, \ldots, V_n)] & \rightarrow & E[[V_1/x_1, \ldots, V_n/x_n]M] & \\
E[\mathsf{let}\ x = V\ \mathsf{in}\ M] & \rightarrow & E[[V/x]M] & \\
E[\pi_i(V_1, \ldots, V_n)] & \rightarrow & E[V_i] & (1 \leq i \leq n)
\end{array}
$$

Table 14.1: A polyadic, call-by-value, λ-calculus: λ

of a CPS term is always at top level, or in another terms, the evaluation context of a CPS term is always the trivial context '$[\,]$'. The reduction rules are essentially those of the λ-calculus modulo the fact that we optimize the rule for the projection to guarantee that CPS terms are closed under reduction. For instance, the term $\mathsf{let}\ x = \pi_1(V_1, V_2)\ \mathsf{in}\ M$ reduces directly to $[V_1/x]M$ rather than going through the intermediate term $\mathsf{let}\ x = V_1\ \mathsf{in}\ M$ which, according to Table 14.2, does *not* belong to the CPS terms. There is a potential ambiguity concerning the CPS transformation of tuples of values. We remove it, by assuming that $(V_1, \ldots, V_n) \mid K$ is transformed according to the case for values. But note that if we follow the general case for tuples we obtain the same result.

Next, we state the properties enjoyed by the presented CPS transformation, which is 'optimized' so as to pre-compute many 'administrative' reductions. In particular, thanks to this optimization, we can show that the CPS transformation of a term such as $E[@(\lambda x.M, V)]$ is a term of the shape $@(\psi(\lambda x.M), \psi(V), K_E)$ for a suitable continuation K_E depending on the evaluation context E.

Proposition 267 (CPS simulation) *Let M be a term of the λ-calculus. If $M \rightarrow N$ then $\mathcal{C}_{cps}(M) \stackrel{*}{\rightarrow} \mathcal{C}_{cps}(N)$.*

PROOF. The proof takes the following steps.

1. We show that for all values V, terms M, and continuations $K \neq x$:

$$[V/x]M \mid [\psi(V)/x]K \equiv [\psi(V)/x](M \mid K)\ .$$

We proceed by induction on M.

variable By case analysis: $M \equiv x$ or $M \equiv y \neq x$.

$\lambda z^{+}.M$ By case analysis on K which is either a variable or a function. We develop the second case with $K \equiv \lambda y.N$. We observe:

$$
\begin{array}{l}
[V/x](\lambda z^{+}.M) \mid [\psi(V)/x]K \\
\equiv [\lambda z^{+}, k.([V/x]M \mid k)/y][\psi(V)/x]N \\
\equiv [\lambda z^{+}, k.[\psi(V)/x](M \mid k)/y][\psi(V)/x]N \\
\equiv [\psi(V)/x][\lambda z^{+}, k.(M \mid k)/y]N \\
\equiv [\psi(V)/x]((\lambda z^{+}.M) \mid K)\ .
\end{array}
$$

$@(M_0, \ldots, M_n)$ We apply the inductive hypothesis on $M_0, \ldots, M_n$ as follows:

$$\begin{array}{l}
[\psi(V)/x](@(M_0, \ldots, M_n) \mid K) \\
\equiv [\psi(V)/x](M_0 \mid \lambda x_0 \ldots M_n \mid \lambda x_n.@(x_0, \ldots, x_n, K)) \\
\ldots \\
\equiv [V/x]M_0 \mid \lambda x_0 \ldots [\psi(V)/x](M_n \mid \lambda x_n.@(x_0, \ldots, x_n, K)) \\
\equiv [V/x]M_0 \mid \lambda x_0 \ldots [V/x]M_n \mid \lambda x_n.@(x_0, \ldots, x_n, [\psi(V)/x]K) \\
\equiv [V/x]@(M_0, \ldots, M_n) \mid [\psi(V)/x]K \ .
\end{array}$$

Note that in this case the substitution $[\psi(V)/x]$ may operate on the continuation. The remaining cases (pairing, projection, let-definition) follow a similar pattern and are omitted.

2. The evaluation contexts for the λ-calculus described in Table 14.1 can also be specified 'bottom up' as follows:

$$E \quad ::= \quad [\,] \mid E[@(V^*, [\,], M^*)] \mid E[\mathsf{let}\ id = [\,]\ \mathsf{in}\ M] \mid E[(V^*, [\,], M^*)] \mid E[\pi_i([\,])] \ .$$

Following this specification, we associate with an evaluation context E a continuation K_E as follows:

$$\begin{array}{lcl}
K_{[\,]} & = & \lambda x.@(\mathit{halt}, x) \\
K_{E[@(V^*,[\,],M^*)]} & = & \lambda x.M^* \mid \lambda y^*.@(\psi(V)^*, x, y^*, K_E) \\
K_{E[\mathsf{let}\ x=[\,]\ \mathsf{in}\ N]} & = & \lambda x.N \mid K_E \\
K_{E[(V^*,[\,],M^*)]} & = & (\lambda x.M^* \mid \lambda y^*.(\psi(V)^*, x, y^*)) \mid K_E \\
K_{E[\pi_i([\,])} & = & \lambda x.\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ y \mid K_E \ ,
\end{array}$$

where $M^* \mid \lambda x^*.N$ stands for $M_0 \mid \lambda x_0 \ldots M_n \mid \lambda x_n.N$ with $n \geq 0$.

3. For all terms M and evaluation contexts E, E' we prove by induction on the evaluation context E that the following holds:

$$E[M] \mid K_{E'} \equiv M \mid K_{E'[E]} \ .$$

For instance, we detail the case where the context has the shape $E[@(V^*, [\,], M^*)]$.

$$\begin{array}{ll}
E[@(V^*, [M], M^*)] \mid K_{E'} & \\
\equiv @(V^*, [M], M^*) \mid K_{E'[E]} & \text{(by inductive hypothesis)} \\
\equiv M \mid \lambda x.M^* \mid \lambda x^*.@(\psi(V)^*, x, x^*, K_{E'[E]}) & \\
\equiv M \mid K_{E'[E[@(V^*,[\,],M^*)]]} \ . &
\end{array}$$

4. For all terms M, continuations K, K', and variable $x \notin \mathsf{fv}(M)$ we prove by induction on M and case analysis that the following holds:

$$[K/x](M \mid K') \begin{cases} \to M \mid K' & \text{if } K \text{ abstraction}, M \text{ value}, K' = x \\ \equiv (M \mid [K/x]K') & \text{otherwise.} \end{cases}$$

5. Finally, we prove the assertion by case analysis on the reduction rule. We consider the case for application. Suppose $E[@(\lambda x^+.M, V^+)] \to E[[V^+/x^+]M]$. We have:

$$\begin{array}{l} E[@(\lambda x^+.M, V^+)] \mid K_{[\,]} \\ \equiv @(\lambda x^+.M, V^+) \mid K_E \\ \equiv @(\lambda x^+, k.M \mid k, \psi(V)^+, K_E) \\ \to [K_E/k, \psi(V)^+/x^+](M \mid k) \\ \equiv [K_E/k]([V^+/x^+]M \mid k) \\ \overset{*}{\to} [V^+/x^+]M \mid K_E \\ \equiv E[[V^+/x^+]M] \mid K_{[\,]} \ . \end{array}$$

□

We illustrate this result on the following example.

Example 268 (CPS) *Let $M \equiv @(\lambda x.@(x, @(x, x)), I)$, where $I \equiv \lambda x.x$. Then*

$$\mathcal{C}_{cps}(M) \equiv @(\lambda x, k.@(x, x, \lambda y.@(x, y, k)), I', H) \ ,$$

where: $I' \equiv \lambda x, k.@(k, x)$ and $H \equiv \lambda x.@(halt, x)$. The term M is simulated by $\mathcal{C}_{cps}(M)$ as follows:

$$\begin{array}{cccccccc} M & \to & @(I, @(I, I)) & \to & @(I, I) & \to & I \\ \mathcal{C}_{cps}(M) & \to & @(I', I', \lambda y.@(I', y, H)) & \to^+ & @(I', I', H) & \to^+ & @(halt, I') \ . \end{array}$$

Exercise 269 *Write down a simplified CPS transformation for a monadic call-by-value λ-calculus without let-definitions and tuples. Then apply the CPS transformation to show that it is possible to simulate the call-by-value λ-calculus in the call-by-name λ-calculus (cf. chapter 8).*

Exercise 270 *So called* control operators *are programming instructions that alter the execution flow. For instance, consider the* continue *and* break *commands of exercise 5 and the control $\mathcal{C}$ and abort $\mathcal{A}$ operators of exercise 166. CPS transformations allow to simulate such operators in a purely functional setting.*

1. *In chapter 1, we have interpreted a statement of the* Imp *language as function of type $(State \to State)$. Define an alternative functional interpretation where a command is regarded as a function of type:*

$$State \to (State \to State) \to State \ .$$

   and show that such interpretation can be extended to interpret a command abort *which stops the computation and returns the current state.*

2. *Define a CPS transformation of the call-by-value λ-calculus extended with the control operators $\mathcal{C}$ and $\mathcal{A}$ defined in exercise 166.*

Syntax CPS terms

$$\begin{array}{lll} V & ::= id \mid \lambda id^{+}.M \mid (V^{*}) & \text{(values)} \\ M & ::= @(V, V^{+}) \mid \mathsf{let}\ id = \pi_i(V)\ \mathsf{in}\ M & \text{(CPS terms)} \\ K & ::= id \mid \lambda id.M & \text{(continuations)} \end{array}$$

Reduction rules

$$\begin{array}{lll} @(\lambda x_1 \ldots x_n.M, V_1, \ldots, V_n) & \rightarrow & [V_1/x_1, \ldots, V_n/x_n]M \\ \mathsf{let}\ x = \pi_i(V_1, \ldots, V_n)\ \mathsf{in}\ M & \rightarrow & [V_i/x]M \quad 1 \leq i \leq n \end{array}$$

CPS transformation

$$\begin{array}{lcl} \psi(x) & = & x \\ \psi(\lambda x^{+}.M) & = & \lambda x^{+}, k.(M \mid k) \\ \psi((V_1, \ldots, V_n)) & = & (\psi(V_1), \ldots, \psi(V_n)) \\ \\ V \mid k & = & @(k, \psi(V)) \\ V \mid (\lambda x.M) & = & [\psi(V)/x]M \\ @(M_0, \ldots, M_n) \mid K & = & M_0 \mid \lambda x_0. \ldots (M_n \mid \lambda x_n.@(x_0, \ldots, x_n, K)) \\ \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \mid K & = & M_1 \mid \lambda x.(M_2 \mid K) \\ (M_1, \ldots, M_n) \mid K & = & M_1 \mid \lambda x_1. \ldots (M_n \mid \lambda x_n.(x_1, \ldots, x_n) \mid K\ ) \\ \pi_i(M) \mid K & = & M \mid \lambda x.\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ y \mid K \\ \mathcal{C}_{cps}(M) & = & M \mid \lambda x.@(halt, x), \qquad halt \text{ fresh variable} \end{array}$$

Table 14.2: CPS λ-calculus (λ_{cps}) and CPS transformation

14.2 Value named form

Table 14.3 introduces a *value named* λ-calculus in CPS form: $\lambda_{cps,vn}$. In the ordinary λ-calculus, the application of a λ-abstraction to an argument (which is a value) may duplicate the argument as in: $@(\lambda x.M, V) \rightarrow [V/x]M$. In the value named λ-calculus, all values are named and when we apply the name of a λ-abstraction to the name of a value we create a new copy of the body of the function and replace its formal parameter name with the name of the argument as in:

$$\mathsf{let}\ y = V\ \mathsf{in}\ \mathsf{let}\ f = \lambda x.M\ \mathsf{in}\ @(f, y)\ \rightarrow\ \mathsf{let}\ y = V\ \mathsf{in}\ \mathsf{let}\ f = \lambda x.M\ \mathsf{in}\ [y/x]M\ .$$

We also remark that in the value named λ-calculus the evaluation contexts are a sequence of let definitions associating values to names. Thus, apart for the fact that the values are not necessarily closed, the evaluation contexts are similar to the environments of abstract machines for functional languages (cf. chapter 8).

Table 14.4 defines the compilation into value named form along with a readback transformation. The latter is useful to state the simulation property. Indeed, it is not true that if $M \rightarrow M'$ in λ_{cps} then $\mathcal{C}_{vn}(M) \stackrel{*}{\rightarrow} \mathcal{C}_{vn}(M')$ in $\lambda_{cps,vn}$. For instance, consider $M \equiv (\lambda x.xx)I$ where $I \equiv (\lambda y.y)$. Then $M \rightarrow II$ but $\mathcal{C}_{vn}(M)$ does not reduce to $\mathcal{C}_{vn}(II)$ but rather to a term where the 'sharing' of the duplicated value I is explicitly represented.

Example 271 (value named form) *Consider the term resulting from the CPS transformation in example 268:*

$$N \equiv @(\lambda x, k.@(x, x, \lambda y.@(x, y, k)), I', H))\ ,$$

SYNTAX

$$
\begin{array}{lll}
V & ::= \lambda id^+.M \mid (id^*) & \text{(values)} \\
C & ::= V \mid \pi_i(id) & \text{(let-bindable terms)} \\
M & ::= @(id, id^+) \mid \mathsf{let}\ id = C\ \mathsf{in}\ M & \text{(CPS terms)} \\
E & ::= [\,] \mid \mathsf{let}\ id = V\ \mathsf{in}\ E & \text{(evaluation contexts)}
\end{array}
$$

REDUCTION RULES

$$
\begin{array}{rcll}
E[@(x, z_1, \ldots, z_n)] & \to & E[[z_1/y_1, \ldots, z_n/y_n]M] & \text{if } E(x) = \lambda y_1 \ldots y_n.M \\
E[\mathsf{let}\ z = \pi_i(x)\ \mathsf{in}\ M] & \to & E[[y_i/z]M]] & \text{if } E(x) = (y_1, \ldots, y_n), 1 \leq i \leq n
\end{array}
$$

$$
\text{where: } E(x) = \begin{cases} V & \text{if } E = E'[\mathsf{let}\ x = V\ \mathsf{in}\ [\,]] \\ E'(x) & \text{if } E = E'[\mathsf{let}\ y = V\ \mathsf{in}\ [\,]], x \neq y \\ \text{undefined} & \text{otherwise} \end{cases}
$$

Table 14.3: A value named CPS λ-calculus: $\lambda_{cps,vn}$

where: $I' \equiv \lambda x, k.@(k, x)$ *and* $H \equiv \lambda x.@(halt, x)$*. The corresponding term in value named form is:*

$$
\begin{array}{l}
\mathsf{let}\ z_1 = \lambda x, k.(\mathsf{let}\ z_{11} = \lambda y.@(x, y, k)\ \mathsf{in}\ @(x, x, z_{11}))\ \mathsf{in} \\
\mathsf{let}\ z_2 = I'\ \mathsf{in} \\
\mathsf{let}\ z_3 = H\ \mathsf{in} \\
@(z_1, z_2, z_3)\ .
\end{array}
$$

Proposition 272 (vn simulation) *Let N be a term in CPS value named form. If $\mathcal{R}(N) \equiv M$ and $M \xrightarrow{\alpha} M'$ then there exists N' such that $N \xrightarrow{\alpha} N'$ and $\mathcal{R}(N') \equiv M'$.*

PROOF. First we fix some notation. We associate a substitution σ_E with an evaluation context E of the $\lambda_{cps,vn}$-calculus as follows:

$$
\sigma_{[\,]} = Id \quad \sigma_{\mathsf{let}\ x=V\ \mathsf{in}\ E} = [\mathcal{R}(V)/x] \circ \sigma_E\ .
$$

Then we prove the property by case analysis. We look at the case:

$$
\mathcal{R}(N) \equiv @(\lambda y^+.M, V^+) \to [V^+/y^+]M\ .
$$

Then $N \equiv E[@(x, x^+)]$, $\sigma_E(x) \equiv \lambda y^+.M$, and $\sigma_E(x^+) \equiv V^+$. Moreover, $E \equiv E_1[\mathsf{let}\ x = \lambda y^+.M'\ \mathsf{in}\ E_2]$ and $\sigma_{E_1}(\lambda y^+.M') \equiv \lambda y^+.M$. Therefore, $N \to E[[x^+/y^+]M'] \equiv N'$ and we check that $\mathcal{R}(N') \equiv \sigma_E([x^+/y^+]M') \equiv [V^+/y^+]M$. □

14.3 Closure conversion

The next step is called *closure conversion*. It consists in providing each functional value with an additional parameter that accounts for the names free in the body of the function and in representing functions using closures. Our closure conversion function implements a closure using a pair whose first component is the code of the transformed function and whose second component is a tuple containing the values of the free variables.

TRANSFORMATION IN VALUE NAMED FORM (FROM λ_{cps} TO $\lambda_{cps,vn}$)

$$
\begin{array}{lcl}
\mathcal{C}_{vn}(@(x_0,\ldots,x_n)) & = & @(x_0,\ldots,x_n) \\
\mathcal{C}_{vn}(@(x^*,V,V^*)) & = & \mathcal{E}_{vn}(V,y)[\mathcal{C}_{vn}(@(x^*,y,V^*))] \quad V \neq id, y \text{ fresh} \\
\mathcal{C}_{vn}(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M) & = & \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{vn}(M) \\
\mathcal{C}_{vn}(\mathsf{let}\ x = \pi_i(V)\ \mathsf{in}\ M) & = & \mathcal{E}_{vn}(V,y)[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{vn}(M)] \quad V \neq id, y \text{ fresh} \\
\mathcal{E}_{vn}(\lambda x^+.M, y) & = & \mathsf{let}\ y = \lambda x^+.\mathcal{C}_{vn}(M)\ \mathsf{in}\ [\,] \\
\mathcal{E}_{vn}((x^*), y) & = & \mathsf{let}\ y = (x^*)\ \mathsf{in}\ [\,] \\
\mathcal{E}_{vn}((x^*,V,V^*), y) & = & \mathcal{E}_{vn}(V,z)[\mathcal{E}_{vn}((x^*,z,V^*), y)] \quad V \neq id, z \text{ fresh}
\end{array}
$$

READBACK TRANSFORMATION (FROM $\lambda_{cps,vn}$ TO λ_{cps})

$$
\begin{array}{lcl}
\mathcal{R}(\lambda x^+.M) & = & \lambda x^+.\mathcal{R}(M) \\
\mathcal{R}(x^*) & = & (x^*) \\
\mathcal{R}(@(x,x_1,\ldots,x_n)) & = & @(x,x_1,\ldots,x_n) \\
\mathcal{R}(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M) & = & \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{R}(M) \\
\mathcal{R}(\mathsf{let}\ x = V\ \mathsf{in}\ M) & = & [\mathcal{R}(V)/x]\mathcal{R}(M)
\end{array}
$$

Table 14.4: Transformations in value named CPS form and readback

It will be convenient to write "$\mathsf{let}\ (y_1,\ldots,y_n) = x\ \mathsf{in}\ M$" for "$\mathsf{let}\ y_1 = \pi_1(x)\ \mathsf{in}\ \cdots \mathsf{let}\ y_n = \pi_n(x)\ \mathsf{in}\ M$" and "$\mathsf{let}\ x_1 = C_1 \ldots x_n = C_n\ \mathsf{in}\ M$" for "$\mathsf{let}\ x_1 = C_1\ \mathsf{in} \ldots \mathsf{let}\ x_n = C_n\ \mathsf{in}\ M$". The transformation is described in Table 14.5. The output of the transformation is such that all functional values are closed.

Example 273 (closure conversion) *Let $M \equiv \mathcal{C}_{vn}(\mathcal{C}_{cps}(\lambda x.y))$, namely:*

$$M \equiv \mathsf{let}\ z_1 = \lambda x, k.@(k,y)\ \mathsf{in}\ @(halt, z_1)\ .$$

Then $\mathcal{C}_{cc}(M)$ is the following term:

$$
\begin{array}{l}
\mathsf{let}\ c = \lambda e, x, k.(\mathsf{let}\ (y) = e, (c,e) = k\ \mathsf{in}\ @(c,e,y))\ \mathsf{in} \\
\mathsf{let}\ e = (y), z_1 = (c,e), (c,e) = halt\ \mathsf{in} \\
@(c,e,z_1)\ .
\end{array}
$$

Proposition 274 (CC simulation) *Let M be a CPS term in value named form. If $M \to M'$ then $\mathcal{C}_{cc}(M) \xrightarrow{*} \mathcal{C}_{cc}(M')$.*

PROOF. As a first step we check that the closure conversion function commutes with name substitution:

$$\mathcal{C}_{cc}([x/y]M) \equiv [x/y]\mathcal{C}_{cc}(M)\ .$$

This is a direct induction on the structure of the term M. Then we extend the closure conversion function to contexts as follows:

$$
\begin{array}{ll}
\mathcal{C}_{cc}([\,]) & = [\,] \\
\mathcal{C}_{cc}(\mathsf{let}\ x = (y^*)\ \mathsf{in}\ E) & = \mathsf{let}\ x = (y^*)\ \mathsf{in}\ \mathcal{C}_{cc}(E) \\
\mathcal{C}_{cc}(\mathsf{let}\ x = \lambda x^+.M\ \mathsf{in}\ E) & = \mathsf{let}\ c = \lambda e, x^+.\mathsf{let}\ (z_1,\ldots,z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(M)\ \mathsf{in} \\
 & \quad \mathsf{let}\ e = (z_1,\ldots,z_k), x = (c,e)\ \mathsf{in}\ \mathcal{C}_{cc}(E) \\
 & \text{where: } \mathsf{fv}(\lambda x^+.M) = \{z_1,\ldots,z_k\}\ .
\end{array}
$$

Syntactic restrictions on $\lambda_{cps,vn}$ after closure conversion
All functional values are closed.

Closure Conversion

$$\begin{array}{lcl}
\mathcal{C}_{cc}(@(x, y^+)) & = & \mathsf{let}\ (c, e) = x\ \mathsf{in}\ @(c, e, y^+) \\
\mathcal{C}_{cc}(\mathsf{let}\ x = C\ \mathsf{in}\ M) & = & \begin{array}{l}\mathsf{let}\ c = \lambda e, x^+.\mathsf{let}\ (z_1, \ldots, z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(N)\ \mathsf{in} \\ \mathsf{let}\ e = (z_1, \ldots, z_k)\ \mathsf{in} \\ \mathsf{let}\ x = (c, e)\ \mathsf{in} \\ \mathcal{C}_{cc}(M) \qquad (\text{if } C = \lambda x^+.N, \mathsf{fv}(C) = \{z_1, \ldots, z_k\})\end{array} \\
\mathcal{C}_{cc}(\mathsf{let}\ x = C\ \mathsf{in}\ M) & = & \mathsf{let}\ x = C\ \mathsf{in}\ \mathcal{C}_{cc}(M) \qquad (\text{if } C \text{ not a function})
\end{array}$$

Table 14.5: Closure conversion on value named CPS terms

We note that for any evaluation context E, $\mathcal{C}_{cc}(E)$ is again an evaluation context, and moreover for any term M we have:

$$\mathcal{C}_{cc}(E[M]) \equiv \mathcal{C}_{cc}(E)[\mathcal{C}_{cc}(M)] \ .$$

Finally we prove the simulation property by case analysis of the reduction rule being applied.

- Suppose $M \equiv E[@(x, y^+)] \to E[[y^+/x^+]M]$ where $E(x) = \lambda x^+.M$ and $\mathsf{fv}(\lambda x^+.M) = \{z_1, \ldots, z_k\}$. Then:

  $$\mathcal{C}_{cc}(E[@(x, y^+)]) \equiv \mathcal{C}_{cc}(E)[\mathsf{let}\ (c, e) = x\ \mathsf{in}\ @(c, e, y^+)] \ ,$$

  with $\mathcal{C}_{cc}(E)(x) = (c, e)$, $\mathcal{C}_{cc}(E)(c) = \lambda e, x^+.\mathsf{let}\ (z_1, \ldots, z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(M)$ and $\mathcal{C}_{cc}(E)(e) = (z_1, \ldots, z_k)$. Therefore:

  $$\begin{array}{l}
  \mathcal{C}_{cc}(E)[\mathsf{let}\ (c', e') = x\ \mathsf{in}\ @(c', e', y^+)] \\
  \xrightarrow{*} \mathcal{C}_{cc}(E)[\mathsf{let}\ (z_1, \ldots, z_k) = e\ \mathsf{in}\ [y^+/x^+]\mathcal{C}_{cc}(M)] \\
  \xrightarrow{*} \mathcal{C}_{cc}(E)[[y^+/x^+]\mathcal{C}_{cc}(M)] \\
  \equiv \mathcal{C}_{cc}(E)[\mathcal{C}_{cc}([y^+/x^+]M)] \qquad (\text{by substitution commutation}) \\
  \equiv \mathcal{C}_{cc}(E[[y^+/x^+]M]) \ .
  \end{array}$$

- Suppose $M \equiv E[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M] \to E[[z_i/x]M]$ where $E(y) = (z_1, \ldots, z_k)$, $1 \leq i \leq k$. Then:

  $$\mathcal{C}_{cc}(E[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M]) \equiv \mathcal{C}_{cc}(E)[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{cc}(M)]$$

  with $\mathcal{C}_{cc}(E)(y) = (z_1, \ldots, z_k)$. Therefore:

  $$\begin{array}{l}
  \mathcal{C}_{cc}(E)[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{cc}(M)] \\
  \to \mathcal{C}_{cc}(E)[[z_i/x]\mathcal{C}_{cc}(M)] \\
  \equiv \mathcal{C}_{cc}(E)[\mathcal{C}_{cc}([z_i/x]M)] \qquad (\text{by substitution commutation}) \\
  \equiv \mathcal{C}_{cc}(E[[z_i/x]M]) \ .
  \end{array}$$

□

Exercise 275 *Define a closure conversion transformation that applies directly to the source language rather than to the CPS, value named form.*

SYNTAX FOR λ_h
Syntactic restrictions on $\lambda_{cps,vn}$ after hoisting
All function definitions are at top level.

$$
\begin{array}{lll}
C & ::= (id^*) \mid \pi_i(id) & \text{(restricted let-bindable terms)} \\
T & ::= @(id, id^+) \mid \mathsf{let}\ id = C\ \mathsf{in}\ T & \text{(restricted terms)} \\
P & ::= T \mid \mathsf{let}\ id = \lambda id^+.T\ \mathsf{in}\ P & \text{(programs)}
\end{array}
$$

SPECIFICATION OF THE HOISTING TRANSFORMATION

$$\mathcal{C}_h(M) = N \text{ if } M \rightsquigarrow \cdots \rightsquigarrow N \not\rightsquigarrow, \quad \text{where:}$$

$$D \quad ::= \quad [\,] \mid \mathsf{let}\ id = C\ \mathsf{in}\ D \mid \mathsf{let}\ id = \lambda id^+.D\ \mathsf{in}\ M \qquad \text{(hoisting contexts)}$$

$$
\begin{array}{lll}
(h_1) & D[\mathsf{let}\ x = C\ \mathsf{in}\ \mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M] \rightsquigarrow & \\
& \quad D[\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ \mathsf{let}\ x = C\ \mathsf{in}\ M] & \text{if } x \notin \mathsf{fv}(\lambda z^+.T) \\
(h_2) & D[\mathsf{let}\ x = (\lambda w^+.\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M)\ \mathsf{in}\ N] \rightsquigarrow & \\
& \quad D[\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ \mathsf{let}\ x = \lambda w^+.M\ \mathsf{in}\ N] & \text{if } \{w^+\} \cap \mathsf{fv}(\lambda z^+.T) = \emptyset
\end{array}
$$

Table 14.6: Hoisting transformation

14.4 Hoisting

The last compilation step consists in moving all function definitions at top level. In Table 14.6, we formalize this compilation step as the iteration of a set of program transformations that commute with the reduction relation. Denote with $\lambda z^+.T$ a function that does *not* contain function definitions. The transformations (h_1) and (h_2) consist in hoisting (moving up) the definition of a function $\lambda z^+.T$. In transformation (h_1), we commute the function definition with a tuple or a projection definition. This is always possible on the terms resulting from a closure conversion since in these terms the functions are closed and therefore cannot depend on a tuple or a projection definition above them. In transformation (h_2), we have a function definition, say f_1 which contains a nested function definition, say f_2. In this case we extract f_2 putting it at the same level, and above f_1. Notice that in doing this f_1 is not closed anymore since it may depend on f_2. It can be shown that the rewriting system induced by the rules (h_1) and (h_2) applied to the terms resulting from the closure conversion terminates and is confluent. We omit this rather technical but not difficult development. The proof that the hoisted program simulates the original one also requires some work because to close the diagram we need to collapse repeated definitions, which may occur, as illustrated in the example below. Again, we omit this development.

Example 276 (hoisting transformations and transitions) *Let*

$$M \equiv \mathsf{let}\ x_1 = \lambda y_1.N\ \mathsf{in}\ @(x_1, z)\ ,$$

where $N \equiv \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ T_1$ *and* $y_1 \notin \mathsf{fv}(\lambda y_2.T_2)$. *Then we either reduce and then hoist:*

$$
\begin{array}{ll}
M & \to \mathsf{let}\ x_1 = \lambda y_1.N\ \mathsf{in}\ [z/y_1]N \\
& \equiv \mathsf{let}\ x_1 = \lambda y_1.N\ \mathsf{in}\ \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ [z/y_1]T_1 \\
& \rightsquigarrow \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ \mathsf{let}\ x_1 = \lambda y_1.T_1\ \mathsf{in}\ \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ [z/y_1]T_1 \not\rightsquigarrow
\end{array}
$$

or hoist and then reduce:

$$\begin{array}{rl} M & \rightsquigarrow \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ \mathsf{let}\ x_1 = \lambda y_1.T_1\ \mathsf{in}\ @(x_1, z) \\ & \to \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ \mathsf{let}\ x_1 = \lambda y_1.T_1\ \mathsf{in}\ [z/y_1]T_1 \quad \not\rightsquigarrow \end{array}$$

In the first case, we end up duplicating the definition of x_2.

We conclude by sketching an alternative definition of the hoisting transformation. Let h be a function that takes a term M in CPS, value named form where all functions are closed and produces a pair (T, F), where T is a term without function definitions as specified in Table 14.6, and F is a one-hole context composed of a list of function definitions of the shape:

$$F ::= [\ ] \mid \mathsf{let}\ id = \lambda id^+.T\ \mathsf{in}\ F\ .$$

The definition of the function h is given by induction on M as follows where C is a tuple or a projection as in Table 14.6:

$$\begin{array}{ll} h(@(x, y^+)) & = (@(x, y^+), [\ ]) \\ h(\mathsf{let}\ x = C\ \mathsf{in}\ M) & = \mathsf{let}\ (T, F) = h(M)\ \mathsf{in}\ (\mathsf{let}\ x = C\ \mathsf{in}\ T, F) \\ h(\mathsf{let}\ x = \lambda y^+.M\ \mathsf{in}\ N) & = \mathsf{let}\ (T, F) = h(M), (T', F') = h(N)\ \mathsf{in} \\ & \qquad (T', F[\mathsf{let}\ x = \lambda y^+.T\ \mathsf{in}\ F']) \ . \end{array}$$

The hoisting transformation of the term M then amounts to compute $(T, F) = h(M)$ and then build the term $F[T]$ which is a program according to the syntax defined in Table 14.6.

Exercise 277 *Apply the hoisting transformation to the terms resulting from the closure conversion of exercise 275.*

14.5 Summary and references

We have studied four program transformations: *continuation passing style* makes the evaluation context an additional parameter, *value naming* assigns a name to every value, *closure conversion* explicits the notion of closure, and *hoisting* removes nested function definitions. By putting these transformations in pipeline it is possible to transform a program written in a higher-order language such as ML into a system of functions whose body includes operations to build and project tuples of names and to perform tail-recursive routine calls. Thus we have an implementation technique for higher-order languages which is alternative to the one based on the abstract machines presented in chapter 8. A similar compilation chain has been analyzed in [Chl10] which provides machine certified simulation proofs. A simpler compilation chain arises if we bypass the CPS transformation. In this case, the function calls are not necessarily tail-recursive and the target code can be described as C code with function pointers (cf. exercises 275 and 277). An early analysis of the CPS transformation is in [Plo75]. The idea of value-naming transformation is associated with various formalizations of *sharing*, see, *e.g.*, [Lau93]. Closure conversion arises naturally when trying to define the interpreter of a higher-order language in the language itself, see, *e.g.*, [Rey98]. Hoisting appears to be folklore.

Chapter 15

Typing the program transformations

We describe a typing of the compilation chain described in chapter 14. Specifically, each λ-calculus of the compilation chain is equipped with a type system which enjoys *subject reduction*: if a term has a type then all terms to which it reduces have the same type. Then the compilation functions are extended to types and are shown to be *type preserving*: if a term has a type then its compilation has the corresponding compiled type.

The two main steps in typing the compilation chain concern the CPS and the closure conversion transformations. The typing of the CPS transformation has already been sketched in chapter 14 where it has served as a guideline. A basic idea is to type the continuation/the evaluation context of a term of type A with its negated type $\neg A = (A \to R)$, where R is traditionally taken as the type of 'results'. In typing closure conversion, one relies on *existential types* (cf. chapter 13) to hide the details of the representation of the 'environment' of a function, *i.e.*, the tuple of variables occurring free in its body. Thus, to type the (abstract) assembly code coming from the compilation of propositionally typed programs, we need to go beyond propositional types.

To represent types we shall follow the notation introduced starting from chapter 10. In particular, we denote with *tid* the syntactic category of *type variables* with generic elements $t, s, \ldots$ and with A the syntactic category of *types* with generic elements $A, B, \ldots$ We write $x^* : A^*$ for a possibly empty sequence $x_1 : A_1, \ldots, x_n : A_n$, and $\Gamma, x^* : A^*$ for the context resulting from Γ by adding the sequence $x^* : A^*$. Hence the variables in x^* must not be in the domain of Γ. If A is a type, we write $\mathsf{ftv}(A)$ for the set of *type variables occurring free* in it and, by extension, if Γ is a type context then $\mathsf{ftv}(\Gamma)$ is the union of the sets $\mathsf{ftv}(A)$ where A is a type in the codomain of Γ. A *typing judgment* is typically written as $\Gamma \vdash M : A$ where M is some term. We shall write $\Gamma \vdash M^* : A^*$ for $\Gamma \vdash M_1 : A_1, \ldots, \Gamma \vdash M_n : A_n$. Similar conventions apply if we replace the symbol '$*$' with the symbol '$+$' except that in this case the sequence is assumed not-empty. A type transformation, say $\mathcal{T}$, is *lifted to type contexts* by defining $\mathcal{T}(x_1 : A_1, \ldots x_n : A_n) = x_1 : \mathcal{T}(A_1), \ldots, x_n : \mathcal{T}(A_n)$. Whenever we write:

$$\text{if } \Gamma \vdash^{S_1} M : A \text{ then } \mathcal{T}(\Gamma) \vdash^{S_2} \mathcal{T}(M) : \mathcal{T}(A)$$

what we actually mean is that if the judgment in the hypothesis is *derivable* in a certain 'type system S_1' then the transformed judgment in derivable in the 'type system S_2'. Proofs are standard and are left as exercises.

Syntax types

$$A \quad ::= tid \mid A^+ \to A \mid \times(A^*) \quad \text{(types)}$$

Typing rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash N : B \quad \Gamma \vdash M : A}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : B}$$

$$\frac{\Gamma, x^+ : A^+ \vdash M : B}{\Gamma \vdash \lambda x^+.M : A^+ \to B} \qquad \frac{\Gamma \vdash M : A^+ \to B \quad \Gamma \vdash N^+ : A^+}{\Gamma \vdash @(M, N^+) : B}$$

$$\frac{\Gamma \vdash M^* : A^*}{\Gamma \vdash (M^*) : \times(A^*)} \qquad \frac{\Gamma \vdash M : \times(A_1, \ldots, A_n) \quad 1 \le i \le n}{\Gamma \vdash \pi_i(M) : A_i}$$

Restricted syntax CPS types, R type of results

$$A \quad ::= tid \mid A^+ \to R \mid \times(A^*) \quad \text{(CPS types)}$$

CPS type compilation

$$\begin{array}{ll} \mathcal{C}_{cps}(t) & = t \\ \mathcal{C}_{cps}(\times(A^*)) & = \times(\mathcal{C}_{cps}(A)^*) \\ \mathcal{C}_{cps}(A^+ \to B) & = (\mathcal{C}_{cps}(A))^+, \neg\mathcal{C}_{cps}(B) \to R \\ & \text{where: } \neg A \equiv (A \to R) \end{array}$$

Table 15.1: Type system for λ and λ_{cps}

15.1 Typing the CPS form

Table 15.1 describes the typing rules for the polyadic, call-by-value, λ-calculus defined in Table 14.1. These rules are a slight generalization of those studied in chapter 10 and they are preserved by reduction. The typing rules described in Table 15.1 apply to the CPS λ-calculus too. Table 15.1 describes the restricted syntax of the CPS types and the CPS type transformation. Then the CPS term transformation defined in Table 14.2 preserves typing in the following sense.

Proposition 278 (type CPS) *If* $\Gamma \vdash M : A$ *then* $\mathcal{C}_{cps}(\Gamma), halt : \neg\mathcal{C}_{cps}(A) \vdash \mathcal{C}_{cps}(M) : R$.

15.2 Typing value-named closures

Table 15.2 describes the typing rules for the value named calculi with functional, product, and existential types. Notice that for the sake of brevity, we shall omit the type of a term since this type is always the type of results R and write $\Gamma \vdash^{vn} M$ rather than $\Gamma \vdash^{vn} M : R$. The first five typing rules are just a specialization of the corresponding rules in Table 15.1, while the last two rules allow for the introduction and elimination of *existential types*. The need for existential types will be motivated next. For the time being, let us notice that in the proposed

formalization we rely on the tuple constructor to introduce an existential type and the first projection to eliminate it. This has the advantage of leaving unchanged the syntax and the reduction rules of the value named λ-calculus. An alternative presentation (cf. chapter 13) consists in introducing specific operators to introduce and eliminate existential types denoted with pack and unpack, respectively. Then one can read (x) as $\mathsf{pack}(x)$ and $\pi_1(x)$ as $\mathsf{unpack}(x)$ when x has an existential type. Notice that the rewriting rule which allows to *unpack* a *packed* value is just a special case of the rule for projection. As in the previous system, typing is preserved by reduction.

Proposition 279 (subject reduction, value named) *If M is a term of the $\lambda_{cps,vn}$-calculus, $\Gamma \vdash^{vn} M$ and $M \to N$ then $\Gamma \vdash^{vn} N$.*

Turning to the transformation from CPS to *value named* CPS form specified in Table 14.4 we notice that it affects the terms but not the types. Therefore we have the following property.

Proposition 280 (type value named) *If M is a term of the λ_{cps}-calculus and $\Gamma \vdash M : R$ then $\Gamma \vdash^{vn} \mathcal{C}_{vn}(M)$.*

Next we discuss the typing of closure conversion via existential types (Table 15.2). We recall that in closure conversion a function, say $\lambda x.M$ with free variables $z_1, \ldots, z_n$, becomes a pair (here we ignore the details of the CPS, value named form):

$$(\lambda e, x.\mathsf{let}\ (z_1, \ldots, z_n) = e\ \mathsf{in}\ \mathcal{C}(M), (z_1, \ldots, z_n)) \tag{15.1}$$

whose first component is the function itself, which is *closed* by taking the environment e as an additional argument, and the second component is a tuple containg the values of the free variables. Now consider the functions *identity* and *successor* on the natural numbers coded as follows:

$$\lambda x.x\ , \qquad \mathsf{let}\ y = 1\ \mathsf{in}\ \lambda x.x + y\ , \tag{15.2}$$

with a type, say, $\mathbf{N} \to \mathbf{N}$. After closure conversion, we obtain the following pairs with the respective *different* types:

$$\begin{array}{ll} (\lambda e, x.\mathsf{let}\ () = e\ \mathsf{in}\ x, ()) & : ((1 \times \mathbf{N}) \to \mathbf{N}) \times 1 \\ (\lambda e, x.\mathsf{let}\ (y) = e\ \mathsf{in}\ x + y, (y)) & : ((\mathbf{N} \times \mathbf{N}) \to \mathbf{N}) \times \mathbf{N}\ . \end{array}$$

Then take a function such as $F : (\mathbf{N} \to \mathbf{N}) \to \mathbf{N}$ which can operate both on the identity and the successor function. This is no longer possible after closure conversion if we stick to the typing outlined above. We can address this issue by *abstracting* the types of the identity and successor functions after closure conversion into the following *existential* type:

$$\exists t.((t \times \mathbf{N}) \to \mathbf{N}) \times t\ . \tag{15.3}$$

To summarize, an environment is a tuple whose size depends on the number of variables occurring free in the function. This information should be abstracted in the type; otherwise, we cannot type functions operating on arguments with environments of different size.

In order to respect our conventions on the introduction and elimination of existential types, the closure conversion transformation is slightly modified in the way described in Table 15.3. This modified closure conversion still enjoys the simulation properties stated in proposition 274 and moreover it preserves typing as follows.

Syntax types

$$A \quad ::= tid \mid (A^+ \to R) \mid \times(A^*) \mid \exists tid.A$$

Typing rules

$$\frac{\Gamma, x^+ : A^+ \vdash^{vn} M}{\Gamma \vdash^{vn} \lambda x^+.M : A^+ \to R} \qquad \frac{x : A^+ \to R, y^+ : A^+ \in \Gamma}{\Gamma \vdash^{vn} @(x, y^+)}$$

$$\frac{x^* : A^* \in \Gamma}{\Gamma \vdash^{vn} (x^*) : \times(A^*)} \qquad \frac{y : \times(A_1, \ldots, A_n) \in \Gamma \quad 1 \leq i \leq n \quad \Gamma, x : A_i \vdash^{vn} M}{\Gamma \vdash^{vn} \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M}$$

$$\frac{\Gamma \vdash^{vn} V : A \quad \Gamma, x : A \vdash^{vn} M}{\Gamma \vdash^{vn} \mathsf{let}\ x = V\ \mathsf{in}\ M}$$

$$\frac{x : [B/t]A \in \Gamma}{\Gamma \vdash^{vn} (x) : \exists t.A} \qquad \frac{y : \exists t.A \in \Gamma \quad \Gamma, x : A \vdash^{vn} M \quad t \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash^{vn} \mathsf{let}\ x = \pi_1(y)\ \mathsf{in}\ M}$$

Closure conversion type compilation

$$\begin{array}{ll} \mathcal{C}_{cc}(t) & = t \\ \mathcal{C}_{cc}(\times(A^*)) & = \times(\mathcal{C}_{cc}(A)^*) \\ \mathcal{C}_{cc}(A^+ \to R) & = \exists t.\, \times((t, \mathcal{C}_{cc}(A)^+ \to R),\ t) \\ \mathcal{C}_{cc}(\exists t.A) & = \exists t.\mathcal{C}_{cc}(A) \end{array}$$

Table 15.2: Type system for the value named calculi and closure conversion

$$\begin{array}{lcll} \mathcal{C}_{cc}(@(x, y^+)) & = & \mathsf{let}\ x = \pi_1(x)\ \mathsf{in} & (\leftarrow \text{existential elimination}) \\ & & \mathsf{let}\ (c, e) = x\ \mathsf{in}\ @(c, e, y^+) & \\ \mathcal{C}_{cc}(\mathsf{let}\ x = C\ \mathsf{in}\ M) & = & \mathsf{let}\ c = \lambda e, x^+.\mathsf{let}\ (z_1, \ldots, z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(N)\ \mathsf{in} & \\ & & \mathsf{let}\ e = (z_1, \ldots, z_k)\ \mathsf{in} & \\ & & \mathsf{let}\ x = (c, e)\ \mathsf{in} & \\ & & \mathsf{let}\ x = (x)\ \mathsf{in} & (\leftarrow \text{existential introduction}) \\ & & \mathcal{C}_{cc}(M) & (\text{if } C = \lambda x^+.N, \mathsf{fv}(C) = \{z_1, \ldots, z_k\}) \end{array}$$

Table 15.3: Modified closure conversion

Proposition 281 (type closure conversion) *If M is a term in $\lambda_{cps,vn}$ and $\Gamma \vdash^{vn} M$ then $\mathcal{C}_{cc}(\Gamma) \vdash^{vn} \mathcal{C}_{cc}(M)$.*

The last step in the compilation chain is the hoisting transformation. Similarly to the transformation in value named form, the hoisting transformation affects the terms but *not* the types.

Proposition 282 (type hoisting) *If M is a term in $\lambda_{cps,vn}$, $\Gamma \vdash^{vn} M$, and $M \rightsquigarrow N$ then $\Gamma \vdash^{vn} N$.*

15.3 Typing the compiled code

We can now extend the compilation function to types by defining:

$$\mathcal{C}(A) = \mathcal{C}_{cc}(\mathcal{C}_{cps}(A))$$

and by composing the previous results we derive the following type preservation property of the compilation function.

Proposition 283 (type preserving compilation) *If M is a term of the λ-calculus and $\Gamma \vdash M : A$ then:*

$$\mathcal{C}(\Gamma), halt : \exists t.\times(t, \mathcal{C}(A) \to R, t) \vdash^{vn} \mathcal{C}(M) \ .$$

Remark 284 *The 'halt' variable introduced by the CPS transformation can occur only in a subterm of the shape $@(halt, x)$ in the intermediate code prior to closure conversion. Then in the closure conversion transformation, we can set $\mathcal{C}_{cc}(@(halt, x)) = @(halt, x)$, and give to '$halt$' a functional rather than an existential type. With this proviso, theorem 283 above can be restated as follows:*

If M is a term of the λ-calculus and $\Gamma \vdash M : A$ then $\mathcal{C}(\Gamma), halt : \neg\mathcal{C}(A) \vdash^{vn} \mathcal{C}(M)$.

Example 285 (typing the compiled code) *We consider again the compilation of the term $\lambda x.y$ (cf. example 273) which can be typed,* e.g.*, as follows:*

$$y : t_1 \vdash \lambda x.y : (t_2 \to t_1) \ .$$

Its CPS transformation is then typed as:

$$y : t_1, halt : \neg\mathcal{C}_{cps}(t_2 \to t_1) \vdash @(halt, \lambda x, k.@(k, y)) : R \ .$$

The value named transformation does not affect the types:

$$y : t_1, halt : \neg\mathcal{C}_{cps}(t_2 \to t_1) \vdash^{vn} \mathsf{let}\ z_1 = \lambda x, k.@(k, y)\ \mathsf{in}\ @(halt, z_1) \ .$$

After closure conversion we obtain the following term M:

$$\begin{array}{l}\mathsf{let}\ c = \lambda e, x, k.\mathsf{let}\ y = \pi_1(e), k = \pi_1(k), c = \pi_1(k), e = \pi_2(k)\ \mathsf{in}\ @(c, e, y)\ \mathsf{in}\\ \mathsf{let}\ e = (y), z_1 = (c, e), z_1 = (z_1), halt = \pi_1(halt), c = \pi_1(halt), e = \pi_2(halt)\ \mathsf{in}\\ @(c, e, z_1)\ ,\end{array}$$

which is typed as follows:

$$y : t_1, halt : \exists t.\times(t, \mathcal{C}(t_2 \to t_1) \to R, t) \vdash^{vn} M \ .$$

In this case no further hoisting transformation applies. If we adopt the optimized compilation strategy sketched in remark 284 then after closure conversion we obtain the following term M':

$$\begin{array}{l}\mathsf{let}\ c = \lambda e, x, k.\mathsf{let}\ y = \pi_1(e), k = \pi_1(k), c = \pi_1(k), e = \pi_2(k)\ \mathsf{in}\ @(c, e, y)\ \mathsf{in}\\ \mathsf{let}\ e = (y), z_1 = (c, e), z_1 = (z_1),\ \mathsf{in}\\ @(halt, z_1)\end{array}$$

which is typed as follows:

$$y : t_1, halt : \mathcal{C}(t_2 \to t_1) \to R \vdash^{vn} M' \ .$$

15.4 Summary and references

We have typed the compilation chain presented in chapter 14 which goes from a higher-order language to an abstract assembly code. The typing of the CPS transformation builds on the double negation translations from classical to intuitionistic logic (see, *e.g.*, [TvD88]). The typing of closure conversion relies on existential types to hide the details of the representation [MMH96]. The paper [MWCG99] shows that the typing can be extend to the impredicative polymorphic types of system F (cf. chapter 13).

Chapter 16

Records, variants, and subtyping

Records and variants are common data types found in many programming languages which allow to aggregate heterogeneous data. Record (variant) types provide a user-friendly alternative to product (sum) types where components can be manipulated by labels rather than by projections (injections).

In this chapter, we start by discussing an extension of the call-by-value, type-free, λ-calculus with records and a possible encoding of records. We then move on to consider a *typed* version of the language. In order to gain in flexibility, we introduce a *subtyping rule* for records and study the properties of the derived type system. We conclude by briefly discussing how the approach with subtyping can be extended to variant types.

16.1 Records

A *record* is a notation to represent a function with a finite domain over a set of labels. Formally, we defined labels as follows.

Definition 286 (labels) *We denote with L a countable and totally ordered set of* labels *with generic elements $\ell, \ell', \ldots$*

We rely on the notation:

$$\{\ell_1 = V_1, \ldots, \ell_n = V_n\} \ , \tag{16.1}$$

to denote the function that associates with the label ℓ_i the value V_i and which is undefined otherwise. Whenever we write a record we assume that the labels are all distinct: $\ell_i \neq \ell_j$ if $i \neq j$. Given a record R, we write $R.\ell$ for the selection of the value of R on the label ℓ. If ℓ is not in the domain of definition of the record then we are in an erroneous situation and the computation is stuck or alternatively an error message is produced.

Table 16.1 describes an extension of the type-free, call-by-value, λ-calculus with records. In order to have a deterministic evaluation strategy, we assume that records are always written with labels in *growing order* and that the evaluation follows this order.

We pause to notice that in principle records could be represented in the pure λ-calculus. For instance, we could associate with each label a natural number and then associate with it a Church numeral. Suppose: (i) $\underline{\ell}$ denotes the Church numeral that corresponds to the label ℓ, (ii) E is a λ-term that decides the equality of two Church numerals (cf. exercise 248), (iii) C is the λ-term that represents the conditional, (iv) F is a special λ-term to represent

failure, and (v) we write $\mathsf{let}\ x_1 = M_1, \ldots, x_n = M_n\ \mathsf{in}\ N$ for $(\lambda x_1, \ldots, x_n.N)M_1 \cdots M_n$. Then we could *compile* the call-by-value λ-calculus with records into the call-by-value λ-calculus as follows (simple cases omitted):

$$\begin{array}{lll} \mathcal{C}(\{\ell_1 = M_1, \ldots, \ell_n = M_n\}) & = \mathsf{let}\ x_1 = \mathcal{C}(M_1), \ldots, x_n = \mathcal{C}(M_n)\ \mathsf{in} \\ & \quad \lambda l.C(E\ l\ \underline{\ell_1})\ x_1(\cdots \quad (C(E\ l\ \underline{\ell_n})\ x_n\ F)\cdots) \\ \\ \mathcal{C}(M.\ell) & = \mathsf{let}\ x = \mathcal{C}(M)\ \mathsf{in}\ (x\ \underline{\ell})\ . \end{array}$$

Thus a record is compiled into a function taking a label as input and then performing a sequence of conditionals. Selecting a record's label just amounts to *apply* the compilation of the record to the encoding of the label.

16.2 Subtyping

Next we turn to the issue of typing the extension of the λ-calculus with records. We take as starting point the type system in Table 10.1 that assigns simple types to λ-terms whose λ-abstractions are decorated with types (Church style). We extend the syntax of types by introducing a notion of *record type* which is a notation for representing a finite function from labels to types:

$$A ::= \mathit{tid} \mid (A \to A) \mid \{\ell : A, \ldots, \ell : A\} \quad \text{(types)}.$$

And then we add two *typing rules* to introduce and eliminate record types.

$$\frac{\Gamma \vdash M_i : A_i \quad i = 1, \ldots, n}{\Gamma \vdash \{\ell_1 = M_1, \ldots, \ell_n = M_n\} : \{\ell_1 : A_1, \ldots, \ell_n : A_n\}} \qquad \frac{\Gamma \vdash M : \{\ell_1 : A_1, \ldots, \ell_n : A_n\}}{\Gamma \vdash M.\ell_i : A_i}\ .$$

The extended type system still has the property that in a given type context each λ-term has at most one type. However, consider the record types:

$$A = \{\ell_1 : A_1, \ell_2 : A_2\}\ , \qquad B = \{\ell_1 : A_1\}\ .$$

If we have a value of type A then we could use it in any context that waits for a value of type B. This simple remark pleads for the introduction of a *subtyping* relation $A \leq B$. Table 16.2 describes a possible definition of the subtyping relation for records and functional types.

Syntax

$$\begin{array}{ll} M ::= \mathit{id} \mid \lambda \mathit{id}.M \mid MM \mid \{\ell = M, \ldots, \ell = M\} \mid M.\ell & (\lambda\text{-terms}) \\ V ::= \lambda \mathit{id}.M \mid \{\ell = V, \ldots, \ell = V\} & (\text{values}) \end{array}$$

Call-by-value evaluation contexts and reduction rules

$$E ::= [\,] \mid EM \mid VE \mid \{(\ell = V)^*, \ell = E, (\ell = V)^*\} \mid E.\ell$$

$$\begin{array}{lcl} (\lambda x.M)V & \to & [V/x]M \\ \{\ldots, \ell = V, \ldots\}.\ell & \to & V \end{array}$$

Table 16.1: Type-free, call-by-value, λ-calculus with records

$$\frac{}{t \leq t} \qquad \frac{A' \leq A \qquad B \leq B'}{A \to B \leq A' \to B'}$$

$$\frac{\{\ell'_1, \ldots, \ell'_m\} \subseteq \{\ell_1, \ldots, \ell_n\} \quad A_{\ell'_i} \leq B_{\ell'_i} \quad i = 1, \ldots, m}{\{\ell_1 : A_{\ell_1}, \ldots, \ell_n : A_{\ell_n}\} \leq \{\ell'_1 : B_{\ell'_1}, \ldots, \ell'_m : B_{\ell'_m}\}}$$

Table 16.2: Subtyping rules for records

We write $\vdash A \leq B$ if the assertion $A \leq B$ can be derived according to the rules in Table 16.2. There are a couple of intriguing points in the definition of the rules. First, notice that the rule for functional types is *anti-monotonic* in the first argument. To get an intuition, suppose we can use natural numbers where integers are expected: $\mathbf{N} \leq \mathbf{Z}$. Then a function f of type $\mathbf{Z} \to \mathbf{N}$ can also be used whenever a function of type $\mathbf{N} \to \mathbf{Z}$ is expected. Indeed, f will be able to handle any natural number since it is built to work on integers and it will return an integer since it is expected to return a natural number. On the other hand, if g has type $\mathbf{N} \to \mathbf{N}$ then it cannot be used where a function of type $\mathbf{Z} \to \mathbf{Z}$ is expected as g may fail to handle a negative integer. Second, the rules are completely *syntax directed*: for each pair of types there is at most one rule that applies and in this case there is only one way to apply it. We have the following properties.

Proposition 287 *The subtyping relation defined in Table 16.2 enjoys the following properties:*

1. *It is reflexive and transitive.*
2. *If $\vdash A \leq B$ then there is a coercion λ-term $C_{A,B}$ such that $\vdash C_{A,B} : A \to B$.*

Proof. (1) Reflexivity follows by induction on the structure of the type A. For transitivity, we build a proof of $B \leq C$ by induction on the height of the proofs of $A \leq B$ and $B \leq C$ and case analysis on the last rules applied. For instance, suppose we have:

$$\frac{B' \leq A' \qquad A'' \leq B''}{A' \to A'' \leq B' \to B''} \qquad \frac{C' \leq B' \qquad B'' \leq C''}{B' \to B'' \leq C' \to C''} \ .$$

Then by inductive hypothesis we can prove $C' \leq A'$ and $A'' \leq C''$ and we conclude as follows:

$$\frac{C' \leq A' \quad A'' \leq C''}{A' \to A'' \leq C' \to C''} \ .$$

(2) We proceed by induction on the proof of $A \leq B$. For the basic case, take the identity. For the functional case, take:

$$C_{A' \to A'', B' \to B''} = \lambda f : A' \to A''. \lambda x : B'. c_{A'', B''}(f(c_{B', A'} x)) \ .$$

For the record case, assume:

$$A = \{\ell_1 : A_{\ell_1}, \ldots, \ell_n : A_{\ell_n}\} \ , \qquad B = \{\ell'_1 : B_{\ell'_1}, \ldots, \ell'_m : B_{\ell'_m}\} \ ,$$

and the conditions specified in Table 16.2 are satisfied. Then define:

$$C_{A,B} = \lambda x : A.\{\ell'_1 = C_{A_{\ell'_1},B_{\ell'_1}}(x.\ell'_1), \ldots, \ell'_m = C_{A_{\ell'_m},B_{\ell'_m}}(x.\ell'_m)\} \ .$$

□

Proposition 287 above guarantees that the subtyping relation defined by the rules in Table 16.2 is indeed a pre-order and moreover that whenever A is a subtype of B we can build a well-typed λ-term of type $A \to B$ that gives us a canonical way to transform a λ-term of type A into a λ-term of type B.

Next we discuss the integration of the subtying rule to the type system for the λ-calculus with records. One possibility would be to add the following typing rule while leaving all the other typing rules unchanged:

$$\frac{\Gamma \vdash M : A \qquad \vdash A \leq B}{\Gamma \vdash M : B} \ . \tag{16.2}$$

The problem with this approach is that typing is no more directed by the syntax of the λ-term (we had a similar problem with the rules $(\forall_I)$ and $(\forall_E)$ in Table 12.1). However, one can remark that the only situation where types need to be matched arises in the application of a λ-term to another one. Hence, we integrate subtyping to the rule for application as follows:

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A' \quad \vdash A' \leq A}{\Gamma \vdash MN : B} \ . \tag{16.3}$$

Notice that the resulting system maintains the property that each λ-term has at most one type. Let us write $\Gamma \vdash_{\leq} M : A$ for a judgment derivable in the resulting type system and let us write $\Gamma \vdash^s_{\leq} M : A$ for a judgment derivable in the ordinary type system extended with the subtyping rule (16.2). Then we have the following proposition.

Proposition 288 *The following properties hold:*

1. *If* $\Gamma \vdash_{\leq} M : A$ *then* $\Gamma \vdash^s_{\leq} M : A$.
2. *If* $\Gamma \vdash^s_{\leq} M : A$ *then there is a type* B *such that* $\Gamma \vdash_{\leq} M : B$ *and* $\vdash B \leq A$.

Proof. (1) Rule (16.3) can be derived from the rule (16.2) and the ordinary rule to type application.

(2) We proceed by induction on the derivation of $\Gamma \vdash^s_{\leq} M : A$. We consider some significant cases.

- Suppose we derive $\Gamma \vdash^s_{\leq} M : A$ from $\Gamma \vdash^s_{\leq} M : A'$ and $\vdash A' \leq A$. Then by inductive hypothesis, we can derive $\Gamma \vdash_{\leq} M : B$ and $\vdash B \leq A'$. And by transitivity of subtyping (proposition 287), we conclude $\vdash B \leq A$.
- Suppose we derive $\Gamma \vdash^s_{\leq} MN : A$ from $\Gamma \vdash^s_{\leq} M : A' \to A$ and $\Gamma \vdash^s_{\leq} N : A'$. Then by inductive hypothesis, we can derive $\Gamma \vdash_{\leq} M : B_1$, $\vdash B_1 \leq A' \to A$, $\Gamma \vdash_{\leq} N : B_2$, and $\vdash B_2 \leq A'$. Then we must have $B_1 \equiv B'_1 \to B''_1$, $\vdash A' \leq B'_1$, and $B''_1 \leq A$. By transitivity, $\vdash B_2 \leq B'_1$. Therefore we can derive: $\Gamma \vdash_{\leq} MN : B''_1$ and $B''_1 \leq A$.

- Suppose we derive $\Gamma \vdash^s_{\leq} \lambda x : A.M : A \to A'$ from $\Gamma, x : A \vdash^s_{\leq} M : A'$. Then by inductive hypothesis, we can derive $\Gamma, x : A \vdash^s_{\leq} M : B$ and $\vdash B \leq A'$. Hence $\Gamma \vdash^s_{\leq} \lambda x : A.M : A \to B$ and $\vdash A \to B \leq A \to A'$. □

Thus the syntax-directed system assigns to a typable λ-term the least type among the types assignable to the λ-term in the more liberal system where the subtyping rule can be freely applied. The statement of the subject reduction property in the syntax-directed system requires some care because the type of a λ-term may grow after reduction. For instance, consider the reduction:

$$M \equiv (\lambda x : \{\ell_1 : A_1\}.x)\{\ell_1 = V_1, \ell_2 = V_2\} \to \{\ell_1 = V_1, \ell_2 = V_2\} \equiv N \ .$$

Then we may have $\emptyset \vdash_{\leq} M : \{\ell_1 : A_1\}$ and $\emptyset \vdash_{\leq} N : \{\ell_1 : A_1, \ell_2 : A_2\}$.

Proposition 289 *If $\Gamma \vdash_{\leq} M : A$ and $M \to N$ then for some type B, $\Gamma \vdash_{\leq} N : B$ and $\vdash B \leq A$.*

Proof. As a preliminary remark, we show that if $\Gamma, x : A \vdash_{\leq} M : B$, $\Gamma \vdash_{\leq} N : A'$, and $\vdash A' \leq A$ then $\Gamma \vdash_{\leq} [N/x]M : B'$ and $\vdash B' \leq B$. The preliminary remark is applied in the analysis of a β-reduction. Suppose $\Gamma \vdash_{\leq} (\lambda x : A.M)N : B$. Then we must have $\Gamma, x : A \vdash_{\leq} M : B$, $\Gamma \vdash_{\leq} N : A'$, and $\vdash A' \leq A$. Thus $\Gamma \vdash_{\leq} [N/x]M : B'$ and $\vdash B' \leq B$. □

The extension of the system with subtyping still guarantees that a well-typed program cannot go wrong. In particular, it is not possible to select a label ℓ in a record where the label is not defined.

Proposition 290 *Suppose $\emptyset \vdash_{\leq} M : A$ then either M is a value or $M \to N$.*

Proof. By induction on the structure of M. Suppose M is not a value. It cannot be a variable because the type context is empty.

If $M \equiv M_1 M_2$ then we must have $\emptyset \vdash_{\leq} M_1 : A \to B$, $\emptyset \vdash_{\leq} M_2 : A'$ and $\vdash A' \leq A$. By inductive hypothesis, if M_1 or M_2 are not values then they reduce and so $M_1 M_2$ reduces too. On the other hand, if M_1 and M_2 are both values then M_1 must be a λ-abstraction and therefore M reduces.

If $M \equiv M'.\ell$ then we must have $\emptyset \vdash_{\leq} M' : \{\ldots \ell : A \ldots\}$. By inductive hypothesis, if M' is not a value then it reduces and so M reduces too. On the other hand, if M' is a value then it must be a record defined on the label ℓ and therefore M reduces. □

16.3 Variants

Variants are data structures dual to records just as sums are dual to products. As such, the subtyping theory developed for records can be easily adapted to variants. As for records, we start with a set of labels (cf. definition 286). Then a variant is a notation to represent an element of a finite disjoint sum indexed over labels. A variant *value* is a λ-term of the shape $[\ell = V]$ and the *reduction rule* for variants is:

$$\mathsf{case}_{\ell_1,\ldots,\ell_n}[\ell = V]V_1 \ldots V_n \to V_i V \qquad \text{if } \ell = \ell_i \ . \tag{16.4}$$

To have a deterministic rule, we assume the labels $\ell_1, \ldots, \ell_n$ are distinct.

We denote a *variant type* with the notation:

$$[\ell_1 : A_1, \ldots, \ell_n : A_n] \ . \tag{16.5}$$

The typing rules for introducing and eliminating variants are as follows:

$$\frac{\Gamma \vdash M : A_i \quad i \in \{1, \ldots, n\}}{\Gamma \vdash [\ell_i = M] : [\ell_1 : A_1, \ldots, \ell_n : A_n]}$$

$$\frac{\Gamma \vdash M : [\ell_1 : A_1, \ldots, \ell_n : A_n] \quad \Gamma \vdash M_i : A_i \to C \quad i = 1, \ldots, n}{\Gamma \vdash \mathsf{case}_{\ell_1,\ldots,\ell_n} M M_1 \ldots M_n : C} \ .$$

Notice that we explicitly label the case constructor The *subtyping rule* for variants is similar to the one for records but *upside down*:

$$\frac{\{\ell_1, \ldots, \ell_n\} \subseteq \{\ell'_1, \ldots, \ell'_m\} \quad A_{\ell_i} \leq B_{\ell_i} \quad i = 1, \ldots, n}{[\ell_1 : A_{\ell_1}, \ldots, \ell_n : A_{\ell_n}] \leq [\ell'_1 : B_{\ell'_1}, \ldots, \ell'_m : B_{\ell'_m}]} \ . \tag{16.6}$$

We leave it as an exercise to adapt the propositions 287, 288, 289, and 290 to variants and to prove them.

16.4 Summary and references

Records and variants are a user-friendly version of products and disjoint unions. The introduction of record and variant types suggests a notion of *subtyping* with the following intuition: if A is a subtype of B then we should be able to use a value of type A whenever a value of type B is expected. We have shown that the subtyping rule can be added to the type system in such a way that typing is still syntax-directed and a typable λ-term is assigned the least type with respect to the sub-typing pre-order. The paper [Car88] is an early reference on the formalization of subtyping and its semantics. Elaborations can be found, *e.g.*, in [Mit88, AC93]. The book [Pie02] contains several chapters dedicated to subtyping.

Chapter 17

References

In chapter 1, we have considered an elementary *imperative* programming language whose programs can be understood as sequences of commands acting on a global *state*. In that context, the state was regarded as an abstraction of the notion of computer memory and was simply modeled as a function from identifiers to (basic) values.

In this chapter, we reconsider the notion of *imperative* programming. We replace the *state* mentioned above with a notion of *heap*. A heap can also be regarded as an abstraction of the notion of computer memory and it is modeled as a function from *references* to (possibly complex) values. In turn, references can be regarded as an abstraction of the notion of memory address. References are first-class values. The value associated with a reference can be read and modified. Moreover, during the computation, it is possible to generate new references and associate values with them.

We formalize a higher-order functional language with references which is inspired by the languages of the *ML*-family. Technically, we introduce the reduction rules of a type-free, call-by-value, λ-calculus with *references* extended with operations to generate, read, and write references. We then discuss a possible compilation of the λ-calculus with 'side effects' on the heap into an ordinary λ-calculus. The compilation turns each expression into a function that takes a heap as an argument and returns a pair composed of a new heap and a value. We conclude the chapter by introducing a propositional type system for the λ-calculus with references which enjoys a subject-reduction property and by discussing some typing anomalies which arise with references.

17.1 References and heaps

References can be regarded as an abstraction of memory addresses and a heap as an abstraction of a computer memory.

Definition 291 (references) *We denote with R a countable set of references with generic elements $r, r', \ldots$ We assume R is equipped with a function $\mathcal{N} : \mathcal{P}_{fin}(R) \to R$ such that for all X, finite subset of R, we have $\mathcal{N}(X) \notin X$.*

Definition 292 (heap) *A heap h is a function over the set of references R whose domain of definition is finite.*

SYNTAX

$$\begin{array}{ll} M ::= id \mid \lambda id.M \mid MM \mid r \mid \mathsf{ref}\ M \mid !M \mid M := M \mid * & (\lambda\text{-terms}) \\ V ::= \lambda id.M \mid r \mid * & (\text{values}) \end{array}$$

CALL BY VALUE EVALUATION CONTEXTS AND REDUCTION RULES

$$E ::= [\,] \mid EM \mid VE \mid \mathsf{ref}E \mid !E \mid E := M \mid V := E$$

$$\begin{array}{lcll} (E[(\lambda x.M)V], h) & \to & ([V/x]M, h) & \\ (E[\mathsf{ref}\ V], h) & \to & (E[r], h[V/r]) & \text{if } r = \mathcal{N}(dom(h)) \\ (E[!r], h) & \to & (E[h(r)], h) & \text{if } r \in dom(h) \\ (E[r := V], h) & \to & (E[*], h[V/r]) & \text{if } r \in dom(h) \end{array}$$

Table 17.1: A call-by-value λ-calculus with references

We manipulate heaps using the standard notation for functions. Thus if h is a heap, then $dom(h)$ is its domain of definition, $h(r)$ its image at r, and $h[v/r]$ is an 'updated' heap defined as follows:

$$h[v/r](r') = \begin{cases} h(r) & \text{if } r = r' \\ v & \text{otherwise.} \end{cases}$$

Notice that we make no assumption on the nature of the values in a heap and that in particular a value can be a reference. In Table 17.1, we introduce an extension of the type-free call-by-value λ-calculus with a notation closely related to the one found in the programming languages of the *ML* family: $\mathsf{ref}\ M$ allocates a new reference which is associated with the value of M, $!M$ reads the value associated with the reference resulting from the evaluation of M, and $M := N$ writes in the reference resulting from the evaluation of M the value of N. We also introduce a constant $*$ which is used as the value resulting from the evaluation of an assignment $M := N$. Ordinary programs are closed λ-terms where references do not occur. However this property is not preserved by reduction and for this reason we include references among the λ-terms and the values of the language. We rely on the following standard abbreviations:

$$\begin{array}{ll} \mathsf{let}\ x = M\ \mathsf{in}\ N & = (\lambda x.M)N \\ M; N & = (\lambda x.N)M \quad x \notin \mathsf{fv}(N)\ . \end{array}$$

References and heaps can be simulated in the pure λ-calculus. As for records' labels (cf. chapter 16), we can use Church numerals to represent references. The operator $\mathcal{N}$ can be implemented by computing the successor of the largest numeral in the set. A heap can then be represented as a list of pairs composed of a reference and a value. Computing the domain of a heap amounts to iterate the first projection on the list. Reading a reference r in the heap means scanning the list till a pair (r, V) is found and updating a reference means building a new heap where the value corresponding to the reference is suitably modified. Let us assume λ-terms *New* to create a new reference, *Ext* to extend a heap with a new pair, *Read* to read a reference, and *Write* to write a value in the heap. In Table 17.2, we describe the compilation of the the λ-calculus with references into a λ-calculus with *pairing*. We also use the following abbreviation for projections:

$$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ N \equiv \mathsf{let}\ z = M,\ x = \pi_1 z,\ y = \pi_2\ z\ \mathsf{in}\ N\ .$$

$$
\begin{array}{ll}
\mathcal{C}(x) & = \lambda h.(h, x) \\
\mathcal{C}(\lambda x.M) & = \lambda h.(h, \lambda x.\mathcal{C}(M)) \\
\mathcal{C}(r) & = \lambda h.(h, \underline{r}) \\
\mathcal{C}(*) & = \lambda h.(h, *) \\
\mathcal{C}(MN) & = \lambda h.\mathsf{let}\ (h', x) = \mathcal{C}(M)h,\ (h'', y) = \mathcal{C}(N)h'\ \mathsf{in}\ (xy)h'' \\
\mathcal{C}(\mathsf{ref} M) & = \lambda h.\mathsf{let}\ (h', x) = \mathcal{C}(M)h,\ r = (New\ h')\ \mathsf{in}\ (Ext\ h'rx, r) \\
\mathcal{C}(!M) & = \lambda h.\mathsf{let}\ (h', r) = \mathcal{C}(M)h\ \mathsf{in}\ (h', Read\ h'r) \\
\mathcal{C}(M := N) & = \lambda h.\mathsf{let}\ (h', r) = \mathcal{C}(M)h,\ (h'', x) = \mathcal{C}(N)h'\ \mathsf{in}\ (Write\ h''rx, *)
\end{array}
$$

Table 17.2: Simulating the heap in a functional language

$$
\frac{x : A \in \Gamma}{\Gamma; \Sigma \vdash x : A} \qquad \frac{r : A \in \Sigma}{\Gamma; \Sigma \vdash r : \mathsf{Ref}\ A}
$$

$$
\frac{\Gamma, x : A; \Sigma \vdash M : B}{\Gamma; \Sigma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma; \Sigma \vdash M : A \to B \quad \Gamma; \Sigma \vdash N : A}{\Gamma; \Sigma \vdash MN : B}
$$

$$
\frac{}{\Gamma; \Sigma \vdash * : 1} \qquad \frac{\Gamma; \Sigma \vdash M : A}{\Gamma; \Sigma \vdash \mathsf{ref}\ M : \mathsf{Ref}\ A}
$$

$$
\frac{\Gamma; \Sigma \vdash M : \mathsf{Ref}\ A}{\Gamma; \Sigma \vdash !M : A} \qquad \frac{\Gamma; \Sigma \vdash M : \mathsf{Ref}\ A \quad \Gamma; \Sigma \vdash N : A}{\Gamma; \Sigma \vdash M := N : 1}
$$

Table 17.3: Typing rules for the λ-calculus with references

We denote with $\underline{r}$ the Church numeral which corresponds to the reference r. A λ-term M of the λ-calculus with references is compiled into a function which takes a heap h as an argument and returns a pair composed of the heap h modified according to the side-effects of M and a value which corresponds to the outcome of the computation of M. There is some similarity between records (cf. chapter 16) and heaps in that a record is a finite function defined on a set of labels and a heap is a finite function defined on a set of references. However, references, unlike labels, can be generated during the computation, are treated as first class-values, and the value associated with a reference can be updated.

17.2 Typing

We consider the problem of extending the propositional type system discussed in chapter 10, Table 10.1, to the λ-calculus with references. To type the value $*$ we introduce a basic type 1 whose only value is $*$. Moreover, we introduce a new type constructor Ref. A value of type $\mathsf{Ref}\ A$ is a reference which can contain values of type A. In order to type a λ-term we have to make hypotheses on the type of its free variables and of the references that occur in it. Consequently, we introduce a notion of *heap context* Σ of the shape $r_1 : A_1, \ldots, r_n : A_n$. If $r : A \in \Sigma$ then the reference r is associated with values of type A. Table 17.3 gives the type system for λ-terms.

Besides λ-terms we need to type heaps too. Consider the λ-term without references:

$$
\mathsf{let}\ x = \mathsf{ref}\ (\lambda z : A.z)\ \mathsf{in}\ \mathsf{let}\ y = \mathsf{ref}\ (\lambda z : A.!xz)\ \mathsf{in}\ x := \lambda z : A.!yz\ . \tag{17.1}
$$

By reducing it, we can produce the following heap: $h_0 = [\lambda x : A.(!r_2)x/r_1, \lambda x : A.(!r_1)x/r_2]$. Notice that the values associated with r_1 and r_2 depend on r_2 and r_1 respectively. Thus to type a heap we have to find a heap context which assigns a type to all the references of the heap which is coherent with the type of the values associated with the references. Also we require that all the references in the values of the heap belong to the domain of definition of the heap. This leads to the following rule for typing a heap with respect to a heap context:

$$\frac{dom(\Sigma) = dom(h) \qquad \emptyset; \Sigma \vdash h(r) : A \text{ for } r \in dom(h)}{\Sigma \vdash h} \ . \tag{17.2}$$

We write $\Gamma; \Sigma \vdash (M, h) : A$ if $\Gamma; \Sigma \vdash M : A$ and $\Sigma \vdash h$.

Example 293 *The heap h_0 produced by the λ-term (17.1) above can be typed in the heap context: $\Sigma = r_1 : A \to A, r_2 : A \to A$.*

Proposition 294 *The typing system enjoys the following properties:*

1. *If $\Gamma, x : A; \Sigma \vdash M : B$ is derivable and $x \notin \mathsf{fv}(M)$ then $\Gamma; \Sigma \vdash M : B$ is derivable.*
2. *If $\Gamma, x : A; \Sigma \vdash M : B$ and $\Gamma; \Sigma \vdash V : A$ are derivable then $\Gamma; \Sigma \vdash [V/x]M : B$ is derivable.*

Proof. By induction on the proof height. □

We now discuss the way typing is preserved by reduction. Notice that during reduction the domain of definition of the heap can grow since the operator ref may dynamically generate new references. Hence we also need to extend the heap context. We write $\Sigma' \supseteq \Sigma$ if Σ' is an extension of Σ. We notice the following weakening property of the heap context.

Proposition 295 *If $\Gamma; \Sigma \vdash M : A$ is derivable and $\Sigma' \supseteq \Sigma$ then $\Gamma; \Sigma' \vdash M : A$ is derivable.*

Then we can state the following subject reduction property (proof left to the reader).

Proposition 296 *If $\Gamma; \Sigma \vdash (M, h) : A$ and $(M, h) \to (M', h')$ then there is $\Sigma' \supseteq \Sigma$ such that $\Gamma; \Sigma' \vdash (M', h') : A$.*

Exercise 297 *Suppose we have 'abstract types' R and H and that we can assign the following types to the heap-manipulating functions, where A can be any type:*

$$\begin{array}{ll} \mathit{New} : H \to R \ , & \mathit{Ext} : H \to R \to A \to H \ , \\ \mathit{Read} : H \to R \to A \ , & \mathit{Write} : H \to R \to A \to H \ . \end{array}$$

For every propositional type A and type context Γ define a type translation $\underline{A}$ and context translation $\underline{\Gamma}$, and show that the compilation function in Table 17.2 is type preserving in the sense that if $\Gamma; \emptyset \vdash M : A$ according to the rules in Table 17.3 then $\underline{\Gamma} \vdash \mathcal{C}(M) : H \to H \times \underline{A}$.

17.3 Typing anomalies

As suggested by the λ-term (17.1) above, simply typed λ-terms with references can produce circular heaps. In fact it is possible to use references to define general recursive functions. First, let us consider a *minimal* example of typable and *looping* computation. Set:

$$M_1 \equiv \mathsf{ref}\ (\lambda x : 1.x)\ , \qquad M_2 \equiv \mathsf{let}\ y = M_1\ \mathsf{in}\ y := (\lambda x : 1.(!y)x)\ ;\ (!y)\ .$$

Then $\vdash M_1 : \mathsf{Ref}\ (1 \to 1)$ and $\vdash M_2 : 1 \to 1$ and there is an infinite reduction starting with M_2*. We can generalize this idea to define a function f of type $A \to B$ which satisfies a recursive equation $f = \lambda x : A.M$ where M may depend on f. Let $\lambda x : A.N$ be any λ-term of type $A \to B$. Then we set:

$$M_1 \equiv \mathsf{ref}\ (\lambda x : A.N)\ , \qquad M_2 \equiv \mathsf{let}\ y = M_1\ \mathsf{in}\ y := (\lambda x : A.[!y/f]M)\ ;\ (!y)\ .$$

Initially, y is a reference containing a fake function. Then we replace the fake function with the real function where each call to f is replaced by $!y$. Then y is a reference which contains a value which refers to the reference y. This circularity allows to simulate recursion.

Another curious phenomenon arises when we try to mix references and subtyping. Namely, from $A \leq B$ we *cannot* infer $\mathsf{Ref}\ A \leq \mathsf{Ref}\ B$ (or $\mathsf{Ref}\ A \leq \mathsf{Ref}\ B$). The Ref type constructor is neither monotonic nor anti-monotonic with respect to the subtyping pre-order. In practice, this means that no proper subtyping is possible on reference types. To see this, suppose $A \leq B$ where for instance:

$$A = \{\ell_1 : C, \ell_2 : C\} \quad \leq \quad \{\ell_1 : C\} = B\ .$$

Assume Ref is anti-monotonic and $x : \mathsf{Ref}\ B$ then we should also have $x : \mathsf{Ref}\ A$ and $(!x.\ell_2)$ will produce an error. On the other hand, if Ref is monotonic and $x : \mathsf{Ref}\ A$ then we should also have $x : \mathsf{Ref}\ B$ and $x := \{\ell_1 = V\};\ !x.\ell_2$ will produce an error.

As a third and final typing anomaly, let us notice that the *polymorphic generalization* (cf. chapter 12) of a *reference* may also lead to *errors*. For instance, consider:

$$\mathsf{let}\ x = \mathsf{ref}\ (\lambda x.x)\ \mathsf{in}\ x := (\lambda x.x + 1); (!x)\ \mathsf{true}\ . \tag{17.3}$$

ML-like languages avoid these problems by allowing polymorphic generalization only on *values*. For instance, the programming language *ocaml accepts*:

$$\mathsf{let}\ x = (\lambda x.x)\ \mathsf{in}\ x*; x\ \mathsf{true}\ , \tag{17.4}$$

but *rejects* the dangerous expression (17.3) above as well as the following innocuous one:

$$\mathsf{let}\ x = (\lambda y.(\lambda x.x))2\ \mathsf{in}\ x*; x\ \mathsf{true}\ . \tag{17.5}$$

In practice, most programs seem to meet this restriction.

17.4 Summary and references

Heaps can be regarded as an abstraction of computer memory. We have considered an extension of the λ-calculus with operations to extend, read, and modify the heap. Expressions in

this extended λ-calculus may have side effects and can be understood as functions that take a heap and produce a new heap and a value.

References introduce the possibility to define recursive data structures and functions. This power comes at a price in that the ideas developed in the purely functional setting cannot be readily lifted to the λ-calculus with side effects. For instance, termination of typable programs fails, no proper subtyping is possible on reference types, and polymorphic generalization is unsound (but in practice it can be fixed [Wri95]).

One may argue that these failures are due to the fact that the usual type systems neglect side effects completely. To address this issue, so called *type and effect* systems [LG88] have been proposed. In these type systems references are abstracted into a finite set of *regions* and types become *dependent* on such regions. In particular, an expression is now expected to produce both an effect and a value (this is an *abstraction* of the idea mentioned above where an expression with side effects is expected to produce a heap and a value). Type and effect system have been applied to the design of static mechanisms for safe memory deallocation [TT97]. It has also been shown that a *stratified* version of the system can guarantee the strong normalization of the typable λ-terms [Bou10, Ama09].

Chapter 18

Object-oriented languages

The programming paradigms discussed so far are built on the notion of *function*. Indeed term rewriting and the λ-calculus can be regarded as formalisms to define first-order and higher-order functions, respectively and imperative programs can also be regarded as functions operating over the heap. In this chapter, we discuss the situation for object-oriented programs. We start with a minimalist object-oriented language which is type-free and without side-effects. We then gradually enrich this language with *side-effects* and *types* to obtain a language which corresponds to a (tiny) fragment of the *Java* programming language (of which the reader is supposed to have a superficial knowledge). We refer to this language as untyped/typed J. Along the way, we discuss the compilation of untyped J to an extension of the λ-calculus with records, recursion, and, possibly, references. Thus objects can also be understood as functions. However, typed object-oriented languages such as *Java* differ from languages of the *ML* family in that they require some degree of type-checking at run time, *i.e.*, type errors at run-time are possible.

18.1 An object-oriented language

In first approximation, an object is a *record* (cf. chapter 16) whose labels are traditionally partitioned into *fields* and *methods*. Usually, fields are mapped to (basic) values describing the internal state of the object while methods are mapped to functions that allow to manipulate this state. As in records, the 'dot-notation' is used to access fields and methods, *e.g.*, if o is an object and f a field then $o.f$ is the value associated with the field f.

In object-oriented languages such as Java, the creation of objects follows certain *patterns* known as *class declarations*. So objects are *classified* according to the class declaration that is used at the moment of their creation. Class declarations are designed so that fields and methods are suitably initialized when the object is created. Unlike in the λ-calculus with *records* of chapter 16, *recursion* is built into object-oriented languages. First, class declarations may be mutually recursive, and second there is a special variable this (self is also used sometimes) which allows to refer to the object itself within, say, the body of one of its methods. For instance, an object o may consist of a field val which is mapped to an integer and a method inc which is mapped to the function:

$$\lambda x.\mathsf{this.val} = \mathsf{this.val} + x \ .$$

Then the effect of invoking $o.\mathsf{inc}$ is that of increasing by x the value contained in the val field

of the object o.

Class declarations

We reserve $C, C', D, \ldots$ for class names. Each class name corresponds to a distinct class declaration. Usually, class declarations are built incrementally. At the very beginning, there is a class Object without fields and methods. Then whenever we introduce a new class declaration we say that it *extends* another class declaration. For instance, one can declare a class C which extends the class D and includes a field f and a method m as follows:

class C extends D = {	(class declaration)
$\cdots$ C′ f $\cdots$	(field declarations)
$\cdots$ D′ m (D_1 $\mathsf{x}_1, \ldots, \mathsf{D}_n$ x_n){ e } $\cdots$}	(method declarations).

We are using here a notation based on *Java* where we specify the class C' of the object in the field f as well as the classes $\mathsf{D}_1, \ldots, \mathsf{D}_n$ of the objects $\mathsf{x}_1, \ldots, \mathsf{x}_n$ the (function associated with the) method m is expecting as input and the class D' of the object it returns as a result.

Expressions

The body of a method is an *expression* whose syntax is defined as follows:

e	::=	id	(variable)
		v	(value)
		new $C(e_1, \ldots, e_n)$	(object generation)
		$e.f$	(field read)
		$e.m(e_1, \ldots, e_n)$	(method invocation)
		$e.f := e$	(field write)
		$e; e$	(sequentialization)
		$(C)(e)$	(casting)

We have split the expressions in 3 groups. The first group is composed of (object) variables, (object) values (to be defined next), an operator new to generate an object of the class C while initializing its fields with the values of the expressions $e_1, \ldots, e_n$, and the selection operator for fields and methods. As already mentioned, among the variables, we reserve the variable this to refer to the object on which a method is invoked. The second group is optional and corresponds to an *imperative extension* of the basic language where fields are modifiable, and therefore the sequentialization of side effects is relevant. The third group is also optional and consists of a casting operator. This operator is only relevant if we are interested in a *type system* for the language. We anticipate that the role of such a type system is *not* to avoid errors (cf. exercise 203) but to *localize* them in certain points of the computation.

Values

The definition of a value expression depends on whether we are considering the imperative extension or not. In the imperative extension, we assume all fields are modifiable. To model field assignment we proceed as in chapter 17. Namely, we assume an infinite set of *references*

R with elements $r, r, \ldots$ and define a *heap* h as a finite domain partial function mapping references to values. In this case, a *value* v has the shape:

$$v ::= C(r_1, \ldots, r_n) \qquad n \geq 0 \qquad \text{(values, imperative case)}, \tag{18.1}$$

where C is a class name (the class of the object) and $r_1, \ldots, r_n \in R$ are references corresponding to the modifiable fields of the object.

In the non-imperative, say, *functional*, case, fields are initialized when the object is created and they are never modified. Then we can just regard values as the closed first-order terms built over the signature of class names where the arity of a class names is the number of the class fields:

$$v ::= C(v, \ldots, v) \qquad \text{(values, functional case)}. \tag{18.2}$$

We pause to remark that to define the reduction rules of the language, it is convenient to include values in the syntactic category of expressions, however values *never* appear in a source program. Incidentally, in chapter 17, we took a similar approach by considering references as values.

Well-formed programs

A *program* is composed of a list of class declarations and a distinguished expression where the computation starts (in *Java* this distinguished expression would be the body of a main method). The final value of the distinguished expression can be taken as the output of the program. As for the input, we shall assume for simplicity that it is coded as part of the distinguished expression.

As mentioned above, each class declaration extends another class declaration. This induces a binary relation on class names. We denote with $\leq$ the reflexive and transitive closure of this relation and we assume that if $C \leq D$ and $D \leq C$ then $C = D$. Under this hypothesis, we can represent the subtyping relation as an *inheritance tree* having as root the Object class.

The feature of declaring a class by extending another one makes programs more compact but requires some verification. A *well-formed program* must satisfy certain conditions concerning fields and methods.

1. If $C \leq D$ then C *inherits* all the fields of D. It is required that there are no name conflicts among the fields. Thus, by crossing the inheritance tree towards the root one must *not* find two fields with the same name.

2. Also, if $C \leq D$ then C inherits all the methods of D. However, in this case C may *redefine* (in the object-oriented jargon one says *override*) a method. A constraint that only concerns the typed version of the language requires that the type of the method does *not* change.

It is convenient to introduce a certain number of functions that will be used in formulating the reduction rules and the typing rules.

- *field*(C) returns the list $f_1 : C_1, \ldots, f_n : C_n$ of the fields accessible by an object of the class C along with their expected classes. Upon generation, an object of the class C must receive n arguments so as to initialize its fields. To avoid ambiguities, we assume an enumeration of the field names and suppose the function *field* returns the fields in growing order. In *Java*, the initialization of the fields is made explicit by defining a constructor method in the class.

```
public class Bool extends Object {
    public Object ite (Object x, Object y){return new Object();} }
public class True extends Bool{
    public Object ite (Object x, Object y){return x;} }
public class False extends Bool{
    public Object ite (Object x, Object y){return y;} }
public class Num extends Object {
    public Bool iszero (){return new Bool();}
    public Num pred(){return new Num();}
    public Num succ(){return new Num();} }
public class NotZero extends Num{
    public Num pd;
    public NotZero(Num x){pd=x;}
    public Bool iszero(){return new False();}
    public Num pred(){return this.pd;}
    public Num succ(){return new NotZero(this);} }
public class Zero extends Num{
    public Bool iszero(){return new True();}
    public Num pred(){return new Zero();}
    public Num succ(){return new NotZero(this);} }
```

Table 18.1: Some class declarations in J (with *Java* syntax).

- $mbody(m, C)$ returns the function that corresponds to the method m in the class C. For instance, if $mbody(m, C) = \lambda x_1, \ldots, x_n.e$ then $x_1, \ldots, x_n$ are the formal parameters and e is the expression associated with the method, respectively.
- In the *typed* version of the language, it will also be useful to have a function *mtype* such that $mtype(m, C)$ returns the type of the method m of the class C and a predicate *override* such that $override(m, D, C^* \to C)$ holds if and only if $mtype(m, D)$ is defined and it coincides with $C^* \to C$.

Example 298 *In Table 18.1, we consider a list of class declarations which allows to represent boolean values and natural numbers in unary notation. The examples are written in the slightly more verbose notation of the Java programming language. As already mentioned, Java requires a constructor method to build an object in a class with fields. Moreover, Java distinguishes between private and public declarations while in* J *all declarations are public. These are really minor syntactic differences and therefore the typed version of the* J *language can be regarded as a subset of Java. Notice that the proposed representation of the conditional via the method* ite *is* strict *(both branches are evaluated); a more realistic fragment of Java would include a* non-strict *conditional.*

Exercise 299 (programming) *With reference to the code in Table 18.1:*

1. *Enrich the classes for the booleans and natural numbers with a printing method which prints (a representation of) the object on the standard output using Java's printing functions.*
2. *Enrich the classes for natural numbers with an* isequal *method that takes a number object and checks whether it is equal to the one on which the method is invoked.*

Call-by-value evaluation contexts

$$E ::= [\,] \mid \mathsf{new}\ C(v^*, E, e^*) \mid E.f \mid E.m(e^*) \mid v.m(v^*, E, e^*) \mid (C)(E) \mid E.f := e \mid v.f := E \mid E; e$$

Reduction rules

$$\frac{r^* \text{ distinct and } \{r^*\} \cap dom(h) = \emptyset}{(E[\mathsf{new}\ C(v^*)], h) \to (E[C(r^*)], h[v^*/r^*])} \quad \text{(object generation)}$$

$$\frac{field(C) = f_1 : C_1, \ldots, f_n : C_n \quad 1 \le i \le n}{(E[C(r_1, \ldots, r_n).f_i], h) \to (E[h(r_i)], h)} \quad \text{(field read)}$$

$$\frac{mbody(m, C) = \lambda x_1, \ldots, x_n.e}{(E[C(r^*).m(v_1, \ldots, v_n)], h) \to (E[[v_1/x_1, \ldots, v_n/x_n, C(r^*)/\mathsf{this}]e], h)} \quad \text{(method invocation)}$$

$$\frac{field(C) = f_1 : C_1, \ldots, f_n : C_n \quad 1 \le i \le n}{(E[C(r_1, \ldots, r_n).f_i := v], h) \to (E[\mathsf{Object}()], h[v/r_i])} \quad \text{(field write)}$$

$$\frac{}{(E[v; e], h) \to (E[e], h)} \quad \text{(sequentialization)}$$

$$\frac{C \le D}{(E[(D)(C(r^*))], h) \to (E[C(r^*)], h)} \quad \text{(casting)}$$

Table 18.2: Evaluation contexts and reduction rules for J

3. *Define classes to represent lists of pairs of natural numbers $(n_1, m_1) \cdots (n_k, m_k)$, where $n_1, \ldots, n_k$ are all distinct, along with methods to: (1) given n, read the number m associated with it, (2) given n, replace the number associated with it with m, (3) extend the list with a new pair (n, m), (4) given n, remove from the list the pair (n, m), (5) print (a representation of) the list on the standard output.*

Reduction rules

Table 18.2 introduces the syntactic category of *evaluation contexts* which correspond to a *call-by-value, left to right* reduction strategy and the related reduction rules which are based on judgments of the shape:

$$(e, h) \to (e', h') \quad \text{(reduction judgment, imperative).} \tag{18.3}$$

At the beginning of the computation we assume that the heap h is empty. Then the reduction rules maintain the following invariant: for all reachable configurations (e, h), all the references in e and all the references that appear in a value in the codomain of the heap h are in the domain of definition of the heap ($dom(h)$). This guarantees that whenever we look for a fresh reference it is enough to pick a reference which is not in the domain of definition of the current heap. Notice that upon invocation of a method on an object, the object replaces the reserved variable this in the body of the method. Also, the reduction rule for casting consists of a form of run-time type-check: the computation of a casted object $(D)(C(r^*))$ may proceed only if $C \le D$.

$$\frac{\mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n}{E[\mathsf{new}\ C(v_1, \ldots, v_n)] \to E[C(v_1, \ldots, v_n)]} \quad \text{(object generation)}$$

$$\frac{\mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n \quad 1 \leq i \leq n}{E[C(v_1, \ldots, v_n).f_i] \to E[v_i]} \quad \text{(field read)}$$

$$\frac{\mathit{mbody}(m, C) = \lambda x_1, \ldots, x_n.e}{E[C(v^*).m(v_1, \ldots, v_n)] \to E[[v_1/x_1, \ldots, v_n/x_n, C(v^*)/\mathsf{this}]e]} \quad \text{(method invocation)}$$

$$\frac{C \leq D}{E[(D)C(v^*)] \to E[C(v^*)]} \quad \text{(casting)}$$

Table 18.3: Simplified reduction rules for the functional fragment of J

The specification of the functional fragment of J where fields are immutable can be substantially simplified. Values are now the closed first-order terms built over the signature of class names (cf. grammar (18.2)). The evaluation contexts and the reduction rules for assignment and sequentialization can be dropped. The remaining rules are based on a judgment of the shape $e \to e'$ (we drop the heap) and are specified in Table 18.3.

18.2 Objects as records

We define an encoding of the functional, type free object-oriented language into a call-by-value λ-calculus extended with records and a fixed point combinator Y (in turn, records and the fixed point combinator could be encoded in the λ-calculus). As a first step, we assume each class declaration is completely expanded so that we can associate with each class name the list of its fields and its methods with the related bodies. So we have a system of class declarations of the shape (class names are omitted when irrelevant):

$$\mathsf{class}\ C\ \{f_1, \ldots, f_h, m_1 = \lambda x_1^*.e_1, \ldots, m_k = \lambda x_k^*.e_k\}\ . \tag{18.4}$$

The methods' bodies e_i may generate objects of other classes and may refer to the object itself via the variable this. This entails that class generators are mutually recursive and the variable this is defined via a fixed point combinator.

Following this intuition, we define a compilation function $\mathcal{C}$. We suppose the class names are enumerated as $C_1, \ldots, C_m$ and we reserve a fresh variable c and the labels $\mathsf{1}, \ldots, \mathsf{m}$. The variable c will be defined recursively as a record with labels $\mathsf{1}, \ldots, \mathsf{m}$ such that the function associated with the label i is the generator for the objects of the class C_i. On expressions (which are not values or casted objects), the compilation function is simply defined as follows:

$$\begin{array}{ll}
\mathcal{C}(x) & = x \\
\mathcal{C}(\mathsf{new}\ C_i(e_1, \ldots, e_n)) & = (c.\mathsf{i})\ \mathcal{C}(e_1) \cdots \mathcal{C}(e_n) \quad (c \text{ fresh variable}) \\
\mathcal{C}(e.f) & = \mathcal{C}(e).f \\
\mathcal{C}(e.m(e_1, \ldots, e_n)) & = (\mathcal{C}(e).m)\ \mathcal{C}(e_1) \cdots \mathcal{C}(e_n)\ .
\end{array}$$

For each declaration of a class C of the shape (18.4), we define the λ-term N_C where $y_1, \ldots, y_h$

are fresh variables:

$$\begin{array}{ll} R_C & \equiv \{f_1 = y_1, \ldots, f_h = y_h, m_1 = \lambda x_1^*.\mathcal{C}(e_1), \ldots, m_k = \lambda x_k^*.\mathcal{C}(e_k)\} \\ N_C & \equiv \lambda y_1, \ldots, y_h.Y(\lambda \mathsf{this}.R_C) \ . \end{array} \tag{18.5}$$

Intuitively, N_C is the generator for objects of the class C. The system of class declarations is reduced to one fixed point equation:

$$C \equiv Y(\lambda c.\{1 = N_{C_1}, \ldots, \mathsf{m} = N_{C_m}\}) \ . \tag{18.6}$$

Finally, a program composed of m class declarations $C_1, \ldots, C_m$ and an expression e is compiled into the λ-term:

$$\mathsf{let}\ c = C\ \mathsf{in}\ \mathcal{C}(e) \ . \tag{18.7}$$

As a concrete example, suppose the program P is composed of 2 class declarations C_i each with a field f_i and a method m_i with body $\lambda x_i.e_i$, $i = 1, 2$, and a main expression e. Then we have:

$$\begin{array}{ll} N_{C_1} & \equiv \lambda y_1.Y(\lambda \mathsf{this}.\{f_1 = y_1, , m_1 = \lambda x_1.\mathcal{C}(e_1)\}) \\ N_{C_2} & \equiv \lambda y_2.Y(\lambda \mathsf{this}.\{f_2 = y_2, , m_2 = \lambda x_2.\mathcal{C}(e_2)\}) \\ C & \equiv Y(\lambda c.\{1 = N_{C_1}, 2 = N_{C_2}\}) \\ \mathcal{C}(P) & \equiv \mathsf{let}\ c = C\ \mathsf{in}\ \mathcal{C}(e) \ . \end{array}$$

Exercise 300 *Extend this encoding to the language with* mutable *fields. In this case, it is convenient to take as target language a call-by-value λ-calculus with records and* references *(cf. chapter 17).*

18.3 Typing

We design a type system for the full object-oriented language we have introduced. To this end, we assume a type context Γ has the shape $x_1 : C_1, \ldots, x_n : C_n$ and consider typing judgments of the shape: $\Gamma \vdash e : C$. A general goal of a type system for an object-oriented language is to guarantee that every invocation of a field or a method on an object is compatible with the class to which the object belongs. Let us notice however that an incorrect application of the casting (*downcasting*) may compromise this property. For instance, we could write the expression:

$$(\mathsf{Bool})((\mathsf{new\ Object}())).\mathsf{ite}(\mathsf{new\ True}(), \mathsf{new\ False}())) \ .$$

To avoid this situation, we could consider the following rule:

$$\frac{\Gamma \vdash e : D \quad D \leq C}{\Gamma \vdash (C)(e) : \ C} \ .$$

In this rule, we can cast an object of the class D as an object of the class C only if the class D extends the class C. This is in agreement with the intuition that objects are records and that an object of the class D can handle all the invocations addressed to an object of the class C (cf. subtyping rules for records in chapter 16). However this rule is *too* constraining. For instance, it does not allow the typing of the expression:

$$(\mathsf{Bool})((\mathsf{new\ True}()).\mathsf{ite}(\mathsf{new\ True}(), \mathsf{new\ False}())) \ ,$$

```
class C extends Object{
    public void m(){return;}}
class D extends Object{
    public void m(){return;}}
class Main{
    public static void main (String[] args){
        D d = new D();
        C c = new C();
        ((C)((Object)(d))).m(); //this types, but rises an exception at run time.
        ((C)(d)).m();           //this does not type, but it is a reduced of the above!
        return; }}
```

Table 18.4: Typing anomaly in *Java*

as the result of the method ite belongs to the class Object and Object $\not\leq$ Bool. Then, in *Java*, the rule for casting can be formulated as follows:

$$\frac{\Gamma \vdash e : D \qquad (C \leq D \text{ or } D \leq C)}{\Gamma \vdash (C)(e) : C} \ .$$

In other terms, the casting is *forbidden* if C and D are *incomparable*. However, this property is not preserved by reduction! Let C, D be two incomparable classes and let e be an expression of type C. Then the expression (D)((Object)(e)) is well typed, but it reduces to the expression (D)(e) which is not. By climbing and descending the inheritance tree we can connect incomparable classes. Table 18.4 gives a concrete example of this phenomenon in *Java*.

Because preservation of typing by reduction is a desirable property, we formulate the typing rule for casting as follows:

$$\frac{\Gamma \vdash e : D}{\Gamma \vdash (C)(e) : C} \ .$$

At typing time, we do not try to verify that the value $C'(r^*)$ resulting form the evaluation of the expression e is such that $C' \leq C$. Instead, we delay this verification at running time. If the condition is not satisfied then reduction is stuck (alternatively, an error message could be produced).

Table 18.5 specifies the rules to type expressions that do not contain values (as source programs do). An important point to notice is that the typing rules allow to use an object of the class C where an object of the class D is expected as long as C is a sub-class of D. This is a form of *subtyping* (cf. chapter 16). The rationale is that an object of the sub-class C will be able to handle all the field and method invocations which could be performed on an object of the super-class D. Indeed, objects of the class C have all fields of the class D and may redefine methods of the class D provided their type is unchanged.

Beyond expressions, we also need to check the typing of the class declarations. Suppose a method m of the class C has the shape:

$$C_0\ m(C_1\ x_1, \ldots, C_n\ x_n)\{e\}\ ,$$

and that the class C extends the class D. Then the following must hold:

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C} \qquad \frac{\begin{array}{c}\mathit{field}(C) = f_1 : D_1, \ldots, f_n : D_n \\ \Gamma \vdash e_i : C_i, \quad C_i \leq D_i, \quad 1 \leq i \leq n\end{array}}{\Gamma \vdash \mathsf{new}\ C(e_1, \ldots, e_n) : C}$$

$$\frac{\Gamma \vdash e : C \quad \mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n}{\Gamma \vdash e.f_i : C_i} \qquad \frac{\begin{array}{c}\Gamma \vdash e : C \quad \mathit{mtype}(m, C) = (C_1, \ldots, C_n) \to D \\ \Gamma \vdash e_i : C_i' \quad C_i' \leq C_i \quad 1 \leq i \leq n\end{array}}{\Gamma \vdash e.m(e_1, \ldots, e_n) : D}$$

$$\frac{\Gamma \vdash e : D}{\Gamma \vdash (C)(e) : C}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : C \quad \mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n \\ \Gamma \vdash e' : D_i \quad D_i \leq C_i\end{array}}{\Gamma \vdash e.f_i := e' : \mathsf{Object}} \qquad \frac{\Gamma \vdash e_1 : C_1 \qquad \Gamma \vdash e_2 : C_2}{\Gamma \vdash e_1; e_2 : C_2}$$

Table 18.5: Typing rules for J program expressions

1. $\mathit{override}(m, D, (C_1, \ldots, C_n) \to C_0)$,
2. $x_1 : C_1, \ldots, x_n : C_n, \mathsf{this} : C \vdash e : C_0'$ and $C_0' \leq C_0$.

A *class* is well typed if all its methods are well typed in the sense above. Finally, a program is well typed if all its classes are well typed and the distinguished expression is well typed in the empty type context. For instance, the reader may check that we can type the class declarations in example 298.

Exercise 301 (more programming) *Design a compiler from the* Imp *language (cf. chapter 1) to the typed* J *language. We outline a possible strategy.*

1. *Consider a restricted set of arithmetic expressions and boolean conditions that can be easily coded in* J*. For instance, just work with natural numbers in unary notation and a boolean condition that checks if a number is zero (cf. Table 18.1).*
2. *Represent variables as unary numbers and implement a state as a finite list of pairs composed of a variable and a number. A state is compiled into an object of a class* State *with methods to read, write, extend, and restrict (cf. exercise 298).*
3. *Define a class* Code *with subclasses* Skip, Assignment, Conditional,... *which correspond to the various ways of composing statements in* Imp*. It is assumed that each object of the class* Code *has a method* execute *that takes as argument an object of the class* State*.*
4. *For all programs P and states s of the* Imp *language define a compilation into a* J *expression $e = \mathcal{C}(P).\mathsf{execute}(\mathcal{C}(s))$ with the following properties: (1) if the expression e evaluates to a value v then v is the representation of a state s' such that $(P, s) \Downarrow s'$. (2) the evaluation of e never produces an exception or a type error,*

As already mentioned, the task of the type system is to localize the type errors around the application of the casting reduction rule. We formalize this property for the functional

case and leave it to the reader the extension to the imperative case. To formulate the subject reduction, we add a rule to type (functional) values which is similar to the rule for the new:

$$\frac{\begin{array}{c}\mathit{field}(C) = f_1 : D_1, \ldots, f_n : D_n \\ \Gamma \vdash v_i : C_i, \quad C_i \leq D_i, \quad 1 \leq i \leq n\end{array}}{\Gamma \vdash \ C(v_1, \ldots, v_n) : C} \ . \tag{18.8}$$

In order to reason about a method selection we need a substitution property (cf. proposition 289).

Proposition 302 *If $x_1 : C_1, \ldots, x_n : C_n \vdash e : C$, $\emptyset \vdash v_i : D_i$, and $\vdash D_i \leq C_i$ for $i = 1, \ldots, n$ then $\emptyset \vdash [v_1/x_1, \ldots, v_n/x_n]e : C'$ and $\vdash C' \leq C$.*

We also need to check the usual decomposition property (cf. proposition 154).

Proposition 303 *Suppose $\emptyset \vdash e : C$. Then either e is a value or there is a unique evaluation context E and redex Δ such that $e \equiv E[\Delta]$, $\emptyset \vdash \Delta : D$ for some D. and Δ has one of the following shapes: $\Delta ::= \mathsf{new}\ C(v^*) \mid (D)C(v^*) \mid C(v^*).f \mid C(v^*).m(v'^*)$.*

We also observe that it is always possible to replace an expression with another expression with a smaller type.

Proposition 304 *if $\emptyset \vdash E[e] : C$, $\emptyset \vdash e : D$, $\emptyset \vdash e' : D'$, and $\vdash D' \leq D$ then $\emptyset \vdash E[e'] : C'$ for some C' such that $\vdash C' \leq C$.*

We can then state the subject reduction property for the typed J language as follows.

Proposition 305 *Given a well-typed functional program in J and a well-typed functional expression $\emptyset \vdash e : C$ one of the following situations arises:*

1. *e is a value.*
2. *$e \to e'$, $\emptyset \vdash e' : C'$, and $\vdash C' \leq C$.*
3. *$e \equiv E[(D)(C(v^*))]$ and $\not\vdash C \leq D$.*

Proof. Suppose $\emptyset \vdash e : C$ and e is not a value. Then e has a unique decomposition as $E[\Delta]$ and $\emptyset \vdash \Delta : D$ (proposition 303). We proceed by case analysis on the typing of Δ to show that either the computation is stuck because of a casting error or it can be reduced to an expression e' such that $\emptyset \vdash e' : D'$ and we can then conclude by proposition 304. Proposition 302 is needed to handle the case of a method selection. □

18.4 Summary and references

An object is basically a record and object-oriented languages introduce user friendly mechanisms to define mutually recursive records. Depending on whether fields are modifiable one can distinguish between functional and imperative object-oriented languages (which are those mainly used in practice). In *typed* object-oriented languages, the introduction of a casting operator is necessary in order to have some programming flexibility. In this setting, the goal of a type system is *not* to avoid typing errors but to *localize* them around the usage of the casting operator. The formalization presented in this chapter builds on the paper [IPW01]. The book [Mit03] introduces the main design issues in object-oriented programming languages.

Chapter 19

Introduction to concurrency

In computer science, we are used to the idea of regarding a piece of software and/or hardware as a *system*, *i.e.*, a compound of interacting and interdependent components with varying names such as *threads* or *processes* that we use as synonymous.

Starting from this chapter, the general goal is to formalize and reason on systems where several threads/processes *compete* for the same resources (e.g. write a variable or a channel). Most of the time, this results into *non-deterministic* behavior which means that with the same input the system can move to several (incomparable) states. For instance, the computation of a circuit may be non-deterministic due to the unpredictable delays in the propagation of signals. Similarly, the computation of an operating system may be non-deterministic due to unpredictable delays in managing the accesses to memory. We stress that non-determinism is both a way of representing our partial knowledge of the system and a method to keep its specification general. For instance, we may want to prove that a certain algorithm is correct independently of the scheduling policy or the evaluation strategy chosen.

Some authors distinguish *parallel* from *concurrent* systems. The former are a subclass of the latter that typically exhibit a deterministic behavior. A standard problem in parallel programming is to decompose the task of computing a (deterministic) function into parallel sub-tasks that when executed on suitable hardware will hopefully provide a faster result in terms of throughput and/or latency. We do not develop at all these algorithmic issues.

Besides being non-deterministic, certain concurrent systems may also exhibit a *probabilistic* behavior. In first approximation, this means that at certain points in the computation the next state of the system is determined by tossing a coin. The basic idea we stress in chapter 29 is that non-deterministic and probabilistic transitions should be kept separated and that a computation in a non-deterministic and probabilistic system is described by a transition relation that relates states to distributions over states.

The concurrent systems we consider can be classified according to *two main criteria*:

asynchronous vs. synchronous and shared memory vs. message passing.

The first criterion concerns the *relative speed* of the processes; we mainly focus on *asynchronous* systems where each process proceeds at its own speed, however we shall see in chapter 28 that the techniques can be adapted to *synchronous/timed systems* too, where computation proceeds in phases or rounds. The second criterion concerns the *interaction mechanism* among the processes. In *shared memory*, processes interact by modifying a shared area of memory. Synchronization arises by waiting that a certain condition is satisfied (cf.

lock/unlock, compare and set, P/V, monitors, synchronized methods,...). In *message passing*, processes interact by sending/receiving messages on communication channels. Synchronization arises when receiving (wait for a message to be there) and possibly when sending (if the capacity of the channel is exceeded). The order of transmission is not necessarily respected and various kinds of channels can be considered according to their capacity (bounded/unbounded), the ordering of the messages, and the number of processes accessing the channel (one-to-one, one-to-many, many-to-many, ...).

19.1 A concurrent language with shared memory

To make things concrete, we start looking at a simple instance of an asynchronous and shared memory model. The recipe is rather straightforward: we select a standard imperative language, namely the imperative language Imp considered in chapter 1, and add (i) the possibility of running several commands in parallel on the same shared memory and (ii) a synchronization mechanism. Table 19.1 describes the abstract syntax of the language. We have identifiers, integers, numerical and boolean expressions, and processes. Besides the standard instructions for assignment, sequentialization, branching, and iteration one can declare and initialize a local identifier, start the execution of two processes in parallel, and wait for a boolean condition to hold and then execute *atomically* a sequence of assignments. In particular, the process $\mathsf{await\ true\ do}\ P$ is supposed to execute atomically the process P. To stress this, we also abbreviate it as $\mathsf{atomic}(P)$. In a process $\mathsf{var}\ x = n\ P$, the identifier x is bound in P and obeys the usual rules of renaming. We denote with $\mathsf{fv}(P)$ the set of identifiers occurring free in P.

Next we describe the possible executions of such processes relatively to a *state* of the shared memory which is described as a total function $s : id \to \mathbf{Z}$ from identifiers to integers (exactly as in chapter 1). We recall that expressions and boolean conditions do *not* produce side-effects. Their evaluations rules are defined in chapter 1, Table 1.2. Next, we revisit the small-step reduction rules defined in chapter 1, Table 1.3. Table 19.2 defines the *immediate termination* predicate '$\downarrow$' and gives the *small-step rules* for process execution where the symmetric rule for parallel composition is omitted. The reader may verify that if $P \downarrow$ then for any state s, (P, s) cannot reduce. We write $(P, s) \Downarrow s'$ if $(P, s) \xrightarrow{*} (P', s')$ and $P' \downarrow$. Notice that unlike for the sequential fragment Imp, the relation $\Downarrow$ is *not* a partial function.

The small-step reduction rules embody a certain number of design choices. First, we have assumed that expressions and assignments are executed *atomically*. This is a (grossly) simplifying hypothesis. We could refine the level of granularity of the small step semantics to some extent and thus complicate the reasoning. However, the basic problem we have to face is that it is difficult to determine the 'right' level of granularity. This is due to the fact that

id	$::= x \mid y \mid \cdots$	(identifiers)
n	$::= 0 \mid 1 \mid -1 \cdots$	(integers)
e	$::= id \mid n \mid (e + e) \mid \cdots$	(expressions)
b	$::= e < e \mid \cdots$	(boolean expressions)
P	$::= \mathsf{skip} \mid id := e \mid P; P \mid \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ P \mid \mathsf{while}\ b\ \mathsf{do}\ P \mid$ $\mathsf{var}\ x = n\ P \mid (P \mid P) \mid \mathsf{await}\ b\ \mathsf{do}\ P$	(processes)

Table 19.1: An asynchronous, shared memory model: $\mathsf{Imp}_{\|}$.

$$\frac{}{\mathsf{skip}\downarrow} \qquad \frac{P\downarrow}{\mathsf{var}\ x = n\ P\downarrow} \qquad \frac{P_i\downarrow \quad i=1,2}{(P_1 \mid P_2)\downarrow}$$

$(x := e, s)$	$\to$	$(\mathsf{skip}, s[v/x])$	if $(e,s) \Downarrow v$
$(\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ P', s)$	$\to$	(P, s)	if $(b,s) \Downarrow \mathsf{true}$
$(\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ P', s)$	$\to$	(P', s)	if $(b,s) \Downarrow \mathsf{false}$
$(\mathsf{while}\ b\ \mathsf{do}\ P, s)$	$\to$	$(P; \mathsf{while}\ b\ \mathsf{do}\ P, s)$	if $(b,s) \Downarrow \mathsf{true}$
$(\mathsf{while}\ b\ \mathsf{do}\ P, s)$	$\to$	(skip, s)	if $(b,s) \Downarrow \mathsf{false}$
$(P; P', s)$	$\to$	(P', s)	if $P\downarrow$
$(P; P', s)$	$\to$	$(P''; P', s')$	if $(P,s) \to (P'', s')$
$(\mathsf{var}\ x = n\ P, s)$	$\to$	$(\mathsf{var}\ x = n'\ P', s'[s(x)/x])$	if $(P, s[n/x]) \to (P', s'[n'/x])$
$(P \mid P', s)$	$\to$	$(P'' \mid P', s')$	if $(P,s) \to (P'', s')$
$(\mathsf{await}\ b\ \mathsf{do}\ P, s)$	$\to$	$(\mathsf{await}\ b\ \mathsf{do}\ P, s)$	if $(b,s) \Downarrow \mathsf{false}$
$(\mathsf{await}\ b\ \mathsf{do}\ P, s)$	$\to$	(P', s')	if $(b,s) \Downarrow \mathsf{true}$, $(P,s) \overset{*}{\to} (P', s'), P'\downarrow$

Table 19.2: Immediate termination and small-step reduction for $\mathsf{Imp}_{\|}$

there is no general agreement on the *abstract memory model* that should be presented to the programmer of a concurrent language with 'shared memory'. Ideally, the model should be 'abstract' while allowing for correct and efficient implementations on a variety of architectures.

Second, we assume that the body of an await *always terminates.* In practice, we can simply enforce this condition by requiring that the body contains no while and no await instruction.

Third, a blocked await reduces (busy waiting) which may seem at odd with the usual idea that a process executing a synchronization condition which is not realized suspends its execution. We could indeed formalize this idea at the price of distinguishing between (properly) terminated and deadlocked statements.

Let us consider a few examples that illustrate the expressivity of the language.

Example 306 (P and V) *Assuming assignment atomic (as we do), the operations P and V for manipulating a* semaphore s of capacity k *can be expressed as follows:*

$$\begin{array}{ll} \textit{Initially:} & s := k\ , \\ P(s) = & \mathsf{await}\ s > 0\ \mathsf{do}\ s := s - 1\ , \\ V(s) = & s := s + 1\ . \end{array}$$

In the special case where the initial capacity is 1*, the operations P and V are also called lock and unlock, respectively. By using them, processes can gain exclusive access to a shared resource,* e.g.*, a process can gain the right to execute without interruption,* i.e.*, atomically, a sequence of statements.*

Example 307 (non-deterministic sum) *We want to define a statement:*

$$[b_1 \to P_1 + \cdots + b_n \to P_n]$$

which selects non-deterministically *one of the branches (if any) for which the condition b_i is satisfied and starts running P_i. This can be defined as follows assuming $x, y \notin \mathsf{fv}(b_i \to P_i)$ for $i = 1, \ldots, n$:*

$$\begin{array}{l} \mathsf{var}\ x = 1\ (Q_1 \mid \cdots \mid Q_n),\ \textit{where} \\ Q_i \equiv \mathsf{var}\ y = 1 \\ \qquad \mathsf{await}\ b_i\ \mathsf{do\ if}\ x = 1\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0\ ; \\ \qquad \mathsf{if}\ y = 1\ \mathsf{then}\ P_i\ \mathsf{else\ skip} \end{array}$$

Exercise 308 *(1) Modify the definition so that once the branch i is selected, the continuation P_i is run atomically. (2) With the current definition, a statement such as* $[\mathsf{true} \to \mathsf{skip} + \mathsf{false} \to \mathsf{skip}]$ *does* not *terminate (which is not very satisfying). Adapt the definition to fix this problem.*

Exercise 309 *Suppose we enrich the* $\mathsf{Imp}_{\parallel}$ *language with a* spawn *operator. The process* spawn P *starts the execution of P in parallel and immediately terminates by reducing, say, to* skip*. (1) Propose a formal semantics of the* $\mathsf{Imp}_{\parallel}$ *language with* spawn*. (2) Explain why in general the process* $(\mathsf{spawn}\ P); Q$ *is* not *equivalent to $(P \mid Q)$. (3) Propose a compilation of the enriched language into the enriched language* without *parallel composition,* i.e.*, find a way to simulate parallel composition with* spawn*.*

Example 310 (compare and set) *The* compare and set *(*cas*) operation can be defined as follows (this operation is also called* compare and swap*):*

$$\mathsf{cas}(x, e_1, e_2) \quad = \mathsf{atomic}(\mathsf{if}\ (x = e_1)\ \mathsf{then}\ x := e_2\ \mathsf{else\ skip})\ .$$

We stress that it is essential that the boolean test $x = e_1$ and the assignment $x := e_2$ are executed atomically*. The* cas *operation can be taken as basic building block to solve more complex problems in concurrency. For instance, it can be used to solve the so called* consensus problem *which can be stated as follows. A collection of parallel processes $P_1, \ldots, P_n$ each holding a non-negative integer $v_1, \ldots, v_n$ have to agree on a value which is equal to one of the values held by the processes. A solution to this problem which treats all processes in the same way and avoids centralization points goes as follows. Set a variable x with initial value -1 and then let each process P_i run the following procedure:*

$$\mathsf{decide}(i) = \mathsf{cas}(x,\ -1,\ v_i); \mathit{result}_i := x\ .$$

The first process that runs the decide *procedure will set x to its value $v_i \geq 0$ (atomically, and thus deciding the outcome of the consensus protocol) while the following ones will keep x unchanged.*

19.2 Equivalences: a taste of the design space

We consider the question of building an equivalence on processes on top of the reduction system. In the sequential framework (cf. chapters 1 and 9), we have already noticed that an answer to this question depends on a certain number of factors such as the choice of the observables, the compositionality properties, and the proof methods. With an enlarged range of choices, these factors play a role in the semantics of concurrent systems too. Moreover, new factors appear such as the hypotheses on the scheduling policy.

Observables The equivalence should be compatible with a notion of observation of the processes. If two processes P and P' are equivalent and P enjoys a certain observable property then P' should enjoy that property too. For instance, we may wait till the system comes to a *proper termination* and then observe its final result. As a second example, we may be informed that the system has reached a *deadlock*, *i.e.*, a situation where it has not properly terminated and it cannot progress. As a third example, we may interact with the system during the computation and observe its capabilities. We refer to this observable as *branching* because, as explained in the following example 313, it amounts to observe the branching structure of the computation as opposed to its linearization.

Scheduling We may assume certain properties of the scheduler that controls the order in which parallel processes are executed. For instance, a *preemptive scheduler* will be allowed to interrupt the execution of a process at any point which is compatible with the atomicity assumptions while a *cooperative scheduler* will wait for the process to yield control or to suspend on a synchronization condition. Further, schedulers can be classified according to their ability to execute the various processes in a *fair* way.

Compositionality If a process P is equivalent to the process P' then we should be able to replace P with P' in any (reasonable) process context. In other words, the notion of equivalence should be preserved by some operators of the language, including at least parallel composition.

Proof method We should have a *practical* proof method to check the equivalence of two processes. Depending on the class of processes we are considering, practical may mean that the equivalence can be efficiently *automated* or that the proof has a certain *locality* property.

We elaborate on the first two points (observables and scheduling hypotheses) in the following examples; we shall come back to compositionality and proof methods in the following sections.

Example 311 (termination) *The following process* diverges *(or at least does not reach immediate termination) while producing a sequence* $f(0), f(f(0)), \ldots$ *on the 'output variable'* y *at a 'rate' determined by* x*.*

$y := 0;$
while true do
await $x = 0$ do $(y := f(y); x := 1;)$

Should it be considered equivalent to while true do skip *(a diverging process)?*

Example 312 (deadlock) *Consider the following deadlocked process:*

$$\mathsf{var}\ x = 0\ \mathsf{await}\ x > 0\ \mathsf{do}\ x := x + 1\ .$$

Should it be considered equivalent to while true do skip *(a diverging process, again) or to* skip *(an immediately terminated process)?*

Example 313 (branching) *Consider the following hypothetical* controls of a vending machine:

$$[b_1 \to P_1; [b_2 \to P_2 + b_3 \to P_3]]$$
$$[b_1 \to P_1; [b_2 \to P_2] \ + \ b_1 \to P_1; [b_3 \to P_3]]$$

with the interpretation:

$$\begin{array}{ll} b_1 = \textit{ there is a coin} & P_1 = \textit{ accept the coin} \\ b_2 = \textit{ there is a second coin} & P_2 = \textit{ accept the coin and deliver coffee} \\ b_3 = \textit{ water request} & P_3 = \textit{ deliver water.} \end{array}$$

Are the two controls equivalent*? Well, one may remark that upon accepting the first coin, the second machine decides non-deterministically whether it is ready to wait for a second coin or to deliver water which is rather annoying for the user.*

Example 314 (cooperative) *In preemptive concurrency, a process can be interrupted after any atomic step. In cooperative concurrency, a process is interrupted only when it has terminated or it is suspended on a waiting statement. For instance, the processes:* $x := 1; x := x+1$ *and* $x := 1; x := 2$ *are equivalent in a cooperative (and a sequential) context but not in a preemptive one.*

Example 315 (weak fairness) *Consider the following process:*

$$x := 0; y := 0; ((\mathsf{while}\ x = 0\ \mathsf{do}\ y := y + 1) \mid x := 1)\ .$$

If the process terminates then y *may contain an arbitrary natural number. This is called* unbounded non-determinism*. Moreover, the process is actually* guaranteed to terminate *if we assume that every process that is ready to run will eventually get a chance of running. This assumption is called* weak fairness*.*

Example 316 (strong fairness) *A weak fairness hypothesis is not always enough to guarantee progress. Consider:*

$$x := 0; y := 0; (\ \mathsf{while}\ y = 0\ \mathsf{do}\ x := 1 - x \mid \mathsf{await}\ x = 1\ \mathsf{do}\ y := 1\ )\ .$$

In this example, the first process makes x *oscillate between* 0 *and* 1 *while the second process can really progress only when* $x = 1$*. A scheduler that gives control to the second process only when* $x = 0$ *will not guarantee termination.* Strong fairness *is the assumption that in any infinite execution a process which is infinitely often 'ready to run' will indeed run infinitely often.*

19.3 Summary and references

Early work on the semantics of concurrent processes started in the 60's [Dij65] and was motivated by synchronization problems in operating systems. The first step in defining the semantics of a concurrent language amounts to decide which actions can be regarded as *atomic*. This is an issue which can be hardly underestimated because there is a tension between atomicity and efficient implementations. At any rate, once atomicity is fixed a *small-step* reduction semantics allows to define precisely the state transformations a concurrent process can go through. The second step amounts to decide the observable properties of the system and the execution hypotheses. This step gives rise to a variety of possible equivalences. Compositionality and the existence of practical proof methods are two basic criteria to assess them. The article [KR90] surveys the parallelization of algorithms (which we do not cover).

Chapter 20

A compositional trace semantics

We consider the problem of defining and characterizing a *compositional* equivalence for the $\mathsf{Imp}_{\|}$ model. For the sequential fragment of the $\mathsf{Imp}_{\|}$ model, the input-output interpretation provides a satisfying answer (cf. chapter 1), but the extension to the full concurrent $\mathsf{Imp}_{\|}$ language is not straightforward and rises some interesting issues which are discussed in this and the following chapter.

20.1 Fixing the observables

Following the discussion in section 19.2, a first problem consists in fixing a notion of *observable*. Building on the semantics of the sequential Imp language (chapter 1) we shall take the *input-output behavior* or, equivalently, the *partial correctness assertions* (pca), as basic observable. We warn the reader that while being reasonable, this notion of observable is definitely not the only possible one for concurrent processes; alternatives will be discussed in the following chapters. Let $P, P', \ldots$ be the processes and $s, s', \ldots$ be the memory states introduced in section 19.1. We adapt to processes the definitions presented in chapter 1.

The IO interpretation (cf. definition 1) of a process P is:

$$[\![P]\!]^{IO} = \{(s, s') \mid (P, s) \xrightarrow{*} (P', s') \downarrow\} \ .$$

Also the notion of pca's validity is extended to processes in the obvious way:

$$\models \{A\}\ P\ \{B\} \text{ if } \ \forall s\ \ (s \models A \text{ and } (P, s) \xrightarrow{*} (P', s') \downarrow \text{ implies } s' \models B) \ .$$

Then the the pca interpretation of a process is:

$$[\![P]\!]^{pca} = \{(A, B) \mid \ \models \{A\}\ P\ \{B\}\} \ .$$

Adapting proposition 11, we derive:

$$[\![P_1]\!]^{IO} = [\![P_2]\!]^{IO} \ \text{ iff } \ [\![P_1]\!]^{pca} = [\![P_2]\!]^{pca} \ .$$

Let us take the input-out behavior (or equivalently the partial correctness assertions) as basic observable. As usual, a *context* C is a process with a hole [ ]. E.g.

$$x := 3; [\ ] \mid \mathsf{await}\ x = 3\ \mathsf{do}\ x := x + 1 \ .$$

As already mentioned in chapters 1, 9, and 19 a desirable property of a semantics is that it is *preserved by contexts*, that is:

$$[\![P_1]\!] = [\![P_2]\!] \text{ implies } [\![C[P_1]]\!] = [\![C[P_2]]\!] .$$

If two processes have the same 'compositional semantics' then we can *replace* one for the other in any context. Unfortunately, the following example shows that, unlike in the sequential case (proposition 3), *compositionality fails* for the IO (and pca) interpretation.

Example 317 (non-compositionality of IO interpretation) *The processes $P_1 \equiv x := 1; x := x + 1$ and $P_2 \equiv x := 2$ are IO-equivalent. However when they are composed in parallel with the process P_2 we have: $[\![P_1 \mid P_2]\!]^{IO} \neq [\![P_2 \mid P_2]\!]^{IO}$.*

20.2 Towards compositionality

As a first attempt at fixing the compositionality issue, we try to refine the semantics of processes. In automata theory, we are used to associate to an automaton the collection of its execution traces. We follow a similar path by considering the traces of the states crossed by a terminating execution.

Definition 318 (trace interpretation) *The trace interpretation of a process P is defined as follows:*

$$[\![P]\!]^T = \{s_1 \ldots, s_n \mid (P, s_1) \xrightarrow{*} (P_2, s_2) \cdots \xrightarrow{*} (P_n, s_n) \downarrow\} .$$

Remark 319 *The IO semantics is exactly the subset of the trace semantics composed of* words of length 2*.*

$$[\![P]\!]^{IO} = \{(s, s') \mid ss' \in [\![P]\!]^T\} .$$

With reference to the previous example 317, it is easy to check that $[\![P_1]\!]^T \neq [\![P_2]\!]^T$. However, for $P_3 \equiv x := 1; x := 2$ we have:

$$[\![P_1]\!]^T = [\![P_3]\!]^T, \qquad [\![P_1 \mid P_2]\!]^T \neq [\![P_3 \mid P_2]\!]^T .$$

So this trace semantics is not compositional either!

While failing to *characterize* the 'right equivalence/pre-order' we can at least *define* it.

Definition 320 (pre-congruences) *A pre-congruence is a pre-order on processes which is preserved by contexts. We define two pre-congruences relatively to the IO and trace interpretations as follows:*

$$\begin{array}{ll} P_1 \leq_{IO} P_2 & \text{if } \forall C \ [\![C[P_1]]\!]^{IO} \subseteq [\![C[P_2]]\!]^{IO} , \\ P_1 \leq_{T} P_2 & \text{if } \forall C \ [\![C[P_1]]\!]^{T} \subseteq [\![C[P_2]]\!]^{T} . \end{array}$$

Exercise 321 *Check that $\leq_{IO}$ ($\leq_T$) is the* largest pre-order *(reflexive and transitive) which refines the IO containment (trace containment) and which is preserved by all contexts.*

Somehow surprisingly, once we require preservation by contexts, it does not matter whether we look at the input-output or at the traces.

Proposition 322 *The pre-congruences $\leq_{IO}$ and $\leq_T$ coincide.*

PROOF. $\leq_T \subseteq \leq_{IO}$. By remark 319, we know that $[\![P_1]\!]^T \subseteq [\![P_2]\!]^T$ implies $[\![P_1]\!]^{IO} \subseteq [\![P_2]\!]^{IO}$. Then it follows *by unfolding* the definitions that:

$$P_1 \leq_T P_2 \text{ implies } P_1 \leq_{IO} P_2 \ .$$

$\leq_{IO} \subseteq \leq_T$. For the other direction, assume *by contradiction* $P_1 \not\leq_T P_2$. This means that for some context C and trace $s_1 \cdots s_n$:

$$s_1 \cdots s_n \in [\![C[P_1]]\!]^T \quad \text{and} \quad s_1 \cdots s_n \notin [\![C[P_2]]\!]^T \ .$$

In particular, this entails, for $Q_1 \equiv C[P_1]$: $(Q_1, s_1) \xrightarrow{*} (Q_1^2, s_2) \xrightarrow{*} \cdots (Q_1^n, s_n)\downarrow$. The *key step* is the following: we build an *observer* O that may terminate iff it sees the state going through $s_1 \cdots s_n$; the observer reads the state without modifying it. Take $X = \mathsf{fv}(C[P_1]) \cup \mathsf{fv}(C[P_2])$ and recall the *IS* predicate from proposition 11:

$$IS(s, X) = \bigwedge_{x \in X} (x = s(x)) \ .$$

Then define:

$$\begin{array}{rcl} O & \equiv & \mathsf{await}\ IS(s_1, X)\ \mathsf{do\ skip}; \\ & & \cdots \\ & & \mathsf{await}\ IS(s_n, X)\ \mathsf{do\ skip}\ . \end{array}$$

We have: $(s_1, s_n) \in [\![C[P_1] \mid O]\!]^{IO}$. On the other hand we claim that:

$$(s_1, s_n) \notin [\![C[P_2] \mid O]\!]^{IO} \ ,$$

because the only way O can terminate is that the state goes through the configurations $s_1, \ldots, s_n$ and since O does not modify the state this would mean $s_1 \cdots s_n \in [\![C[P_2]]\!]^T$. □

Following these preliminary remarks, we can define our *goal* as follows:

find an *interpretation* $[\![\_]\!]$ such that: $[\![P_1]\!] = [\![P_2]\!]$ iff $\forall C\ \ [\![C[P_1]]\!]^{IO} = [\![C[P_2]]\!]^{IO}$.

Such an interpretation (if it exists) will be *compositional* by definition. Sometimes one is happy with the left to right implication. In this case, the interpretation is called *adequate* in that it provides a sufficient criterion to determine the equivalence of two processes. If moreover the right to left implication holds, then one speaks of a fully adequate (or fully abstract) interpretation. Notice that this last property can be reformulated as follows:

$$[\![P_1]\!] \neq [\![P_2]\!] \text{ implies } \exists C\ \ [\![C[P_1]]\!]^{IO} \neq [\![C[P_2]]\!]^{IO} \ .$$

In words, whenever the interpretations differ we can find a context where the IO behaviors, *i.e.*, observable behaviors of the processes differ.

20.3 A trace-environment interpretation

To address the compositionality issue, we are guided by the following intuition:

> to analyze a process in a concurrent system we have to account for the perturbations induced by the environment (the external world).

In particular, in the framework of a trace semantics, we allow the environment (the external world) to modify the state after any sequence of transitions.

Definition 323 (trace-environment interpretation) *Let P be a process. Its trace-environment (TE) interpretation is defined as follows:*

$$\begin{array}{rll} [\![P]\!]^{TE} & = \{ & (s_1, s_1') \cdots (s_n, s_n') \mid \\ & & (P, s_1) \xrightarrow{*} (P_2, s_1') \\ & & \cdots \\ & & (P_n, s_n) \xrightarrow{*} (P_{n+1}, s_n') \downarrow \} \ . \end{array}$$

Exercise 324 *In remark 319, we have observed the equivalence in the trace interpretation of the processes $P_1 \equiv x := 1; x := x + 1$ and $P_2 \equiv x := 1; x := 2$. Check that: $[\![P_1]\!]^{TE} \neq [\![P_2]\!]^{TE}$.*

Remark 325 *An equivalent view of the TE-interpretation is to* add a labelled rewriting rule *that explicitly accounts for the actions of the environment:*

$$\frac{}{(P, s) \xrightarrow{e} (P, s')} \tag{20.1}$$

Thus this labelled rule allows for an arbitrary modification of the state while leaving unchanged the control of the observed process. Then we define:

$$\begin{array}{rll} [\![P]\!]^{TE} & = \{ & s_1, s_1' \cdots s_n, s_n' \mid \\ & & (P, s_1) \xrightarrow{*} (P_2, s_1') \xrightarrow{e} (P_2, s_2) \\ & & \cdots \\ & & (P_{n-1}, s_{n-1}) \xrightarrow{*} (P_n, s_{n-1}') \xrightarrow{e} (P_n, s_n) \\ & & (P_n, s_n) \xrightarrow{*} (P_{n+1}, s_n') \downarrow \} \ . \end{array}$$

The traces in the sense of definition 318 can be regarded as the trace-environment traces where $s_{i+1} = s_i'$, for $i = 1, \ldots, n-1$.

In chapter 21, we shall show that this interpretation is preserved by all the operators of the language. For the time being we just consider the problematic case of parallel composition.

Proposition 326 *The TE-inclusion is preserved by parallel composition.*

Proof. First notice the following properties:

$$\begin{array}{lll} (P_1 \mid P_2) \downarrow & \text{implies} & P_1 \downarrow \text{ and } P_2 \downarrow \ , \\ (P_1 \mid P_2, s) \to (P, s') & \text{implies} & (P_1, s) \to (P_1', s') \text{ and } P \equiv (P_1' \mid P_2) \text{ or} \\ & & (P_2, s) \to (P_2', s') \text{ and } P \equiv (P_1 \mid P_2') \ . \end{array}$$

Thus from a reduction such as:

$$\begin{array}{lcl}(P \mid Q, s_1) & \stackrel{*}{\rightarrow} & (P_2 \mid Q_2, s_1') \\ \cdots & & \\ (P_n \mid Q_n, s_n) & \stackrel{*}{\rightarrow} & (P_{n+1} \mid Q_{n+1}, s_n')\downarrow\end{array}$$

one can extract a reduction for P where all the reduction steps taken by the other process are *simulated by the environment*. As a *concrete example*, suppose $[\![P_1]\!]^{TE} \subseteq [\![P_1']\!]^{TE}$ and

$$(P_1 \mid Q_1, s_1) \rightarrow (P_2 \mid Q_1, s_2) \rightarrow (P_2 \mid Q_2, s_3) \rightarrow (P_3 \mid Q_2, s_4)\downarrow \ .$$

We can turn this into:

$$(P_1, s_1) \rightarrow (P_2, s_2) \stackrel{e}{\rightarrow} (P_2, s_3) \rightarrow (P_3, s_4)\downarrow \ .$$

Then $(s_1, s_2)(s_3, s_4) \in [\![P_1]\!]^{TE} \subseteq [\![P_1']\!]^{TE}$ means:

$$(P_1', s_1) \stackrel{*}{\rightarrow} (P_2', s_2) \stackrel{e}{\rightarrow} (P_2', s_3) \stackrel{*}{\rightarrow} (P_3', s_4)\downarrow \ .$$

Now put back the Q_1 process and let him play the *role of the environment*:

$$(P_1' \mid Q_1, s_1) \stackrel{*}{\rightarrow} (P_2' \mid Q_1, s_2) \rightarrow (P_2' \mid Q_2, s_3) \stackrel{*}{\rightarrow} (P_3' \mid Q_2, s_4)\downarrow \ .$$

This argument can be generalized. Suppose $\alpha = (s_1, s_1') \cdots (s_n, s_n')$, $\alpha \in [\![P_1 \mid Q]\!]^{TE}$, and $[\![P_1]\!]^{TE} \subseteq [\![P_2]\!]^{TE}$. Derive a reduction for P_1 which must also belong to P_2. Then, by putting back the thread Q, conclude that $\alpha \in [\![P_2 \mid Q]\!]^{TE}$. □

Since the TE interpretation refines the IO interpretation, its adequacy will follow by the announced compositionality property shown in chapter 21. We now address the full abstraction problem.

Proposition 327 *Let P_1 and P_2 be processes such that $[\![P_1]\!]^{TE} \not\subseteq [\![P_2]\!]^{TE}$. Then there is a context C such that $[\![C[P_1]]\!]^{IO} \not\subseteq [\![C[P_2]]\!]^{IO}$.*

Proof. Let $\alpha = (s_1, s_1') \cdots (s_n, s_n')$ be a trace-environment sequence such that $\alpha \in [\![P_1]\!]^{TE}$ and $\alpha \notin [\![P_2]\!]^{TE}$. We build an *observer process* O that in a sense *plays the role of the environment* and works as follows:

upon observing s_1' builds s_2 and
$\cdots$
upon observing s_{n-1}' builds s_n and terminates.

Notice that in this case the observer *does modify the state*. Formally, assume $X = \mathsf{fv}(P_1) \cup \mathsf{fv}(P_2)$. The command that *builds a new state* is defined as follows:

$$MAKE_{s,\{x_1,\ldots,x_n\}} = x_1 := s(x_1); \cdots; x_n := s(x_n) \ ,$$

and the *observer process* O is defined by:

$$\begin{array}{lcl}O & \equiv & O_1 \\ O_i & \equiv & \mathsf{await}\ IS(s_i', X)\ \mathsf{do}\ MAKE_{s_{i+1},X}; \\ & & \cdots \\ & & \mathsf{await}\ IS(s_{n-1}', X)\ \mathsf{do}\ MAKE_{s_n,X} \ .\end{array}$$

Then take as *process context* $C = [\,] \mid O$ and let $C_i = [\,] \mid O_i$. We have that $(s_1, s'_n) \in [\![C[P_1]]\!]^{IO}$ because:

$$(C[P_1], s_1) \xrightarrow{*} (C[P_2], s'_1) \xrightarrow{*} (C_2[P_2], s_2)$$
$$\cdots$$
$$(P_n \mid \mathsf{skip}, s_n) \xrightarrow{*} (P'_n \mid \mathsf{skip}, s'_n) \downarrow$$

On the other hand, $(s_1, s'_n) \notin [\![C[P_2]]\!]^{IO}$ because O terminates only if it can observe the states s'_i and build atomically the states s_{i+1} for $i = 1, \ldots, n-1$. And this contradicts the hypothesis that $(s_1, s'_1) \cdots (s_n, s'_n) \notin [\![P_2]\!]^{TE}$. □

20.4 Summary and references

We have described a *trace-environment interpretation* for the $\mathsf{Imp}_{\|}$ language. The interpretation is compositional and abstract. The key point for *compositionality* is that we describe the way both the process and the environment may affect the store (which is what can be observed). The key point for *abstraction* is that $\mathsf{Imp}_{\|}$ can simulate the environment's actions; the await statement is crucial here. The presentation is based on [Bro96].

Chapter 21

A denotational presentation of the trace semantics

The trace-environment interpretation introduced in chapter 20 assigns a meaning (or denotation) to a process which is formally a set of finite sequences of pairs of states. In this chapter, our main task is to show that this meaning can be computed in a *compositional* way in the sense that the denotation of a program phrase can be built out of the denotations of its sub-phrases. Concretely, this amounts to define a *domain* of interpretation, say D, and a collection of functions on D that correspond to the operators of the programming language. For instance, we have to find a function par on D which corresponds to parallel composition and satisfies:

$$[\![P_1 \mid P_2]\!]^{TE} = [\![P_1]\!]^{TE} \ \mathsf{par} \ [\![P_2]\!]^{TE} \ . \tag{21.1}$$

21.1 The interpretation domain

We recall that St is the set of states. As a first step, we notice that the interpretation of a process $[\![P]\!]^{TE}$ belongs to the power-set $L = 2^{(St \times St)^*}$ which when ordered by set-theoretic inclusion is a *complete lattice* (cf. chapter 9).

$$[\![P]\!]^{TE} \in L = 2^{(St \times St)^*} \ . \tag{21.2}$$

Definition 328 (closed set of traces) *We say that $X \in L$ is* closed *if it satisfies the following conditions:*

$$\frac{\alpha\beta \in X}{\alpha(s,s)\beta \in X} \ , \qquad \frac{\alpha(s,s')(s',s'')\beta \in X}{\alpha(s,s'')\beta \in X} \ .$$

These are *a kind of reflexivity and transitivity* properties which are called *stuttering and mumbling*, respectively, in the trace theory jargon. Note that all *process interpretations are closed*.

Definition 329 (closure function) *The* closure function $c : L \to L$ *is defined by:*

$$c(X) = \bigcap \{Y \in L \mid X \subseteq Y, Y \ \textit{closed}\} \ .$$

Thus the function c associates to a set X the least set of closed traces that contains it. We notice the following properties.

Proposition 330 *Let X, Y, X_i vary over L and let c be the closure function. Then:*

1. *If $X \subseteq Y$ then $c(X) \subseteq c(Y)$.*
2. *$c(c(X)) = c(X) \supseteq X$.*
3. *The union of closed sets is closed.*
4. *$c(\bigcup_{i\in I} X_i) = \bigcup_{i\in I} c(X_i)$.*

Proof. We leave properties 1-3 as exercises and consider property 4, The inclusion from left to right follows by monotonicity (property 1). For the reverse inclusion, we know from property 3 that $\bigcup_{i\in I} c(X_i)$ is closed. Thus it suffices to check that: $\bigcup_{i\in I} X_i \subseteq \bigcup_{i\in I} c(X_i)$ which holds since by property 2, $X_i \subseteq c(X_i)$. □

It follows that $(c(L), \subseteq)$ is again a complete lattice where the sup are set-theoretic unions. We take $D = c(L)$ as our *domain of interpretation*.

21.2 The interpretation

First, we define some standard operations on the domain D which are instrumental to the interpretation of $\mathsf{Imp}_{\|}$ processes. The reader may recognize definition patterns found in formal languages.

Skip We define: $\mathbf{Skip} = c(\{(s, s) \mid s \in St\}) \in D$. Notice that this is different from the closure of the empty-set.

Concatenation For $X, Y \in D$ let $X; Y = c(\{\alpha\beta \mid \alpha \in X, \beta \in Y\}) \in D$. Notice that we need to *close* the concatenation of X and Y in the ordinary language-theoretic sense.

Iteration For $X \in D$ let:

$$X^0 = \mathbf{Skip} \in D \ , \quad X^{n+1} = X; X^n \in D \ , \quad X^* = \bigcup_{n\geq 0} X^n \in D \ .$$

By proposition 330(3), there is no need to close the countable union.

Parallel A general *shuffle operation* $\mid$ *on words* can be defined as follows:

$$\begin{aligned} &\epsilon \mid \alpha = \alpha \mid \epsilon = \{\alpha\} \ , \\ &a\alpha \mid b\beta = \{a\gamma \mid \gamma \in (\alpha \mid b\beta)\} \cup \{b\gamma' \mid \gamma' \in (a\alpha \mid \beta)\} \ . \end{aligned}$$

Notice that the shuffle of two words is a *set of words* which is *not* necessarily closed. Then define a *parallel operator* on $X, Y \in D$ as:

$$X \mid Y = \bigcup_{\alpha\in X, \beta\in Y} c(\alpha \mid \beta) \ .$$

Exercise 331 (continuity) *Show that the concatenation, iteration, and parallel operators we have defined on the complete lattice $(D, \subseteq)$ are monotonic and preserve unions (continuity).*

We associate a closed set with a *boolean condition* b (without side effects) as follows:

$$[\![b]\!] \quad = c(\{(s,s) \mid (b,s) \Downarrow \mathsf{true}\}) \ .$$

Intuitively, this is the closed set induced by the set of states satisfying the boolean condition. Then associate a closed set to processes as follows:

$$\begin{array}{ll}
[\![\mathsf{skip}]\!] & = \mathbf{Skip} \\
[\![x := e]\!] & = c(\{(s, s[n/x]) \mid (e,s) \Downarrow n\}) \\
[\![P; P']\!] & = [\![P]\!]; [\![P']\!] \\
[\![\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ P']\!] & = ([\![b]\!]; [\![P]\!]) \cup ([\![\neg b]\!]; [\![P']\!]) \\
[\![\mathsf{while}\ b\ \mathsf{do}\ P]\!] & = ([\![b]\!]; [\![P]\!])^*; [\![\neg b]\!] \\
[\![P \mid P']\!] & = [\![P]\!] \mid [\![P']\!] \\
[\![\mathsf{await}\ b\ \mathsf{do}\ P]\!] & = c(\{(s,s') \mid (s,s) \in [\![b]\!], (s,s') \in [\![P]\!]\}) \ .
\end{array}$$

To force the *atomic* execution of the body of an await statement, we select the traces of length 1 which correspond to the input-output behaviors (remark 325).

The extension to variable declarations requires some work. Given $X \subseteq (St \times St)^*$, x variable, n integer, define:

$$\begin{array}{rcl}
X[x=n] & = & \{(s_1, s_1') \cdots (s_n, s_n') \in X \mid \\
 & & \quad s_1(x) = n, s_{i+1}(x) = s_i'(x), i = 1, \ldots, n-1\}
\end{array}$$

$$\begin{array}{rcl}
X \backslash x & = & \{(s_1[m_1/x], s_1'[m_1/x]) \cdots (s_n[m_n/x], s_n'[m_n/x]) \mid \\
 & & \quad (s_1, s_1') \cdots (s_n, s_n') \in X,\ m_1, \ldots, m_n \in \mathbf{Z}\} \ .
\end{array}$$

The operator $(\_)[x = n]$ fixes the initial value of x to n and makes sure the environment cannot affect the value of x by forcing $s_{i+1}(x) = s_i'(x)$. The operator $(\_)\backslash x$ makes sure that the internal modifications of x are not observable by the environment (the value of the state at x is never modified by a process transition). Then define:

$$[\![\mathsf{var}\ x = n\ P]\!] = c(\ (\ [\![P]\!][x = n]\ )\backslash x\ ) \ .$$

In words, first we select the traces where the initial value of the variable x is n and the environment cannot affect x's value and second we hide to the environment the way x is manipulated.

This concludes the compositional definition of the interpretation. The reader can check that this interpretation does indeed follow the pattern outlined in (21.1). Moreover, it turns out to be equivalent to the operational interpretation.

Proposition 332 (denotational characterisation) *For all processes P, $[\![P]\!]^{TE} = [\![P]\!]$.*

Proof. The proof proceeds by induction on the structure of P. As an example, we show:

$$[\![P; Q]\!] = [\![P; Q]\!]^{TE} \tag{21.3}$$

assuming $[\![P]\!] = [\![P]\!]^{TE}$ and $[\![Q]\!] = [\![Q]\!]^{TE}$.

$[\![P; Q]\!] \subseteq [\![P; Q]\!]^{TE}$. Suppose:

$$\begin{array}{ll}
\alpha & = (s_1, s_1') \cdots (s_n, s_n') \in [\![P]\!] = [\![P]\!]^{TE} \\
\beta & = (t_1, t_1') \cdots (t_m, t_m') \in [\![Q]\!] = [\![Q]\!]^{TE} \ .
\end{array}$$

We have already observed that the operational interpretation of a program is closed. Then by the properties of the closure operator (proposition 330), it suffices to show that $\alpha\beta \in [\![P;Q]\!]^{TE}$. Indeed, we have:

$$\begin{array}{lcl}
(P;Q,s_1) & \stackrel{*}{\to} & (P_1;Q,s_1') \\
\cdots & \stackrel{*}{\to} & \cdots \\
(P_{n-1};Q,s_n) & \stackrel{*}{\to} & (P_n;Q,s_n') \to (Q,s_n') \quad (\text{where: } P_n\downarrow) \\
(Q,t_1) & \stackrel{*}{\to} & (Q_1,t_1') \\
\cdots & \stackrel{*}{\to} & \cdots \\
(Q_{m-1},t_m) & \stackrel{*}{\to} & (Q_m,t_m')\downarrow \ .
\end{array}$$

$[\![P;Q]\!] \supseteq [\![P;Q]\!]^{TE}$. Suppose $\gamma \in [\![P;Q]\!]^{TE}$ is generated as follows:

$$\begin{array}{lcl}
(P;Q,s_1) & \stackrel{*}{\to} & (P_1;Q,s_1') \\
\cdots & \stackrel{*}{\to} & \cdots \\
(P_{n-1};Q,s_n) & \stackrel{*}{\to} & (Q_1,s_n') \\
(Q_1,s_{n+1}) & \stackrel{*}{\to} & (Q_2,s_{n+1}') \\
\cdots & \stackrel{*}{\to} & \cdots \\
(Q_m,s_{n+m}) & \stackrel{*}{\to} & (Q_{m+1},s_{n+m}')\downarrow \ .
\end{array}$$

By the semantics of concatenation, we must also have the following transitions:

$$\begin{array}{lcl}
(P_{n-1},s_n) & \stackrel{*}{\to} & (P_n,s_n'')\downarrow \\
(Q,s_n'') & \stackrel{*}{\to} & (Q_1,s_n') \ .
\end{array}$$

It follows that:

$$\begin{array}{l}
\alpha = (s_1,s_1')\cdots(s_n,s_n'') \in [\![P]\!]^{TE} = [\![P]\!] \ , \\
\beta = (s_n'',s_n')\cdots(s_{n+m},s_{n+m}') \in [\![Q]\!]^{TE} = [\![Q]\!] \ .
\end{array}$$

Then $\alpha\beta \in [\![P]\!];[\![Q]\!]$ and by definition of closure (mumbling), $\gamma \in [\![P;Q]\!]$. □

An immediate corollary is that the trace-environment interpretation is preserved by process contexts.

Corollary 333 *If* $[\![P]\!]^{TE} = [\![P']\!]^{TE}$ *then* $[\![C[P]]\!]^{TE} = [\![C[P']]\!]^{TE}$.

Proof. For instance, if $[\![P]\!]^{TE} = [\![P']\!]^{TE}$ then by proposition 332, $[\![P]\!] = [\![P']\!]$. Thus, for any Q: $[\![P \mid Q]\!]^{TE} = [\![P \mid Q]\!] = [\![P' \mid Q]\!] = [\![P' \mid Q]\!]^{TE}$. □

Another interesting application of the characterization is that it provides an angle to analyze process equivalence.

Exercise 334 *Show that the following processes (in-)equivalences hold in the TE semantics:*

$$\begin{array}{rclcrcl}
\mathsf{skip};P & = & P \ , & & P & = & P;\mathsf{skip} \ , \\
(P;P');P'' & = & P;(P';P'') \ , & & P \mid P' & = & P' \mid P \ , \\
(P \mid P') \mid P'' & = & P \mid (P' \mid P'') \ , & & P \mid \mathsf{skip} & = & P \ , \\
\mathsf{while\ true\ do\ skip} & \leq & P \ , & & \mathsf{while\ true\ do\ skip} & = & \mathsf{await\ false\ do}\ P \ .
\end{array}$$

Thus skip is the unit for both sequential and parallel composition. Further, sequential composition is associative while parallel composition is both associative and commutative. Finally, the diverging computation is the least element of the interpretation.

Exercise 335 (invalid equivalences) *Show that the following equivalences (which hold in the sequential IO semantics) fail in the TE semantics:*

$$\begin{array}{lcllcll} x := y; y := x & = & x := y \ , & & x := y; x := z & = & x := z \qquad (x \neq z) \ , \\ x := y; z := x & = & x := y; z := y \ , & & x := y; z := y & = & z := y; x := y \ . \end{array}$$

Exercise 336 (await **from** atomic) *In section 19.1, we have regarded* atomic(P) *as an abbreviation for* await true do P. *Suppose we regard* await$'$ b do P *as an abbreviation for:*

$$\mathsf{var}\ x = 1\ \mathsf{while}\ x = 1\ \mathsf{do}\ (\mathsf{atomic}(\mathsf{if}\ b\ \mathsf{then}\ (P; x := 0)))$$

where x is a fresh variable. Show that the following equality holds in the considered semantics:

$$\mathsf{await}\ b\ \mathsf{do}\ P = \mathsf{await}'\ b\ \mathsf{do}\ P \ .$$

Exercise 337 (shuffling of infinite words) *Let Σ be an* alphabet *(a non-empty set) with generic elements $a, b, c, \ldots$ If X is a set let X^ω be the set of* infinite words *on X (countable and not finite). If α is a word then α^ω is $\alpha\alpha\cdots$ We denote with $R, S, \ldots$ relations on $D = \Sigma^\omega \times \Sigma^\omega \times \Sigma^\omega$ and write $R(\alpha, \beta, \gamma)$ as an abbreviation for $(\alpha, \beta, \gamma) \in R$. We say that a relation R is* admissible *if:*

$$\begin{array}{ll} R(\alpha, \beta, a\gamma) \text{ implies} & (\ \alpha = a\alpha' \text{ and } R(\alpha', \beta, \gamma)\ ) \text{ or} \\ & (\ \beta = a\beta' \text{ and } R(\alpha, \beta', \gamma)\ )\ . \end{array}$$

We define:

$$\begin{array}{lll} S_0 & = D, & \\ S_{n+1} & = \{(\alpha, \beta, a\gamma) \mid & (\ \alpha = a\alpha' \text{ and } S_n(\alpha', \beta, \gamma)\ ) \text{ or} \\ & & (\ \beta = a\beta' \text{ and } S_n(\alpha, \beta', \gamma)\ )\ \}\ , \\ S_\omega & = \cap_{n<\omega} S_n\ . & \end{array}$$

Problems: (1) Show that there is a largest admissible relation that we denote with Shuffle. (2) Prove or disprove: Shuffle $= S_\omega$. (3) Prove or disprove: (i) Shuffle$(a^\omega, b^\omega, (ab)^\omega)$. (ii) Shuffle$((ab)^\omega, a^\omega, (abb)^\omega)$. (iii) Shuffle$(a^\omega, b^\omega, a^\omega)$.

Exercise 338 (fair schedules and associativity) *A* k-schedule *is a vector $(f_1, \ldots, f_k)$ of k functions on the natural numbers $\mathbf{N}$ such that:*

- *for $j = 1, \ldots, k$ and $n \in \mathbf{N}$: $f_j(n) < f_j(n+1)$ (the functions are strictly growing).*
- *for $i, j \in \{1, \ldots, k\}$ and $i \neq j$: $im(f_i) \cap im(f_j) = \emptyset$ (the ranges of the functions are disjoint).*
- *$\cup_{j=1,\ldots,k} im(f_j) = \mathbf{N}$ (the union of the ranges covers the natural numbers).*

Let Σ be a non-empty set with generic elements $a, b, c, \ldots$ and let Σ^ω be the (countably) infinite words over Σ with generic elements $\alpha, \beta, \ldots$. If $\alpha \in \Sigma^\omega$ and $i \in \mathbf{N}$ then $\alpha[i]$ denotes

the character at position i of the word where we start counting from 0. For instance, if $\alpha = ababab\cdots$ then $\alpha[3] = b$.

If $(f_1, \ldots, f_k)$ is a k-schedule and $\alpha_i \in \Sigma^\omega$ for $i = 1, \ldots, k$ then $M[f_1, \ldots, f_k](\alpha_1, \ldots, \alpha_k)$ is a word whose value at position $i \in \mathbf{N}$ is defined as follows:

$$M[f_1, \ldots, f_k](\alpha_1, \ldots, \alpha_k)[i] = \alpha_j[f_j^{-1}(i)] \qquad \text{if } i \in im(f_j)$$

where $f_j^{-1}(i)$ denotes the (unique!) number that the function f_j maps to i.

1. *Suppose $\alpha = a^\omega$ and $\beta = b^\omega$. (i) Assuming $f_1(i) = 2 \cdot i$ and $f_2(i) = 2 \cdot i + 1$, compute $M[f_1, f_2](\alpha, \beta)$. (ii) Is there a 2-schedule (f, g) such that $M[f, g](\alpha, \beta) = (aab)^\omega = aabaabaab\cdots$? (iii) Is there a 2-schedule (f, g) such that $M[f, g](\alpha, \beta) = ab^\omega = abbbbb\cdots$?*

2. *Suppose (f_1, f_2) and (g_1, g_2) are two 2-schedules. Show that there is a 3-schedule (h_1, h_2, h_3) such that for all words α_i, $i = 1, 2, 3$ we have:*

$$M[g_1, g_2](M[f_1, f_2](\alpha_1, \alpha_2), \alpha_3) = M[h_1, h_2, h_3](\alpha_1, \alpha_2, \alpha_3).$$

3. *Now suppose (h_1, h_2, h_3) is a 3-schedule. Define two 2-schedules (f_1, f_2) and (g_1, g_2) such that for all words α_i, $i = 1, 2, 3$ we have:*

$$M[g_1, g_2](M[f_1, f_2](\alpha_1, \alpha_2), \alpha_3) = M[h_1, h_2, h_3](\alpha_1, \alpha_2, \alpha_3) \ .$$

4. *We define a binary merge operation M that associates a set of words to two words as follows:*

$$M(\alpha, \beta) = \{M[f, g](\alpha, \beta) \mid (f, g) \text{ is a 2-schedule}\} \ .$$

We then extend the operation to sets of words by defining for $X, Y \subseteq \Sigma^\omega$:

$$M(X, Y) = \bigcup_{\alpha \in X, \beta \in Y} M(\alpha, \beta) \ .$$

Show that this merge operation is associative, i.e.*, for all sets of words $X_i \subseteq \Sigma^\omega$, $i = 1, 2, 3$:*

$$M(M(X_1, X_2), X_3) = M(X_1, M(X_2, X_3)) \ .$$

21.3 Summary and references

The trace-environment interpretation can be organized in a denotational style where the meaning of a program (process) is computed by composition of the meaning of its subprograms. This makes manifest the compositionality of the interpretation. We refer the reader to [Bro96] for a variation over the presented semantics which takes into account fairness constraints. This requires working over infinite traces; exercises 337 and 338 go in this direction by defining shuffling operations on infinite words.

Chapter 22

Implementing atomicity

The operational $\mathsf{Imp}_{\|}$ model assumes the possibility of executing atomically a process. A simple implementation strategy could consist in having a global *lock variable* that must be acquired by a process before turning into 'atomic mode' and is released upon termination (cf. example 306). Such a strategy is intuitively inefficient because it limits the degree of parallelism of the computation. This intuition can be supported by a simple numerical argument known as *Amdahl's law*. For instance, the law entails that if 10% of a task has to be executed sequentially while the remaining 90% can be executed in parallel then by allocating 10 processors to the task we can expect a speed up of at most (roughly) 5, *i.e.*, by multiplying the cost of the hardware by 10 we can only divide the computation time by 5 (which is rather disappointing).

In the following we discuss some process transformations that aim at reducing the amount of computation that has to be executed atomically. This should be regarded both as an opportunity to have a glimpse at some basic implementation strategies and as a case study where we practice the operational model.

22.1 An optimistic strategy

In an *optimistic* implementation strategy of an atomic transaction mechanism we run the steps of the transaction concurrently with those of other parallel processes hoping that they will not affect the variables relevant to the transaction. If they do then we start again the transaction. Intuitively, such an approach works well if the chances that two atomic transactions try to modify the same variables at about the same time are low.

In more detail, the transformation can be described as follows. Given a process P, we can statically determine an over-approximation of the visible variables that P may read or write during its execution. For instance, this can correspond to the set $\mathsf{fv}(P)$ of variables occurring free in P. For each variable x let us assume we dispose of fresh variables x_r and x_l. The super-script r and l stand for *read* and *local*, respectively, for reasons that we explain next. Let us write x^* for the list of distinct variables in $\mathsf{fv}(P)$ and let us denote with x_r^* and x_l^* the corresponding lists of fresh variables. Rather than running P atomically we run *non-atomically* a modified process $P' = [x_l^*/x^*]P$ where each read/write operation to the variables x^* is replaced by a reference to the fresh local variables x_l^* which are initialized with the values of x^*. Before running P, we also save the initial value of the variables x^* in the fresh local variables x_r^*. If and when we are done with the execution of P' we check *atomically*

that the current value of x^* equals that of x_r^*. If this is the case, in the same atomic step we write x_l^* in x^* and we conclude successfully the transaction, otherwise we try again. Notice that it may happen that the variables x^* are modified during the computation above. All that matters is that the value of x^* is the same as the value of x_r^* just before writing in x^* the variables x_l^*.

The transformation can be described formally by a function $\mathcal{C}_o$ (o for optimistic) on $\mathsf{Imp}_{\|}$ processes. The key case concerns the await and it is defined as follows assuming $\mathsf{fv}(\mathsf{await}\ b\ \mathsf{do}\ P) = \{x_1, \ldots, x_n\}$, $x^* = x_1, \ldots, x_n$, and using vectorial notations such as $x^* := v^*$ and $x_r^* = x^*$ as an abbreviation for $x_1 := v_1; \cdots; x_n := v_n$ and $x_{r,1} = x_1 \wedge \cdots \wedge x_{r,n} = x_n$, respectively. We also assume that the variables c, x_r^*, x_l^* do not appear free in $\mathsf{await}\ b\ \mathsf{do}\ P$ and that P does not contain await statements.

$$
\begin{array}{rcl}
\mathcal{C}_o(\mathsf{await}\ b\ \mathsf{do}\ P) & = & \mathsf{var}\ c = 1,\ x_r^* = 0^*,\ x_l^* = 0^* \\
& & \mathsf{while}\ c = 1\ \mathsf{do} \\
& & (\ x_r^* := x^*; \\
& & \ \ x_l^* := x_r^*; \\
& & \ \ \mathsf{if}\ [x_l^*/x^*]b\ \mathsf{then} \\
& & \qquad (\ [x_l^*/x^*]P; \\
& & \qquad\ \ \mathsf{atomic}(\mathsf{if}\ x_r^* = x^*\ \mathsf{then}\ x^* := x_l^*; c := 0;)\ )\ )\ .
\end{array}
$$

There are a number of possible variations on this schema. For instance, one can distinguish the variables which are read from those that are written. In another direction, instead of computing an over-approximation of the collection of variables which are affected by the atomic statement, we could determine this set at *run time*. Also, it should be noticed that in the translation above the computation of the process $[x_l^*/x^*]\mathcal{C}_o(P)$ may operate on unexpected states which in more complex programming settings may lead to exceptions or diverging computations. Certain implementations of atomic transactions ensure that the program always operates over consistent states, *i.e.*, states which could actually arise in the reference semantics. The following exercise elaborates on this point.

Exercise 339 *Suppose the* $\mathsf{Imp}_{\|}$ *language is extended with a command* abort *which stops the computation and returns the current state (such command was discussed in exercise 270). Extend the optimistic compilation function so that it handles* abort *commands.*

22.2 A pessimistic strategy

A more pessimistic (or conservative) implementation strategy for atomic transactions consists in gaining control of all the resources relevant to the atomic process before running it. For instance, suppose we associate a *lock variable* ℓ_x with every (shared) variable x. Recall that a *lock variable* is simply a variable that is supposed to be used as a semaphore of capacity 1 (see example 306).

As in the optimistic strategy, given a process P we can statically determine an over-approximation of the variables the process P may read or write during its execution. Let us denote these variables with $x_1, \ldots, x_n$. Then an implementation of $\mathsf{atomic}(P)$ consists in a process that acquires the locks for $x_1, \ldots, x_n$, then runs P, and eventually releases the locks for $x_1, \ldots, x_n$. Such an implementation scheme is known as *two phase locking*: the first phase is the one where the process acquires the locks and the second the one where it

releases them. This locking scheme can be refined by distinguishing between reading and writing accesses. Indeed a *write* access must be exclusive but a *read* access can be shared by an arbitrary number of processes. The function $\mathcal{C}_p$ formalizes this pessimistic transformation on $\mathsf{Imp}_{\|}$ processes. The key cases concern the variable declaration and the atomic statement:

$$\begin{array}{ll} \mathcal{C}_p(\mathsf{var}\ x = v\ \mathsf{in}\ P) & = \mathsf{var}\ x = v, \ell_x = 1\ \mathsf{in}\ \mathcal{C}_p(P) \\ \mathcal{C}_p(\mathsf{atomic}(P)) & = \mathit{lock}(\ell_{x_1}); \cdots; \mathit{lock}(\ell_{x_n}); P; \mathit{unlock}(\ell_{x_1}); \cdots; \mathit{unlock}(\ell_{x_n})\ . \end{array}$$

It should be noticed that parallel processes running a two phase locking protocol may end up in a *deadlock*. For instance, suppose P_1 tries to acquire the locks for x_1 and x_2 while P_2 tries to acquire the locks for x_2 and x_1. We can arrive at a deadlocked configuration where P_1 has acquired the lock for x_1 and P_2 the lock for x_2. In general, one can represent a deadlock associated with locks as a circular waiting situation where all parallel processes which are not properly terminated are waiting to acquire a lock which is currently held by another process.

An approach to *deadlock resolution* consists in introducing a *monitor process* that at appropriate times detects circular waiting and breaks the circle by *aborting* one of the processes. This means that the selected process must release all the acquired locks and start again.

Rather than taking action after the deadlock has happened, another approach consists in *preventing* it. One basic approach that works if the locks can be *totally ordered* consists in acquiring the locks in growing order. A more general approach not requiring a total order consists in introducing an information on the *age* of the atomic transactions. For instance, the so called *wait-die* scheme works as follows. If an older transaction tries to acquire a lock held by a younger transaction then it waits the lock is released, while if a younger transaction tries to acquire a lock held by an older one then it must release all the acquired locks and start again (while keeping its age).

Exercise 340 *Suppose the age of a transaction is a positive natural number. Write pseudo-code for an* acquire *function that takes as input a list of locks and an age and tries to acquire the locks following the wait-die strategy sketched above.*

22.3 A formal analysis of the optimistic strategy

We conclude this section by sketching a formal analysis of the optimistic strategy. With reference to the trace-environment interpretation defined in section 20.3, one would like to show that for any $\mathsf{Imp}_{\|}$ process P, we have $[\![P]\!]^{TE} = [\![\mathcal{C}_o(P)]\!]^{TE}$. We shall approach this problem through the notion of *simulation* which we have already met in chapter 9. Recall that $\downarrow$ is a predicate on programs that defines immediate termination and $\rightarrow$ is a binary relation that defines the small-step reduction of $\mathsf{Imp}_{\|}$. As usual, we denote with $\stackrel{*}{\rightarrow}$ the reflexive and transitive closure of $\rightarrow$.

Definition 341 *A binary relation $\mathcal{R}$ on $\mathsf{Imp}_{\|}$ programs is a* weak simulation *if whenever $P\ \mathcal{R}\ Q$ the following holds for any state s:*

- *if $P \downarrow$ then $(Q, s) \stackrel{*}{\rightarrow} (Q', s)$ and $Q' \downarrow$.*
- *if $(P, s) \rightarrow (P', s')$ then $\exists\, Q'$ ($(Q, s) \stackrel{*}{\rightarrow} (Q', s')$ and $P'\ \mathcal{R}\ Q'$).*

We denote with $\leq$ the union of all weak simulations. The reader may check that this is again a weak simulation. Also we notice the following properties.

Proposition 342 *Let P, Q be $\mathsf{Imp}_{\|}$ processes. Then:*

(1) *If $P \leq Q$ then $[\![P]\!]^{TE} \subseteq [\![Q]\!]^{TE}$.*

(2) *The reverse implication does not hold.*

(3) *If $P \leq Q$ then for any program R we have that $(P \mid R) \leq (Q \mid R)$.*

(4) *If $P \leq Q$ then for any variable x and integer value n we have that $\mathsf{var}\ x = n\ P \leq \mathsf{var}\ x = n\ Q$.*

Proof. (1) Suppose $P \leq Q$ and $\alpha \in [\![P]\!]^{TE}$. We proceed by induction on the length of the trace α to show that $\alpha \in Q$.

$\alpha = (s_1, s_1')$ This means $(P, s_1) \xrightarrow{*} (P_2, s_1')$ and $P_2 \downarrow$. Then by repeatedly applying the second condition defining a simulation we have:

$$(Q, s) \xrightarrow{*} (Q_2, s_1') \text{ and } P_2 \leq Q_2 \ .$$

Also by the first condition $(Q_2, s_1') \xrightarrow{*} (Q_2', s_1')$ and $Q_2' \downarrow$. Thus

$$(Q, s) \xrightarrow{*} (Q_2', s_1') \downarrow$$

which means $\alpha \in [\![Q]\!]^{TE}$.

$\alpha = (s_1, s_1')\alpha', \alpha' \neq \epsilon$ This means $(P, s_1) \xrightarrow{*} (P_2, s_1')$ and $\alpha' \in [\![P_2]\!]^{TE}$. Then

$$(Q, s_1) \xrightarrow{*} (Q_2, s_1') \text{ and } P_2 \leq Q_2 \ .$$

By inductive hypothesis, $\alpha' \in [\![Q_2]\!]^{TE}$. It follows $\alpha \in [\![Q]\!]^{TE}$.

(2) Recall that a non-deterministic sum can be defined in $\mathsf{Imp}_{\|}$. Then we consider:

$$\begin{array}{rcl} P & \equiv & (a = 0) \to a := 1; [(b = 0) \to b := 1 + (c = 0) \to c := 1] \\ Q & \equiv & [(a = 0) \to a := 1; [(b = 0) \to b := 1] \ + \ (a = 0) \to a := 1; [(c = 0) \to c := 1]] \ . \end{array}$$

It is intended that only the first assignment after the test-for-zero is executed atomically. Thus for instance:

$$(P, [0/a]s) \to\to ([(b = 0) \to b := 1 + (c = 0) \to c := 1],\ [1/a]s)$$

Moreover notice that:

$$([(b = 0) \to b := 1 + (c = 0) \to c := 1],\ [0/b, 0/c]s) \to (P', [1/b, 0/c]s), (P'', [0/b, 1/c]s) \ ,$$

where $P', P'' \downarrow$. On the other hand, Q cannot simulate the first step of P. If it takes the first branch it cannot modify c and if it takes the second it cannot modify b. Another possibility is to notice that in the trace-environment semantics all looping programs are interpreted as the empty set while the weak simulation semantics may distinguish two looping programs such as $P = \mathsf{while\ true\ do}\ x := 1$ and $Q = \mathsf{while\ true\ do\ skip}$. Indeed, we have $P \not\leq Q$: the move $(P, [0/x]s) \to (P, [1/x]s)$ cannot be matched by Q.

(3) We show that the following relation $\mathcal{R}$ is a weak simulation:

$$\mathcal{R} = \leq \cup \{(P \mid R, Q \mid R) \mid P \leq Q, R \text{ program}\} \ .$$

Suppose $P \leq Q$ and $(P \mid R, s) \to (P' \mid R', s')$. We analyze the two possible cases:

$(P,s) \to (P',s'), R' = R$ Then $(Q,s) \xrightarrow{*} (Q',s')$ and $P' \leq Q'$. So $((Q \mid R),s) \xrightarrow{*} ((Q' \mid R),s')$ and $(P' \mid R)\ \mathcal{R}\ (Q' \mid R)$.

$(R,s) \to (R',s'), P' = P$ Then $((Q \mid R),s) \to ((Q \mid R'),s')$ and $(P \mid R')\ \mathcal{R}\ (Q \mid R')$.

(4) We show that the following relation $\mathcal{R}$ is a weak simulation:

$$\mathcal{R} = \leq \cup \{(\mathsf{var}\ x = n\ P, \mathsf{var}\ x = n\ Q \mid P \leq Q, n \in \mathbf{Z}\}$$

Suppose $P \leq Q$ and $(\mathsf{var}\ x = n\ P, s) \to (\mathsf{var}\ x = m\ P', s'[s(x)/x])$ because $(P, s[n/x]) \to (P', s'[m/x])$. Then $(Q, s[n/x]) \xrightarrow{*} (Q', s'[m/x])$ and $P' \leq Q'$. Thus $(\mathsf{var}\ x = n\ Q, s) \xrightarrow{*} (\mathsf{var}\ x = m\ Q', s'[s(x)/x])$ and $(\mathsf{var}\ x = m\ P')\ \mathcal{R}\ (\mathsf{var}\ x = m\ Q')$. □

Exercise 343 *Show that if $P_i \leq Q_i$ for $i = 1, 2$ then $P_1; P_2 \leq Q_1; Q_2$.*

Proof techniques for simulation (and bisimulation) are developed in the more abstract setting of labelled transition systems in chapter 24. For the time being, we recall from chapter 9 that to show that $P \leq Q$ it suffices to exhibit a relation $\mathcal{R}$ which contains the pair (P, Q) and which is a weak simulation. As an application of this technique, let us show the following.

Proposition 344 *Let P be a $\mathsf{Imp}_{\|}$ process then $P \leq \mathcal{C}_o(P)$.*

Proof. We consider the relation:

$$\mathcal{R} = \quad \{(P, \mathcal{C}_o(P)) \mid P\ \mathsf{Imp}_{\|}\ \text{process}\} \cup \{(P,Q) \mid P\downarrow, Q\downarrow\}\ .$$

We have to check that whenever $(P,Q) \in \mathcal{R}$ then the two conditions specified in the definition 341 above hold. For the first condition, we check that if $P\downarrow$ then $\mathcal{C}_o(P)\downarrow$ by induction on the definition of immediate termination. For the second condition, we proceed by induction on the reduction $(P,s) \to (P',s')$ according to the rules specified in table 19.2 of chapter 19. The only interesting case is when $P \equiv \mathsf{await}\ b\ \mathsf{do}\ P_1$, $(b,s) \Downarrow \mathsf{true}$ and $(P_1,s) \xrightarrow{*} (P',s')$ with $P'\downarrow$. First, we need a lemma that relates the reductions of (P_1,s) to those of $([x_l^*/x^*]P_1, s[s(x)^*/x_l^*])$. Then one exhibits a sequence of reductions such that $(\mathcal{C}_o(P),s) \xrightarrow{*} (Q,s')$ and $Q'\downarrow$. Now in general it is not true that $Q \equiv \mathcal{C}_o(P')$, and this is precisely the reason we enlarged the definition of $\mathcal{R}$ to include all the pairs of immediately terminated processes. □

It follows from propositions 342 and 344 that $[\![P]\!]^{TE} \subseteq [\![\mathcal{C}_o(P)]\!]^{TE}$. For the sake of simplicity, we discuss the reverse inclusion in a simplified case and leave the generalization to the reader.

Proposition 345 *Suppose $P \equiv \mathsf{await}\ b\ \mathsf{do}\ P_1$ is a $\mathsf{Imp}_{\|}$ process and P_1 does* not *contain parallel composition,* while *and* await *statements. Then $\mathcal{C}_o(P) \leq P$.*

Proof. For any state s, the reduction of (P_1,s) is deterministic and terminates. We write $(P_1,s) \Downarrow s'$ if $(P_1,s) \xrightarrow{*} (\mathsf{skip}, s')$. We also write $P \Downarrow$ if for all states s, all reductions starting from (P,s) terminate and do not modify the state s.
Preliminary remark. For all states s, s' if $(\mathcal{C}_o(P),s) \xrightarrow{*} (P',s')$ then $P' \equiv \mathsf{var}\ c = n\ P''$ and $n \in \{0,1\}$. Moreover:

- If $n = 1$ then $s' = s$.
- If $n = 0$ then $P' \Downarrow$, $(b, s) \Downarrow \mathsf{true}$, and $(P_1, s) \Downarrow s'$.

To show this, we notice that a residual of $(\mathcal{C}_o(P), s)$ sets to 0 the variable c if and only if:

1. The boolean condition $[x_l^*/x^*]b$ evaluates to true. Notice that when this happens the contents of x_l^* and x_r^* are identical.
2. The boolean condition $x_r^* = x^*$ in the atomic statement evaluates to true.

This means that when c is set to 0, we know that the current state s is such that $(b, s) \Downarrow \mathsf{true}$ and $(P_1, s) \Downarrow s'$. As long as the two conditions above are not satisfied a residual of $\mathcal{C}_o(P)$ has the shape:

$$\mathsf{var}\ c = 1\ \cdots; \mathsf{while}\ c = 1\ \cdots$$

and cannot modify the visible state. Next, we define a relation $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ as follows:

$$\begin{aligned} \mathcal{R}_1 &= \{(Q, P) \mid \exists\, s\ \ (\mathcal{C}_o(P), s) \xrightarrow{*} (\mathsf{var}\ c = 1\ P', s)\} \\ \mathcal{R}_2 &= \{(Q, \mathsf{skip}) \mid \exists\, s, s'\ \ (\mathcal{C}_o(P), s) \xrightarrow{*} (\mathsf{var}\ c = 0\ P', s')\} \end{aligned}$$

We conclude by checking that the relation $\mathcal{R}$ is a simulation. For $\mathcal{R}_1$ we use Property 1. As long as the reduction does not affect c, there is no modification of the visible state and no immediate termination. So we can just simulate by doing nothing. As soon as the transition set c to 0 we simulate by reducing atomically P. For $\mathcal{R}_2$, it suffices to check that:

$$\{(Q, \mathsf{skip}) \mid Q \Downarrow\}$$

is a simulation. □

22.4 Summary and references

Atomicity is a major issue in concurrency theory starting from early work on the implementation of atomic transactions in databases [Pap79, BHG87, LMWF94]. Later, related concepts have been developed in the framework of concurrent programming. In particular, let us mention the notion of *concurrent object* and *linearizability* [HW90] and the related results that classify the synchronization power of various concurrent objects [Her91] (see also chapter 32). Nowadays, the various strategies to implement atomicity we have discussed are applied to standard programming languages (*C++*, *Java*, *Haskell*, *ML*,...). In particular, the work on so called hardware/software transactional memories [HM93, ST95] is mainly concerned with the problem of finding an efficient implementation of the atomic operator. Amdahl's law is presented in [Amd67]. An early and quite readable description of the optimistic strategy in the framework of database systems can be found in [KR81].

Chapter 23

Rely-guarantee reasoning

We have seen in chapter 20 that the *semantics* of concurrent processes calls for new techniques. Not surprisingly, a similar and related phenomenon arises in the *specification* of concurrent processes. In chapter 1, we have introduced the notion of partial correctness assertion (pca). Table 1.4 gives the rules to reason on a sequential fragment of the $\mathsf{Imp}_{\|}$ language. These rules are *sound* (proposition 1.4) and can be *inverted* (proposition 9) thus providing a syntax-directed method to reduce a pca to an ordinary logical statement. Is it possible to extend these results to the $\mathsf{Imp}_{\|}$ language?

23.1 Rely-guarantee assertions

We recall and extend some of the notation introduced in section 1.2 to reason on pca. We associate with a program P the input-output relation on states:

$$(s, s') \in [\![P]\!]^{IO} \text{ if } (P, s) \Downarrow s' \ .$$

For example, in the case P is an assignment $x := e$, we have:

$$[\![P]\!]^{IO} = \{(s, s[v/x]) \mid (e, s) \Downarrow v\} \ ,$$

which turns out to be (the graph of) a total function.

In the assertions, we identify a boolean predicate b with the set of states that satisfy it, thus b stands for $\{s \mid s \models b\}$. We denote the set of states with St, unary relations on St with $A, B, \ldots$ and binary relations on St with $R, G, \ldots$. To manipulate relations on states, we use the following notation:

Id	$= \{(s, s) \mid s \in St\}$	(identity relation)
Top	$= \{(s, s') \mid s, s' \in St\}$	(top relation)
$A; R$	$= \{s' \mid \exists\, s \ \ s \in A \text{ and } (s, s') \in R\}$	(image)
$R; A$	$= \{s \mid \exists\, s' \ \ s' \in A \text{ and } (s, s') \in R\}$	(pre-image)
$R; R'$	$= \{(s, s'') \mid \exists\, s' \ \ (s, s') \in R \text{ and } (s', s'') \in R'\}$	(composition).

As usual, if R is a relation then R^* is its reflexive and transitive closure.

As mentioned above, the generation of the logical conditions follows the structure of the program (proposition 9). One would like to follow this pattern for the concurrent programs of the $\mathsf{Imp}_{\|}$ language too. The following exercise suggests we can handle local variables.

Exercise 346 (rule for local variables) *We write:*

$$\begin{array}{ll} A[n/x] & = \{s[n/x] \mid s \in A\} \ , \\ s =_X s' & \textit{if } \ \forall x \in X \ \ s(x) = s'(x) \ , \\ x \notin \mathsf{fv}(B) & \textit{if } \ \forall s \ \ (s \in B, \ s =_{\{x\}^c} s' \ \supset \ s' \in B) \ , \\ \exists x.B & = \{s \mid \exists\, s' \in B \ \ s =_{\{x\}^c} s'\} \ . \end{array}$$

Then we formulate the rule for local variables as follows:

$$\frac{\{A[n/x]\} \ P \ \{\exists x.B\} \quad x \notin \mathsf{fv}(B)}{\{A\} \ \mathsf{var} \ x = n \ P \ \{B\}} \ .$$

Show that the rule is sound and that it can be inverted modulo α-renaming of the bound variable provided $\{y \mid y \in \mathsf{fv}(B)\}$ is finite (which is always the case if the set B is defined by a formula).

Then let us turn to parallel composition, which is the core of the matter, and let us try to formulate a rule of the shape:

$$\frac{\{A_i\} \ P_i \ \{B_i\} \quad i = 1, 2}{\{f(A_1, A_2)\} \ P_1 \mid P_2 \ \{g(B_1, B_2)\}} \ ,$$

where f and g are two ways of combining predicates. Take: $P_1 \equiv x := 1; x := x + 1$ and $P_2 \equiv x := 2$. We already know from example 317 that:

$$[\![P_1]\!]^{pca} = [\![P_2]\!]^{pca} \quad \text{and} \quad [\![P_1 \mid P_2]\!]^{pca} \neq [\![P_2 \mid P_2]\!]^{pca} \ .$$

In particular, this means that any derivation that would end with a proof of the shape:

$$\frac{\{A_1\} \ P_2 \ \{B_1\} \quad \{A_2\} \ P_2 \ \{B_2\}}{\{\mathsf{true}\} \ P_2 \mid P_2 \ \{x \leq 2\}} \ ,$$

where $\models g(B_1, B_2) \subseteq \{s \mid s(x) \leq 2\}$ could be turned into a derivation of the triple $\{\mathsf{true}\} \ P_1 \mid P_2 \ \{x \leq 2\}$ which is obviously *not* valid! An early approach to this problem goes back to Owicki and Gries. Their rule has the shape:

$$\frac{\{A_1\} \ P_1 \ \{B_1\} \quad \{A_2\} \ P_2 \ \{B_2\}}{\{A_1 \cap A_2\} \ P_1 \mid P_2 \ \{B_1 \cap B_2\}} \tag{23.1}$$

provided the proofs of the premises do not 'interfere'. Having to look at the internal structure of processes is not very satisfying and it is clearly at odd with one basic principle of module composition: to compose proofs (modules) one should just know what is proved (the interface) without depending on the details of the proof (the implementation). A way to tackle these limitations is to consider a richer specification language whose judgments have the shape:

$$P : (A, R, G, B) \tag{23.2}$$

where: (1) A and B are a pre-condition and a post-condition, respectively as in Floyd-Hoare rules, hence sets of states, (2) R is a *relation on states* that describes the environment transitions that are admitted (thus R is part of the pre-conditions), and (3) G is *relation on states* that describes the program transitions that are guaranteed (thus G is part of the

post-condition). We refer to assertions of the shape (23.2) as *rely-guarantee assertions*, or rga for short. To define their validity, we recall from chapter20 that $\mathsf{Imp}_{\|}$ programs may perform the following *labelled transitions*:

$$\begin{array}{ll} (P,s) \xrightarrow{p} (P',s') & \text{(program transition, cf. Table 19.2)} \\ (P,s) \xrightarrow{e} (P,s') & \text{(environment transition, cf. rule (20.1)).} \end{array}$$

Definition 347 (computation) *A* computation *of a program P is a (finite or infinite) sequence:*

$$(P,s_0) \xrightarrow{\lambda_0} (P_1,s_1) \xrightarrow{\lambda_1} (P_2,s_2) \xrightarrow{\lambda_2} \cdots \tag{23.3}$$

where s_i are states and $\lambda_i \in \{p,e\}$.

Definition 348 (validity rely-guarantee) *The rely-guarantee assertion $P : (A,R,G,B)$ is valid if* for all computations *of P of the shape (23.3) such that the following* pre-condition *holds:*

$$s_0 \in A \wedge \forall\, i \;\; (\lambda_i = e \supset (s_i, s_{i+1}) \in R) \;,$$

it follows that the following post-condition *holds:*

$$\forall\, i \;\; (P_i \downarrow\!\supset s_i \in B) \wedge (\lambda_i = p \supset (s_i, s_{i+1}) \in G) \;.$$

Thus the pre-condition concerns the initial configuration and all transitions performed by the environment (including those after termination) while the post-condition concerns the final configurations (if any) and all the transitions performed by the program.

Exercise 349 *The validity of a given pca is equivalent to the validity of a derived rga. Specifically, show that the pca $\{A\}\ P\ \{B\}$ is valid iff the rga $P : (A, Id, Top, B)$ is valid.*

Remark 350 *Rga's can discriminate programs which are trace-environment equivalent. For instance, consider the programs* $\mathsf{loop} \equiv \mathsf{while\ true\ do\ skip}$ *and $P \equiv x := 0; \mathsf{loop}$. We know that $[\![\mathsf{loop}]\!]^{TE} = [\![P]\!]^{TE}$, since all the diverging computations receive the empty interpretation. On the other hand, the rga* $(\mathsf{true}, Id, Id, \mathsf{false})$ *is satisfied by* loop *but not by P because the assignment $x := 0$ does not respect the guarantee condition Id.*

Definition 351 (stability) *Let A be a unary relation and R a binary relation on some set. Then we write $\mathcal{S}(A,R)$ if $s \in A$ and $(s,s') \in R$ implies $s' \in A$ and say that A is stable with respect to R.*

Proposition 352 *Let A, B be unary relations and R be a binary relation. Then:*

1. *$\mathcal{S}(A,R)$ iff $A; R^* \subseteq A$.*
2. *$A; R^* \subseteq B$ iff $\exists\, A' \;\; (A' \supseteq A, \mathcal{S}(A',R), A' \subseteq B)$.*

Table 23.1 provides a collection of rules to derive valid rely-guarantee assertions, with the proviso that the guarantee relation G in the conclusion of the rules contains the identity relation Id. Without this hypothesis, the rules are unsound. For instance, one can derive $\mathsf{skip; skip} : (\mathsf{true}, Top, \emptyset, \mathsf{true})$ from $\mathsf{skip} : (\mathsf{true}, Top, \emptyset, \mathsf{true})$.

$$\frac{A \subseteq A' \quad R \subseteq R' \quad B' \subseteq B \quad G' \subseteq G \quad P : (A', R', G', B')}{P : (A, R, G, B)} \qquad \frac{P : (A, R, G, C) \quad Q : (C, R, G, B)}{P; Q : (A, R, G, B)}$$

$$\frac{P : (A \cap b, R, G, B) \quad Q : (A \cap \neg b, R, G, B)}{\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q : (A, R, G, B)} \qquad \frac{A \subseteq B \quad \mathcal{S}(A, R)}{\mathsf{skip} : (A, R, G, B)}$$

$$\frac{\mathcal{S}(A, R) \quad \mathcal{S}(B, R) \quad A; [\![x := e]\!]^{IO} \subseteq B \quad [\![x := e]\!]^{IO} \subseteq G}{x := e : (A, R, G, B)} \qquad \frac{\mathcal{S}(A, R) \quad \mathcal{S}(B, R) \quad (A \cap \neg b) \subseteq B \quad P : (A \cap b, R, G, A)}{\mathsf{while}\ b\ \mathsf{do}\ P : (A, R, G, B)}$$

$$\frac{\mathcal{S}(A, R) \quad \mathcal{S}(B, R) \quad b; [\![P]\!]^{IO} \subseteq G \quad P : (A \cap b, Id, Top, B)}{\mathsf{await}\ b\ \mathsf{do}\ P : (A, R, G, B)} \qquad \frac{(R \cup G_1) \subseteq R_2 \quad (R \cup G_2) \subseteq R_1 \quad (G_1 \cup G_2) \subseteq G \quad (B_1 \cap B_2) \subseteq B \quad P : (A, R_1, G_1, B_1) \quad Q : (A, R_2, G_2, B_2)}{P \mid Q : (A, R, G, B)}\ .$$

Table 23.1: Rely-guarantee rules for $\mathsf{Imp}_{\|}$

Proposition 353 (soundness rga rules) *The rely-guarantee assertions derivable in the system in Table 23.1 are valid.*

Proof. We present the argument for the rule for parallel composition. A computation of the shape:

$$(P \mid Q, s_0) \xrightarrow{\lambda_0} (P_1 \mid Q_1, s_1) \xrightarrow{\lambda_1} \cdots$$

can be turned into a computation of P where the moves played by Q are actually attributed to the environment:

$$(P, s_0) \xrightarrow{\lambda'_0} (P_1, s_1) \xrightarrow{\lambda'_1} \cdots \tag{23.4}$$

with $\lambda'_i = e$ if $\lambda_i = e$ or $\lambda_i = p$ and the i^{th} move is due to Q, and $\lambda'_i = p$ otherwise. By a symmetric argument, we derive a computation for Q too:

$$(Q, s_0) \xrightarrow{\lambda''_0} (Q_1, s_1) \xrightarrow{\lambda''_1} \cdots \tag{23.5}$$

We claim that:

$$\begin{array}{lll} \lambda'_i = p & \text{implies} & (s_i, s_{i+1}) \in G_1 \\ \lambda''_i = p & \text{implies} & (s_i, s_{i+1}) \in G_2\ . \end{array}$$

We reason by contradiction and take j to be least natural number where the property above fails. For instance, suppose: $\lambda'_j = p$ and $(s_j, s_{j+1}) \notin G_1$. Now for a transition λ'_i with $i < j$ we have 3 cases: either $\lambda'_i = p$ and $(s_i, s_{i+1}) \in G_1$, or $\lambda_i = e$, $\lambda'_i = e$ and $(s_i, s_{i+1}) \in R$, or $\lambda_i = p$, $\lambda''_i = p$ and $(s_i, s_{i+1}) \in G_2$ because $i < j$. In particular, if $\lambda'_i = e$ we have that $(s_i, s_{i+1}) \in R \cup G_2 \subseteq R_1$. Then the following computation of P:

$$(P, s_0) \xrightarrow{\lambda'_0} \cdots (P_j, s_j) \xrightarrow{p} (P_{j+1}, s_{j+1})$$

contradicts the hypothesis $P : (A, R_1, G_1, B)$. It follows that if $\lambda_i = p$ then $(s_i, s_{i+1}) \in G_1 \cup G_2 \subseteq G$. Finally, we notice that $(P_i \mid Q_i)\downarrow$ entails $P_i\downarrow$ and $Q_i\downarrow$. Thus $s_i \in B_1 \cap B_2 \subseteq B$. □

Exercise 354 *Prove the pca* $\{x = 0\}\ x := 1; x := x + 1 \mid x := 2\ \{x \in \{2, 3\}\}$ *using the rely-guarantee system.*

Unfortunately, the move from pca's to rga's is not quite sufficient to reason about concurrent programs. For instance, it is problematic to prove simple assertions such as:

$$\{x = 0\}\ x := x + 1 \mid x := x + 1\ \{x = 2\}\ . \tag{23.6}$$

The problem is that in the rule for parallel composition we need to *abstract* the possible state transformations of the parallel processes into a relation on states. In doing this, we lose information. For instance, we can record the fact that the variable x can be incremented by one, but we lose the information that this state transition can occur *at most once*. Further, in slightly more complicated programs such as $x := x + 1; x := x + 2$, one loses information on the *order* of the state transformations too.

A 'solution' which goes back to the Owicki-Gries system is to allow for an *instrumentation* of the program. This means enriching the program with auxiliary variables and assignments which allow to record (essential parts of) the history of the computation without affecting it. The 'auxiliary variable' rule states that if we can prove a rely-guarantee assertion of the instrumented program then we can transfer this property to the original program where the instrumentation is removed. For instance, with reference to the pca (23.6) above we can prove:

$$\{x = 0, p = 0, q = 0\}\ \mathsf{atomic}(x := x + 1; p := 1) \mid \mathsf{atomic}(x := x + 1; q := 1)\ \{x = 2\} \tag{23.7}$$

and then derive the desired pca (23.6) by erasing the auxiliary variables p and q along with the related assignments and atomic statements. In practice, one can insert *program counters* in all processes and describe exactly in the assertions the way the computation progresses.

It turns out that when adding an *auxiliary variable rule*, it is possible to invert the rules presented in Table 23.1 along the spirit of proposition 9. However, as for the non-interference rule (23.1), this is not very satisfying and really points to a weakness of the specification language. A more powerful and flexible approach consists in building a full fledged modal logic that allows to describe the transitions of the program and the environment. Examples of such modal logics are discussed later in chapter 25 in the more abstract framework of labelled transition systems.

23.2 A coarse grained concurrent garbage collector

We suppose the reader is familiar with the idea that a program may need to allocate memory at run time and that such memory should be collected and reused whenever possible so that the program can carry on its execution within certain given memory bounds.

In general, it is hard to predict when a memory block becomes useless to the rest of the computation and can be collected. One approach to this issue consists in designing a specific program called *garbage collector* which periodically analyzes the state of the memory and collects the blocks which are useless. Specifically, the memory is modelled as a directed graph

with a collection of root nodes which are the entry points of the program to the memory. All nodes which are not *accessible* from the roots are considered as *garbage* and can be collected. Thus, at least from a logical point of view, the activity of the garbage collector can be decomposed in two phases: a *marking* phase where the accessible nodes are determined and a *collecting* phase where the inaccessible nodes become again available for future usage. In practice, it is convenient to relax a bit this specification. Namely, one determines an *over-approximation* of the accessible nodes and consequently one collects a *subset* of the inaccessible ones.

Going towards a formalization, let us assume a fixed set of nodes N and a fixed subset of roots $\mathit{Roots} \subseteq N$. We also use $E \subseteq N \times N$ to denote the collection of directed edges which varies over time. For any given collection of edges E, we have a collection of nodes which are accessible from the roots: $\mathit{Acc}(E) = \mathit{Roots}; E^*$.

The activity of the program, henceforth called *Mutator*, on the memory graph can be summarized as follows: it selects two accessible nodes i, j and redirects an outgoing edge of the first node, say (i, k), towards j. In other terms, the edge (i, k) is replaced by the edge (i, j), where i, j, k are not necessarily distinct. The fundamental property which is guaranteed by the *Mutator* is that the collection of accessible nodes can only decrease. Thus if we denote with E and E' the collection of edges before and after a *Mutator*'s action we have that:

$$\mathit{Acc}(E') \subseteq \mathit{Acc}(E) \ . \tag{23.8}$$

This representation of the *Mutator* seems very simple but it is actually reasonable provided we assume that among the root nodes there is: (i) a special node called nil without outgoing edges and (ii) a special node called fl (free list) that points to the list of free nodes that can be used to allocate memory. With these hypotheses, expected operations of the *Mutator* such as setting a pointer to nil or redirecting a pointer towards a newly allocated node, fall within the scope of the model.

The activity of the garbage collector is a bit more complex. The task of the marking phase is to determine a set $M \subseteq N$ which over-approximates the collection of reachable nodes. A natural way to approach the task is to compute iteratively the least fixed point of the function associating to a set of nodes M the set of nodes $\mathit{Roots} \cup M; E$. This can be expressed in an imperative programming notation as follows:

$$\begin{array}{ll} \mathit{Marker} \equiv & M := \emptyset;\ M_1 := \mathit{Roots}; \\ & \mathsf{while}\ M_1 \not\subseteq M\ \mathsf{do} \\ & (\ M := (M \cup M_1);\ M_1 := (M; E);\ ) \end{array}$$

The *Marker* program satisfies the pca:

$$\{\mathsf{true}\}\ \mathit{Marker}\ \{(\mathit{Roots} \subseteq M)\ \wedge\ (M; E \subseteq M)\}\ . \tag{23.9}$$

It follows by induction that: $\mathit{Acc}(E) \subseteq M$. Once the marking phase is completed, the collecting phase consists in inserting all the nodes which are not in the set M in the *free list* pointed by fl.

Our goal in the following is to reason on the properties of the *Mutator* and *Marker* when they are run in parallel. The property we want to check is that once the *Marker* terminates the set M is indeed an over-approximation of the set $\mathit{Acc}(E)$. In order to express the specification we introduce an auxiliary variable *done* which is initially set to false and becomes true when *Marker* terminates.

Formally, we regard a *memory state* s of our parallel program as a function from the variables E, M, M_1, *done* to the appropriate value domains. When defining a relation R on memory states we say that a variable x is stable if $(s, s') \in R$ implies $s(x) = s'(x)$. We will also say that $Acc(E)$ decreases if $(s, s') \in R$ implies $Acc(s(E)) \supseteq Acc(s'(E))$. We define:

$$Marker' \equiv (done := \mathsf{false};\ Marker;\ done := \mathsf{true})\ .$$

Then we want to show that:

$$Mutator \mid Marker' : (\emptyset, \emptyset, G, \mathsf{true}) \tag{23.10}$$

where $G = G_1 \cup G_2$ and:

$$\begin{array}{ll} G_1 & = \{(s, s') \mid M, M_1, done \text{ stable}, Acc(E) \text{ decreasing}\} \\ G_2 & = \{(s, s') \mid E \text{ stable}, \\ & \qquad s(done) = \mathsf{true} \text{ implies } (Acc(s(E)) \subseteq s(M),\ M, M_1, done \text{ stable}\}\ . \end{array}$$

By the rule for parallel composition in Table 23.1, the assertion (23.10) is reduced to:

$$\begin{array}{ll} (1) & Mutator : (\emptyset, R_1, G_1, \mathsf{true}) \\ (2) & Marker' : (\emptyset, R_2, G_2, \mathsf{true}) \\ (3) & R_1 = \{(s, s') \mid E \text{ stable}\} \\ (4) & R_2 = G_1\ . \end{array}$$

The related inclusions are easily checked.

Exercise 355 *Apply the rules in Table 23.1 to deduce the property (2) above.*

Having outlined a formal analysis of the garbage collector, let us reconsider our description of the *Marker* to notice that we are assuming that the computation of the set $M; E$ of nodes reachable in one step from the set M is performed *atomically*. This is a potentially long operation and one would like to split it in smaller pieces. The difficulty that arises in this case is that while visiting the nodes in M the set E may be modified by the *Mutator* in a non-monotonic way. For instance, we can have $Roots = \{i, j\}$ and the collection of edges oscillating between $E_1 = \{(i, k)\}$ and $E_2 = \{(j, k)\}$. If the *Marker* visits i (j) while the collection of edges is E_2 (E_1, respectively) then it will never notice that the node k is accessible! For this reason, it is usually assumed that in finer grained concurrent garbage collectors the mutator must *help* the marker. At our abstract level, that could mean that the mutator is also allowed to add elements to the set M.

23.3 Summary and references

Rely-guarantee assertions to reason about concurrent programs are based on both unary and binary relations on states. The latter describe the transitions of the environment we can rely upon and the transitions of the program that are guaranteed. Owicki-Gries system is presented in [OG76]. Rely-guarantee assertions for reasoning about $\mathsf{Imp}_{\|}$ programs were put forward in [Jon83] and then developed by Stirling [Sti88]. The presentation above is close to [Nie03] which in turn is based on [XdRH97].

The compact modeling of the garbage collector is introduced in [DLM+78]. More refined solutions to the concurrent garbage collection problem are described and analyzed, *e.g.*, in [DLM+78, Gri77, BA84, vdS87]. The number of proof obligations to be checked for fine grained garbage collectors becomes quickly overwhelming and a machine-assisted proof is instrumental to raise the confidence in the proposed solution. Examples of such developments can be found in [NE00, DG94]. Also it seems useful to cast the problem of concurrent garbage collection in the more general framework of lock-free concurrent data structures [HM92] which is discussed in chapter 32.

Chapter 24

Labelled transition systems and bisimulation

So far we have considered a rather concrete model of concurrent computation, namely parallel imperative programs with shared variables. To analyze the design spectrum which is available in the semantics of concurrency, it is convenient to move to a more abstract framework where systems perform some set of *actions*. Such a system is called *labelled transition system* (lts). Concretely, an action could consist in changing the contents of a shared variable or sending a message. In this chapter, we formalize the notion of (bi-)simulation over a lts, consider a way to abstract away internal computation steps (*weak* (bi-)simulation), and present some proof techniques for (bi-)simulation.

24.1 Labelled transition systems

A labelled transition system (lts) can be regarded as an *automaton* where we do not specify the set of initial and final states.

Definition 356 (labelled transition system) *A* labelled transition system *is a ternary relation $\rightarrow$ such that $\rightarrow \subseteq S \times Act \times S$, S is a set of states, and Act is a set of actions. We also write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$.*

Remark 357 *In section 19.1, we have presented a transition system for the $\mathsf{Imp}_{\|}$ language. It is possible to regard this system as a lts by taking the set of states S as the collection of $\mathsf{Imp}_{\|}$ processes and Act as the collection of pairs of* memory states *(not to be confused with the states of the lts we are defining). Then we would write $(P, (s, s'), Q) \in \rightarrow$ if $(P, s) \rightarrow (Q, s')$ according to the rules in Table 19.2.*

Inspired by definition 318 of trace for $\mathsf{Imp}_{\|}$ processes, we introduce a notion of trace and trace equivalence on lts.

Definition 358 (traces) *We define the* set of traces *of a state s in a lts as:*

$$tr(s) = \{\alpha_1 \cdots \alpha_n \mid s \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}\} \ .$$

We say that two states s and t are trace equivalent if $tr(s) = tr(t)$.

There are a few points to be noticed concerning the definition 358 above. First, it neglects *termination* since this notion is not even present in the definition 356 of lts (but a termination predicate on states could be added). And since termination is neglected, the set of traces is closed under prefex. Second, there is no notion of closure of the traces under reflexivity and transitivity. This point is treated later in section 24.3 once the notion of *internal action* is introduced. Third, the *environment* seems to play no role in the behavior of the lts and the related definition of trace equivalence. We shall see in chapter 26 that it is possible to enrich lts with a notion of synchronization and parallel composition and then prove that the notion of trace equivalence in definition 358 is indeed preserved by parallel composition.

24.2 Bisimulation

In chapter 19, we have motivated the interest of accounting for the *branching behavior* of a system (example 313 of the vending machine). The notion of *(bi-)simulation* is a very popular approach to this issue. We have already met this notion in chapter 9 in the framework of the λ-calculus and in chapter 22 in the framework of the $\mathsf{Imp}_{\|}$ concurrent language. Next, we reconsider this notion in the setting of labelled transition systems. The proposed definition ignores certain observables such as termination and deadlock. However, it is quite possible to enrich the notion of lts with predicates that represent termination and/or deadlock and then to formulate a notion of (bi-)simulation which depends on these predicates.

Definition 359 (bisimulation) *Let* $\rightarrow \subseteq S \times Act \times S$ *be a* labelled transition system. *A binary relation* $\mathcal{R}$ *on* S *is a* simulation *if:*

$$\frac{s\ \mathcal{R}\ t, \quad s \xrightarrow{\alpha} s'}{\exists\, t'\ \ t \xrightarrow{\alpha} t', \quad s'\ \mathcal{R}\ t'}\ . \tag{24.1}$$

Moreover we say that $\mathcal{R}$ *is a* bisimulation *if:*

$$\frac{s\ \mathcal{R}\ t, \quad t \xrightarrow{\alpha} t'}{\exists\, s'\ \ s \xrightarrow{\alpha} s', \quad s'\ \mathcal{R}\ t'}\ . \tag{24.2}$$

Remark 360 *In definition 359 as well as in the following ones there is an implicit universal quantification on the states which are not existentially quantified.*

Proposition 361 (on bisimulation) *The following properties hold for the collection of bisimulations over a labelled transitions system:*

1. *The* empty *and* identity *relations are bisimulations.*
2. *The collection of bisimulations is closed under* inverse, composition, *and arbitrary* unions.
3. *There is a* greatest bisimulation *which is defined as the union of all bisimulations and that we represent with* $\sim$ *(some authors call it* bisimilarity*).*
4. *Bisimulations are* not *closed under (finite) intersection.*

Proof. Properties (1-3) follow by a simple unravelling of the definitions. For property (4), consider the lts $1 \xrightarrow{a} 2$, $1 \xrightarrow{a} 3$, $x \xrightarrow{a} y$, $x \xrightarrow{a} z$ and the relations $\mathcal{R}_1 = \{(1,x),(2,y),(3,z)\}$, $\mathcal{R}_2 = \{(1,x),(2,z),(3,y)\}$. □

Exercise 362 (on simulation) *Show that the properties above are* true of simulations too *but for closure under inverse. Further, denote with* $\leq$ *the* greatest simulation. *Find a lts with states* s, t *such that* $s \leq t$, $t \leq s$, *and* $s \not\sim t$.

Exercise 363 (trace vs. simulation) *For* s, t *states of a lts show that* $s \leq t$ *implies* $tr(s) \subseteq tr(t)$, *while the converse may fail.*

Remark 364 (alternative definition of bisimulation) *Sometimes a bisimulation is defined as a* symmetric relation $\mathcal{R}$ *such that:*

$$\frac{s \mathcal{R} t, \quad s \stackrel{\alpha}{\to} s'}{\exists t' \;\; t \stackrel{\alpha}{\to} t', \quad s' \mathcal{R} t'} \; .$$

The advantage of this definition is that one can omit the second condition (24.2). The inconvenience is that by forcing a bisimulation to be symmetric we make it larger than really needed. However, notice that given a bisimulation $\mathcal{R}$ *one can* always *derive a symmetric relation which is a bisimulation by taking* $\mathcal{R} \cup \mathcal{R}^{-1}$.

Let $\to \subseteq S \times Act \times S$ be a *labelled transition system.* Notice that $L = 2^{S \times S}$ is a *complete lattice* with respect to inclusion (cf. definition 172). Bisimulation can be characterized as the greatest fixed point of a certain monotonic function $\mathcal{F}$ on binary relations which we introduce below.

Definition 365 (function $\mathcal{F}$) *We define* $\mathcal{F} : L \to L$ *as:*

$$\begin{aligned} \mathcal{F}(\mathcal{R}) = \quad \{(s,t) \mid \quad & s \stackrel{\alpha}{\to} s' \text{ implies } \exists t' \;\; t \stackrel{\alpha}{\to} t' \text{ and } s' \mathcal{R} t' \text{ and} \\ & t \stackrel{\alpha}{\to} t' \text{ implies } \exists s' \;\; s \stackrel{\alpha}{\to} s' \text{ and } s' \mathcal{R} t'\} \; . \end{aligned}$$

The following properties of the function $\mathcal{F}$ are easily checked (cf. proposition 175).

Proposition 366 *The following properties hold:*

1. $\mathcal{R}$ *is a* bisimulation *iff* $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$.
2. $\mathcal{F}$ *is* monotonic *on* L.
3. *The* greatest bisimulation $\sim$ *is the* greatest fixed point *of* $\mathcal{F}$.

Remark 367 (transfinite definition of bisimulation) *The bisimulation* $\sim$ *being the greatest fixed point of the monotonic function* $\mathcal{F}$, *it can be* approximated from above *as follows (cf. chapter 9):*

$$\sim_{\kappa+1} \; = \; \mathcal{F}(\sim_\kappa) \; , \qquad \sim_\kappa \; = \; \bigcap_{\kappa' < \kappa} \sim_{\kappa'} \qquad (\kappa \text{ limit ordinal}).$$

Thus to show $s \not\sim t$ *it suffices to find an ordinal* κ *such that* $s \not\sim_\kappa t$.

Definition 368 (image finite lts) *A lts* $\to \subseteq S \times Act \times S$ *is* image finite *if for all* P, α *the set* $\{s' \mid s \stackrel{\alpha}{\to} s'\}$ *is* finite.

For finite (image finite) lts the greatest fixed point is reached in a finite (countable) number of iterations.

Proposition 369 (bisimulation for (image) finite lts) *The following properties hold:*

1. *If the support S of the lts is* finite *then there is a natural number n such that the greatest bisimulation coincides with $\sim_n$.*

2. *If the lts is* image finite *then the greatest bisimulation coincides with $\sim_\omega$,* i.e.*, on image finite lts $\mathcal{F}$ is* co-continuous *(preserves intersections).*

Proof. For the first property, see exercise 177. For the second property, suppose $s \sim_\omega t$ and $s \xrightarrow{\alpha} s'$. Then:

$$\forall\, n \;\; \exists\, t_n \;\; t \xrightarrow{\alpha} t_n \text{ and } s' \sim_n t_n \ .$$

Since the set $\{t' \mid t \xrightarrow{\alpha} t'\}$ is finite there must be a t' in this set such that $\{n \mid t_n = t'\}$ is infinite.[1] Thus there is an infinite sequence $n_1 < n_2 < n_3 < \cdots$ such that $s' \sim_{n_j} t_{n_j} = t'$. Then for any n we can find a $n_j \geq n$ such that $s' \sim_{n_j} t'$; and this entails $s' \sim_n t'$. Hence $s' \sim_\omega t'$. □

We conclude this section by introducing a notation to denote lts which will be extended in chapter 26 to a full language of processes known as *CCS*. The notation is generated by the following grammar:

$$P ::= 0 \mid \alpha.P \mid P + P \qquad \alpha \in Act \tag{24.3}$$

Here 0 denotes the empty lts, also called *nil*, $\alpha.P$ is the lts denoted by P *prefixed* by the transition α, and $P + Q$ is the non-deterministic sum of the lts denoted by P and Q. In this notation, the states' identities are immaterial; what matters of a state is not its name but the actions it can do. Using this notation, the lts version of the vending machines in example 313 can be represented as follows:

$$a.(b.0 + c.0) \text{ vs. } a.b.0 + a.c.0 \ ,$$

Notice that identifying the two machines amounts to *distribute* the prefix over the non-deterministic sum. We use the following abbreviations: b for $b.0$, b^n for $b.\ldots.b.0$ (b prefixed n times), and b^ω for the infinite lts $b.b.\cdots$. If I is a (possibly infinite) set then $\Sigma_{i\in I} P_i$ denotes the non-deterministic sum of the lts denoted by P_i. We apply the notation in the following exercise.

Exercise 370 (non-bisimilar lts) *Consider the lts P_i, Q_i defined as follows:*

$$\begin{array}{lcllcl} P_0 & = & b.0 & Q_0 & = & c.0 \\ P_{i+1} & = & a.(P_i + Q_i) & Q_{i+1} & = & a.P_i + a.Q_i \ . \end{array}$$

1. *Show that for all i natural number: (i) $P_i \sim_i P_i + Q_i \sim_i Q_i$, (ii) $P_i \not\sim_{i+1} P_i + Q_i \not\sim_{i+1} Q_i \not\sim_{i+1} P_i$.*

2. *Show that: (i) $\forall\, i \leq n \;\; b^n \sim_i b^\omega$, (ii) $\Sigma_{i\geq 0} b^i + b^\omega \sim_\omega \Sigma_{i\geq 0} b^i$, (iii) $\forall\, i \;\; b^i \not\sim_\omega b^\omega$, (iv) $\Sigma_{i\geq 0} b^i + b^\omega \not\sim_{\omega+1} \Sigma_{i\geq 0} b^i$.*

[1] This is a version of the so called *pigeonhole principle* which states that if infinitely many pigeons are put in finitely many boxes then at least one box must contain infinitely many pigeons.

24.3 Weak transitions

Certain computation steps should not be directly observable. For instance, in sequential programs usually one is just interested in the input-output behavior and not in the way the output is computed. To model this situation in lts, we enrich the collection of actions with a *distinct internal action* τ. For instance, τ^ω is a diverging system which never interacts with the environment. As another example, we could regard a system such as $a.\tau.b.0$ equivalent to $a.b.0$. Though the internal action is not directly observable, it may make a difference. For instance, consider the lts $a.0 + b.0$ and $\tau.a.0 + \tau.b.0$ with the interpretation: $a =$ 'accepts to deliver coffee' and $b =$ 'accepts to deliver tea'. The second system decides 'internally' whether to deliver coffee or tea while the first will take a decision that may be controlled by the environment.

Given a lts with τ transitions, we derive a related lts with the same states but where an observable transition may be preceded and followed by an arbitrary number of internal transitions (think of ϵ transitions in automata theory).

Definition 371 (derived weak lts) *Let a lts* $\rightarrow \subseteq S \times (Act \cup \{\tau\}) \times S$ *be given where* $\tau \notin Act$ *is a* distinct internal action. *We derive from this lts another* weak lts $\Rightarrow \subseteq S \times (Act \cup \{\tau\}) \times S$ *where:*

$$\stackrel{\alpha}{\Rightarrow} = \begin{cases} (\stackrel{\tau}{\rightarrow})^* & \text{if } \alpha = \tau \\ (\stackrel{\tau}{\rightarrow})^* (\stackrel{\alpha}{\rightarrow}) (\stackrel{\tau}{\rightarrow})^* & \text{otherwise.} \end{cases}$$

Remark 372 *When working with weak transitions, lts tend to be image* infinite*, and therefore proposition 368* cannot *be applied.*

The notion of bisimulation for lts with internal actions is simply the standard notion of bisimulation on the derived weak lts.

Definition 373 (weak bisimulation) *Let* $\rightarrow \subseteq S \times (Act \cup \{\tau\}) \times S$ *be a lts with a distinct internal action* τ*. A binary relation* $\mathcal{R}$ *on* S *is a* weak bisimulation *if it is a bisimulation with respect to the weak transition system* $\Rightarrow$*. We denote with* $\approx$ *the largest weak bisimulation.*

The following definition of weak bisimulation is the one which is used *in practice.*

Definition 374 (one step weak bisimulation) *A relation* $\mathcal{R}$ *is a* one step weak bisimulation *if:*

$$\frac{s \mathcal{R} t \quad s \stackrel{\alpha}{\rightarrow} s'}{\exists t' \ t \stackrel{\alpha}{\Rightarrow} t', \quad s' \mathcal{R} t'}, \qquad \frac{s \mathcal{R} t \quad t \stackrel{\alpha}{\rightarrow} t'}{\exists s' \ s \stackrel{\alpha}{\Rightarrow} s', \quad s' \mathcal{R} t'}.$$

Proposition 375 *A relation* $\mathcal{R}$ *on a lts is a* weak bisimulation *iff it is a* one step weak bisimulation.

Proof. By diagram chasing. □

Henceforth we just speak of *weak bisimulation* and use the more convenient definition.

Definition 376 (weak up to strong) *We say that a relation* $\mathcal{R}$ *on a lts is a* weak bisimulation up to strong bisimulation *if:*

$$\frac{s \mathcal{R} t, \quad s \stackrel{\alpha}{\rightarrow} s'}{\exists t' \ t \stackrel{\alpha}{\Rightarrow} t', \quad s' \sim \mathcal{R} \sim t'}, \qquad \frac{s \mathcal{R} t, \quad t \stackrel{\alpha}{\rightarrow} t'}{\exists s' \ s \stackrel{\alpha}{\Rightarrow} s', \quad s' \sim \mathcal{R} \sim t'}.$$

Exercise 377 *Show that if* $\mathcal{R}$ *is a weak bisimulation up to strong bisimulation then* $\mathcal{R} \subseteq \approx$.

24.4 Proof techniques for bisimulation

The standard method to prove $s \sim t$ is to *exhibit a relation* $\mathcal{R}$ such that $s\ \mathcal{R}\ t$ and $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$, where $\mathcal{F}$ is as in definition 365. Exercise 377 suggests that it is possible to refine this proof technique by exhibiting a relation $\mathcal{R}$ which is a bisimulation *up to a relation* 'with suitable properties'. In the following, we provide a rather general treatment of what 'with suitable properties' means. First, some preliminary remarks. Let $(L, \leq)$ be a complete lattice and $f : L \to L$ be a monotonic function on L. The function f induces a *transitive relation* $<_f$ which refines $\leq$:

$$x <_f y \text{ if } x \leq y \text{ and } x \leq f(y) \ .$$

Notice that $<_f$ is *anti-symmetric* but *not necessarily reflexive*. In the case we are interested in, L is the power-set $2^{S\times S}$, f is $\mathcal{F}$, and $\mathcal{R}$ is a bisimulation iff $\mathcal{R} <_{\mathcal{F}} \mathcal{R}$.

Definition 378 *We say that a function $h : L \to L$* preserves $<_f$ *if:*

$$x <_f y \text{ implies } h(x) <_f h(y) \ .$$

Exercise 379 *Show that the set of functions preserving the order $<_f$ is closed under* composition *and* supremum*.*

Proposition 380 (key property) *Let $(L, \leq)$ be a complete lattice, $f : L \to L$ be a monotonic function (with greatest fixed point $gfp(f)$), and $h : L \to L$ be a function that* preserves $<_f$*. Then:*

$$x \leq f(h(x)) \quad \textit{implies} \quad x \leq gfp(f) \ .$$

Proof. Given x, we build a bigger element y such that $y \leq f(y)$. To this end, we define a sequence $x_0 = x$, $x_{n+1} = x_n \vee h(x_n)$. Let $y = \bigvee_{n\geq 0} x_n$; obviously $x_n \leq x_{n+1}$ and $x \leq y$. We show that $x_n <_f x_{n+1}$, by induction on n.

$n = 0$ $x_0 = x \leq f(h(x)) \leq f(x \vee h(x))$, since $x \leq f(h(x))$ by hypothesis and f is monotonic.

$n > 0$ We have to show:

$$x_n = x_{n-1} \vee h(x_{n-1}) \leq f(x_n \vee h(x_n)) = f(x_{n+1}) \ .$$

Since f is monotonic, we have $f(x_n) \vee f(h(x_n)) \leq f(x_n \vee h(x_n))$. By inductive hypothesis, we know $x_{n-1} \leq f(x_n)$. Moreover, since h preserves $<_f$, we have $h(x_{n-1}) \leq f(h(x_n))$.

Finally, we remark that $y \leq f(y)$, as:

$$y = \bigvee_{n\geq 0} x_n \leq \bigvee_{n\geq 0} f(x_{n+1}) \leq f(\bigvee_{n\geq 1} x_n) = f(y) \ .$$

Since $y \leq f(y)$ implies $y \leq gfp(f)$, we conclude $x \leq gfp(f)$. □

Exercise 381 (when h is a closure) *We say that a function h on a lattice L is a closure if $id \leq h = h \circ h$. Show that if h is a closure then $y = h(x)$ in the previous construction.*

In our *application scenario*, this means that to prove that s and t are bisimilar it suffices to find: (1) a function $\mathcal{H}$ that preserves $<_{\mathcal{F}}$ and (2) a relation $\mathcal{R}$ such that $\mathcal{R} \subseteq \mathcal{F}(\mathcal{H}(\mathcal{R}))$.

Exercise 382 *Let $\mathcal{H}(\mathcal{R}) = \sim \circ \mathcal{R} \circ \sim$. Check that $\mathcal{H}$ preserves $<_{\mathcal{F}}$.*

We introduce a notion of weak bisimulation up to expansion which is often used in applications.

Definition 383 (expansion) *A binary relation $\mathcal{R}$ on a lts is an* expansion *if:*

$$\frac{s\ \mathcal{R}\ t, \quad s \stackrel{\alpha}{\to} s'}{\exists t'\ \ t \stackrel{\hat{\alpha}}{\Rightarrow} t' \quad s'\ \mathcal{R}\ t'} \ , \qquad \frac{s\ \mathcal{R}\ t \quad t \stackrel{\alpha}{\to} t'}{\exists s'\ \ s'\ \mathcal{R}\ t' \text{ and } (s \stackrel{\hat{\alpha}}{\to} s' \text{ or } (\alpha = \tau, s' = s))} \ .$$

We denote with $\preceq$ be the largest expansion *and with $\succeq$ its inverse. Also we read $s \preceq t$ as s* expands *to t.*

Note that an expansion is a hybrid object which is *weak on the left* and *almost strong on the right*. The intuition is that the state on the right is a kind of implementation of the one on the left, *i.e.*, the state on the right may take more internal steps to perform the 'same task'.

Exercise 384 (weak bisimulation up to expansion) *Define: $\mathcal{H}(\mathcal{R}) = \succeq \circ \mathcal{R} \circ \preceq$. Let $\mathcal{F}$ be the monotonic function induced by the definition of (one step) weak bisimulation. Show that:*

1. *$s \sim t$ implies strictly $s \preceq t$ (and $s \succeq t$).*
2. *$\preceq \cup \succeq$ implies strictly $\approx$.*
3. *$\mathcal{H}$ preserves $<_{\mathcal{F}}$. And explicit the condition that needs to be checked to ensure that a relation $\mathcal{R}$ is a weak bisimulation up to expansion.*

The following exercise highlights two possible pitfalls in the usage of up-to techniques.

Exercise 385 (pitfalls) *Define: $\mathcal{H}'(\mathcal{R}) = \approx \circ \mathcal{R} \circ \approx$ and $\mathcal{H}''(\mathcal{R}) = \preceq \circ \mathcal{R} \circ \succeq$. As in the previous exercise, let $\mathcal{F}$ be the monotonic function induced by the definition of (one step) weak bisimulation. Show that:*

1. *$\mathcal{H}'$ does* not *preserve $<_{\mathcal{F}}$. Suggestion: consider $\mathcal{R} = \{(\tau.a, 0)\}$ and check that $\mathcal{R} \subseteq \mathcal{F}(\mathcal{H}'(\mathcal{R}))$ while obviously $\mathcal{R} \not\subseteq \approx$.*
2. *$\mathcal{H}''$ does* not *preserve $<_{\mathcal{F}}$, by an argument similar to the one used for $\mathcal{H}'$.*

24.5 Summary and references

The notion of *labelled transition system* provides an abstract setting to explore the variety of possible semantics of concurrent systems. In particular, we have developed the notion of *bisimulation* which corresponds to the greatest fixed point of a certain monotonic function on lts. Bisimulation is a *natural notion* and Park [Par81] seems the first to have used it in the semantics of programming languages. In order to abstract the internal behavior of a system, we have introduced the notion of *internal action* and the related notions of *weak transition* and *weak bisimulation*. Finally, we have discussed an *up to proof technique* which allows to reduce the size of the relation to be exhibited to show that two lts are bisimilar.

Chapter 25

Modal logics

In chapter 23, we have considered partial correctness and rely-guarantee assertions as means to specify the behaviour of concurrent processes and in doing this we have faced some problems due to the limited expressive power of the specification language. In this chapter, we take a bold step in that for a given notion of equivalence on lts we aim at a specification language which captures exactly the equivalence. The presented languages build on the notion of (propositional) *modal logic* which is an extension of usual logic with *modalities* that qualify the validity of the assertions: possibly true, necessarily true,... In particular, we introduce a *diamond modality* indexed over the actions of the lts and stipulate:

$$s \models \langle\alpha\rangle A \text{ if } \exists\, s' \;\; s \xrightarrow{\alpha} s' \text{ and } s' \models A \;,$$

which is read as follows: a state s satisfies the formula $\langle\alpha\rangle A$ if there is a state s' such that $s \xrightarrow{\alpha} s'$ and s' satisifies A. It turns out that the full *infinitary* specifications generated by this extension characterize bisimulation, while restricted versions correspond to coarser equivalences such as simulation or trace equivalences.

In practice, one needs finite means to describe 'infinitary' specifications. An elegant way to achieve this, is to define (monotonic) formulae by least and greatest fixed points (in the spirit of proposition 175). The resulting modal language is called the μ-calculus. For finite state lts, we present a simple algorithm to decide whether a state satisfies a formula of the μ-calculus.

25.1 Modal logics vs. equivalences

We introduce a modal logic which consists of a classical propositional logic enriched with a so called *diamond* modality describing the ability to perform an action.

Definition 386 (formulae) *The collection of formulae of a propositional modal logic is defined as:*

$$A ::= \bigwedge_{i\in I} A_i \mid \neg A \mid \langle\alpha\rangle A \qquad \alpha \in Act \tag{25.1}$$

where the set I can also be empty or infinite. By convention, we write true *for $\bigwedge \emptyset$ and $[\alpha]A$ for $\neg\langle\alpha\rangle\neg A$.*

Definition 387 (formulae satisfaction) *We define when a state in a lts satisfies a formula, written $s \models A$, as follows:*

$$\begin{array}{ll} s \models \bigwedge_{i\in I} A_i & \text{if } \forall i \in I \ s \models A_i \\ s \models \neg A & \text{if } s \not\models A \\ s \models \langle\alpha\rangle A & \text{if } \exists s' \ s \xrightarrow{\alpha} s' \text{ and } s' \models A \ . \end{array}$$

We also write:

$$\begin{array}{lll} [\![A]\!] & = \{s \mid s \models A\} & \text{(formula interpretation)} \\ [\![s]\!] & = \{A \mid s \models A\} & \text{(state interpretation)} \\ s \sim_L s' & \text{if } [\![s]\!] = [\![s']\!] & \text{(logical equivalence) .} \end{array}$$

Exercise 388 (on modal formulae) *Spell out what it means to satisfy $[\alpha]A$. Find a formula showing that $a.(b.0 + c.0) \not\sim_L a.b.0 + a.c.0$.*

It is easily checked that two bisimilar states are logically equivalent.

Proposition 389 *Let s, s' be states in a lts. If $s \sim s'$ then $s \sim_L s'$.*

To show the converse of proposition 389, we introduce the (possibly infinite) so called *characteristic formulae*.

Definition 390 (characteristic formula) *Given a state s in a lts and an ordinal κ the characteristic formula $C^\kappa(s)$ is defined as follows:*

$$\begin{array}{lcll} C^{\kappa+1}(s) & = & \bigwedge_{s\xrightarrow{\alpha}s'} \langle\alpha\rangle C^\kappa(s') \ \wedge & \\ & & \bigwedge_{\alpha\in Act} [\alpha](\ \bigvee_{s\xrightarrow{\alpha}s'} C^\kappa(s')\ ) & \\ C^\kappa(s) & = & \bigwedge_{\kappa'<\kappa} C^{\kappa'}(s) & \text{(κ limit ordinal).} \end{array} \tag{25.2}$$

Proposition 391 *For any state s and ordinal κ:*

1. $\models s : C^\kappa(s)$.
2. $\models s' : C^\kappa(s)$ *iff* $s \sim_\kappa s'$, *where $\sim_\kappa$ is the approximation of bisimulation defined in remark 367.*

Proof. To prove a property for all ordinals one relies on the principle of transfinite induction. Namely one shows that if a property is true of all ordinals less than κ then it is true of κ. □

Exercise 392 *Suppose that the set of actions Act is* finite. *Then show for* image finite *lts the following property: if two processes are not bisimilar then there is a* finite formula *that distinguishes them.*

Given that full modal logic characterizes bisimulation, one may look for fragments of the logic that characterize coarser equivalences. We consider the cases of trace (definition 358) and simulation equivalence (definition 359).

Proposition 393 (trace equivalence) *The modal formulae A of the following shape characterize trace equivalence:*

$$A ::= B \mid C \mid \bigwedge_{i \in I} A_i \ ,$$

where: $B ::= \mathsf{true} \mid \langle\alpha\rangle B$ *and* $C ::= \mathsf{false} \mid [\alpha] C$.

Proof. One defines the characteristic formula as follows:

$$C(s) = \bigwedge_{\alpha_1 \cdots \alpha_n \in tr(s)} \langle\alpha_1\rangle \cdots \langle\alpha_n\rangle \mathsf{true} \wedge \bigwedge_{\alpha_1 \cdots \alpha_n \notin tr(s)} [\alpha_1] \cdots [\alpha_n] \mathsf{false} \ .$$

□

Proposition 394 (simulation equivalence) *The modal formulae A of the following shape characterize simulation equivalence:*

$$A ::= \bigwedge_{i \in I} A_i \mid \langle\alpha\rangle A \ .$$

Proof. We build the formula $C^\kappa(s)$ taking the left hand side of the formula (25.2) that works for bisimulation. Then (1) $\models s : C^\kappa(s)$ and (2) $\models s' : C^\kappa(s)$ iff $s \leq_\kappa s'$, where $\leq_\kappa$ is the approximation of simulation. One shows that $s \leq s'$ and $\models s : A$ implies $\models s' : A$. On the other hand if s and s' are logically equivalent then $\models s' : C(s)$ and $\models s : C(s')$. Therefore $s \leq s'$ and $s' \leq s$. □

25.2 A modal logic with fixed points: the μ-calculus

In the presented modal language, to express, *e.g.*, that a process can do infinitely many actions α we need an *infinite formula*. It is possible to increase the expressive power of formulae while keeping the syntax *finite*. An elegant extension known as *μ-calculus* consists in adding to the logical formulae *least fixed points*. Then the syntax of modal formulae given in definition 386 is revisited as follows.

Definition 395 (formulae with fixed points) *The modal formulae with fixed points have the following syntax:*

$$\begin{array}{ll} id ::= x \mid y \mid \ldots & \textit{(formula identifiers)} \\ A ::= \bigwedge_{i \in I} A_i \mid \neg A \mid \langle\alpha\rangle A \mid id \mid \mu id.A & \textit{(formulae).} \end{array}$$

In a formula $\mu x.A$ the identifier x is bound in A by the least fixed point operator μ. Also we assume that each free occurrence of x in A is *positive*, *i.e.*, under an even number of negations. This positivity condition is essential to show that the function induced by the formula is monotonic and therefore has a least (and a greatest) fixed point (cf. exercise 396 below).

Since a formula may contain free identifiers, its interpretation is given relatively to an assignment $\rho : id \to 2^S$ as follows:

$$\begin{array}{ll} [\![\bigwedge_{i \in I} A_i]\!]\rho & = \bigcap_{i \in I} [\![A_i]\!]\rho \\ [\![\neg A]\!]\rho & = ([\![A]\!]\rho)^c \\ [\![\langle\alpha\rangle A]\!]\rho & = \{s \mid s \xrightarrow{\alpha} s' \text{ and } s' \in [\![A]\!]\rho\} \\ [\![x]\!]\rho & = \rho(x) \\ [\![\mu x.A]\!]\rho & = \bigcap \{X \subseteq S \mid [\![A]\!]\rho[X/x] \subseteq X\} \ . \end{array}$$

Of course, if A is a closed formula its interpretation does *not* depend on the assignment and we can write $s \models A$ if $s \in [\![A]\!]\rho$, for some ρ.

Exercise 396 (positivity) *Check that for all well-formed formulae A, identifier x, and assignments ρ, the function $X \mapsto [\![A]\!]\rho[X/x]$ is monotonic on 2^S. Conclude that the semantics of a formula $\mu x.A$ does indeed correspond to a least fixed point.*

An intuitive way to understand the meaning of a formula $\mu x.A$ is to unfold it as an infinite disjunction $\bigvee_\kappa A^\kappa$ where: $A^{\kappa+1} = [A^\kappa/x]A$ and $A^\kappa = \bigvee_{\kappa'<\kappa} A^{\kappa'}$ for κ limit ordinal. This viewpoint is based on the iterated definition of the least fixed point mentioned in chapter 9.

Greatest fixed points are derived by duality from least fixed points by defining:

$$\nu x.A = \neg\mu x.\neg([\neg x/x]A) \ .$$

For instance: $\nu x.\langle\alpha\rangle x = \neg\mu x.\neg\langle\alpha\rangle\neg x$.

Exercise 397 (greatest fixed points) *Check that the interpretation of ν does indeed correspond to a greatest fixed point, namely:*

$$[\![\nu x.A]\!]\rho = \bigcup\{X \subseteq S \mid X \subseteq [\![A]\!]\rho[X/x]\} \ .$$

We have seen that disjunction and greatest fixed points can be derived from conjunction, least fixed points, and negation. An alternative approach consists in dropping negation and taking conjunction, disjunction, μ and ν operators as primitive. This way we have to deal with an additional operator but we can drop the positivity condition on the fixed points since conjunction and disjunction are guaranteed to induce monotonic functions.

Exercise 398 (deriving negation) *Show that the negation operator can be defined (on closed formulas). Hint: Consider the following equations:*

$$\neg\langle\alpha\rangle A = [\alpha]\neg A, \qquad \neg[\alpha]A = \langle\alpha\rangle\neg A, \qquad \neg\mu x.A = \nu x.\neg A, \quad \neg\nu x.A = \mu x.\neg A \ .$$

It turns out that for *finite lts*, the modal logic with fixed points can express the characteristic formula of a state by a *finite* formula.

Proposition 399 *Let s be a state in a finite lts. Then there is a closed finite characteristic formula $C(s)$ involving only greatest fixed points such that for any state s', $\models s' : C(s)$ iff $s \sim s'$.*

Proof. For every state s introduce a propositional variable x_s and an equation based on the characteristic formula in definition 390:

$$x_s \ = \ \bigwedge_{s \xrightarrow{\alpha} s'} \langle\alpha\rangle x_{s'} \ \wedge \ \bigwedge_{\alpha \in Act} [\alpha](\bigvee_{s \xrightarrow{\alpha} s'} x_{s'})$$

Then the general idea is to take the *greatest* fixed point of this system of equations and project on the component which corresponds to the state s. □

Another interesting property of the μ-calculus on finite lts is that the model-checking problem is *decidable*. We spend the rest of the section to present a proof of this fact that relies on the following elementary property of fixed points.

Proposition 400 (reduction) *Let f be a monotonic function over 2^S and $s \in S$ be a state. Consider the following monotonic functions over 2^S: $(f \cup s)(x) = f(x) \cup \{s\}$ and $(f \backslash s)(x) = f(x) \backslash \{x\}$. Also if g is a monotonic function denote by $\nu(g)$ and $\mu(g)$ its greatest and least fixed point. Then:*

1. $s \in \nu(f)$ *iff* $s \in f(\nu(f \cup s))$.
2. $s \in \mu(f)$ *iff* $s \in f(\mu(f \backslash s))$.
3. $s \in \nu(f \cup s)$.
4. $s \notin \nu(f \backslash s)$.

Proof. (1) Suppose $s \in \nu(f)$. Then:

$$f(\nu(f)) \cup \{s\} = \nu(f) \cup \{s\} = \nu(f) \ .$$

By definition of $\nu(f \cup s)$ this implies $\nu(f) \subseteq \nu(f \cup s)$. By monotonicity, $\nu(f) = f(\nu(f)) \subseteq f(\nu(f \cup s))$, and therefore $s \in f(\nu(f \cup s))$. On the other hand, suppose $s \in f(\nu(f \cup s))$. It follows:

$$\nu(f \cup s) = f(\nu(f \cup s)) \cup \{s\} = f(\nu(f \cup s)) \ .$$

By definition of $\nu(f)$ this implies $\nu(f \cup s) \leq \nu(f)$. By monotonicity, $f(\nu(f \cup s)) \leq f(\nu(f)) = \nu(f)$, and therefore $s \in \nu(f)$.

(2) Prove by a dual argument: $s \notin \mu(f)$ iff $s \notin f(\mu(f \backslash s))$.

$(3-4)$ Immediate by unfolding the fixed point. □

This proposition suggests a strategy to unfold recursive formulae. The starting idea is to tag each fixed point with a set of states. Then properties (1-2) of proposition 400 when read from left to right suggest to record in the tag the states that are crossed when unfolding a fixed point while properties (3-4) of proposition 400 provide the halting conditions. To formalize this idea, we begin by introducing the syntax of *tagged* formulae.

Definition 401 (formulae with tagged fixed points) *The modal formulae with tagged fixed points have the following syntax:*

$id ::= x \mid y \mid \ldots$	*(formula identifiers)*
$T ::= \{s_1, \ldots, s_n\}$	*(tags, finite sets of states)*
$A ::= \bigwedge_{i \in I} A_i \mid \bigvee_{i \in I} A_i \mid \langle \alpha \rangle A \mid [\alpha] A \mid id \mid \mu id : T.A \mid \nu id : T.A$	*(tagged formulae).*

The interpretation of tagged fixed points is as follows while the interpretation of the logical and modal operators is left unchanged:

$$\begin{aligned} [\![\mu x : T.A]\!]\rho &= \bigcap \{X \subseteq S \mid ([\![A]\!]\rho[X/x]) \backslash T \subseteq X\} \ , \\ [\![\nu x : T.A]\!]\rho &= \bigcup \{X \subseteq S \mid X \subseteq ([\![A]\!]\rho[X/x]) \cup T\} \ . \end{aligned}$$

Based on this interpretation and proposition 400, we introduce in Table 25.1 the collection of rules to model-check states against finite formulae of the μ-calculus.

Proposition 402 (soundness) *Let A be a closed formula of the modal, tagged μ-calculus. If we can derive the assertion $s : A$ according to the rules in Table 25.1 then $s \in [\![A]\!]$.*

$$\frac{s : A \quad s : B}{s : A \wedge B}$$

$$\frac{s : A}{s : A \vee B} \qquad \frac{s : B}{s : A \vee B}$$

$$\frac{s' : A}{s : \langle\alpha\rangle A} \text{ for some } s \xrightarrow{\alpha} s' \qquad \frac{s' : A}{s : [\alpha]A} \text{ whenever } s \xrightarrow{\alpha} s''$$

$$\frac{s \notin T \quad s : [\mu x : T \cup \{s\}.A/x]A}{s : \mu x : T.A} \qquad \frac{s \notin T \quad s : [\nu x : T \cup \{s\}.A/x]A}{s : \nu x : T.A}$$

$$\frac{s \in T}{s : \nu x : T.A}$$

Table 25.1: A model checker for the μ-calculus

PROOF. By induction on the height of the proof, relying on the reduction proposition 400 for the rules that fold the fixed points. □

Proving completeness of the method for finite state lts amounts to prove the termination of the unfolding process. Suppose we look at the rules in Table 25.1 bottom up. All rules but those that unfold fixed points either entail termination or shrink the size of the formula to be proved. Hence any infinite backward development must include an infinite number of applications of the rules unfolding fixed points. Now we remark that these rules add new elements to the tags. Since $T \subseteq S$ and S is finite, we might conjecture that this process eventually terminates. We prove this property in two steps. First, we present a simple rewriting system whose termination proof exposes the kernel of the combinatorial problem. Second, we show termination of the bottom up proof development by exhibiting a reduction preserving translation from judgments to terms of the simple rewriting system.

Definition 403 *We define a collection of σ-terms as follows where n is a natural number:*

$$\begin{array}{ll} id ::= x \mid y \mid \cdots & \textit{(identifiers)} \\ \theta ::= id \mid 1 \mid \theta * \theta \mid \bullet\theta \mid \sigma^n id.\theta & \textit{(σ-terms)} \end{array}$$

Definition 404 *A term θ can be reduced according to the following rules where the rules can be applied at top level only:*

$$\sigma^{n+1}x.\theta \to [\sigma^n x.\theta/x]\theta, \quad \bullet\theta \to \theta, \quad \theta * \theta' \to \theta, \quad \theta * \theta' \to \theta' \ .$$

Proposition 405 *The rewriting system defined in 404 terminates.*

PROOF. Let WF be the collection of terminating σ-terms. If $\theta \in WF$ let $d(\theta)$ be the length of the longest reduction sequence (this is well defined because the reduction tree is finitely branching). We want to prove:

$$\theta, \theta' \in WF \quad \text{implies} \quad [\theta'/x]\theta \in WF \ . \tag{25.3}$$

We prove (25.3) by induction on $d(\theta)$. The only interesting case is when θ has the shape $\sigma^{n+1}y.\theta$. Then we observe:

$$[\theta'/x](\sigma^{n+1}y.\theta) \equiv \sigma^{n+1}y.[\theta'/x]\theta \to [\sigma^n y.[\theta'/x]\theta/y]([\theta'/x]\theta) \equiv [\theta'/x][\sigma^n y.\theta/y]\theta \ .$$

We note that $(\sigma^{n+1}y.\theta) \to [\sigma^n y.\theta/y]\theta \in WF$. Hence, we can apply the inductive hypothesis on $d([\sigma^n y.\theta/y]\theta)$, and we conclude that $[\theta'/x][\sigma^n y.\theta/y]\theta \in WF$.

Next we prove that all σ-terms terminate. We proceed by induction on a relation $\succ$ which is the least transitive relation such that:

$$\sigma^{n+1}x.\theta \succ \sigma^n x.\theta \ , \quad \sigma^{n+1}x.\theta \succ \theta \ , \quad \bullet\theta \succ \theta \ , \quad \theta * \theta' \succ \theta \ , \quad \theta * \theta' \succ \theta' \ .$$

Clearly $\succ$ is a well founded relation. Again the only interesting case is when the term has the shape $\sigma y^{n+1}.\theta$. By the inductive hypothesis $\sigma y^n.\theta \in WF, \theta \in WF$, and by (25.3) $[\sigma y^n.\theta/y]\theta \in WF$. □

Definition 406 *Given a finite lts with a set of states S, we associate a σ-term to a modal formula as follows, where $n = \sharp S + 1$:*

$$\langle x\rangle = x \ , \qquad \langle A \wedge B\rangle = \langle A\rangle * \langle B\rangle \ , \quad \langle A \vee B\rangle = \langle A\rangle * \langle B\rangle \ ,$$
$$\langle\langle\alpha\rangle A\rangle = \bullet\langle A\rangle \ , \qquad \langle[\alpha]A\rangle = \bullet\langle A\rangle \ , \qquad \langle\mu x : T.A\rangle = \sigma x^{(n-\sharp T)}.\langle A\rangle \ ,$$
$$\langle\nu x : T.A\rangle = \sigma x^{(n-\sharp T)}.\langle A\rangle \ .$$

Suppose $s' : A'$ is a premise of $s : A$ in the proof development. We show $\langle A\rangle \to \langle A'\rangle$ by inspection of the proof rules. The only interesting case is when we unfold a fixed point. Since in the translation we have picked $n = \sharp S + 1$ bigger than $\sharp T$ we can compute, *e.g.*, in the case of the least fixed point:

$$\langle\mu x : T.A\rangle = \sigma x^{(n-\sharp T)}.\langle A\rangle \to [\sigma x^{(n-\sharp T-1)}.\langle A\rangle/x]\langle A\rangle = \langle[\mu x : T \cup \{s\}.A/X]A\rangle \ .$$

Proposition 407 *The model checker is complete on finite structures.*

Proof. We show by induction on A that $\models s : A$ iff a proof rule applies. We can bound the depth of a path in a bottom up proof development. Hence, if $\models s : A$ by developing the proof bottom up we eventually obtain a proof of $s : A$. □

25.3 Summary and references

We have described a family of *modal logical languages* which can be used to characterize bisimulation as well as coarser equivalences. We have also presented a few basic results on a fixed point extension of modal logic known as μ-calculus. The μ-calculus is a kind of basic modal logical language to which more user-friendly logical languages can be compiled. It was introduced by Kozen [Koz83], following previous work by V. Pratt. The simple proof of decidability of the model checking problem for finite lts we have presented is based on [Win89]. The model-checking problem for the μ-calculus is known to be in NP ∩ co-NP (like the graph isomorphism problem). Upper bounds on the time complexity are polynomial in the size of the lts and exponential in the so called *alternation depth* of the formula. This is a measure that counts the number of alternations of nested greatest and least fixed points. It is also known [Bra96, Len96] that bounding the alternation depth limits the expressivity of the logic, *i.e.*, the hierarchy of formulae obtained by measuring the alternation depth is *strict*. The basic theory of the μ-calculus is developed systematically in [AN01].

Chapter 26

Labelled transition systems with synchronization

One can make the basic model of labelled transition systems a bit more interesting by adding some *parallelism and synchronization* mechanisms. One elegant way to provide a synchronization mechanism is to introduce a *notion of co-action* and suppose that synchronization happens when a process can perform an action and another parallel process can perform the corresponding co-action. Thus, given a set A, take the set of actions to be:

$$Act = \{a, \overline{a} \mid a \in A\} \cup \{\tau\} \ . \tag{26.1}$$

It is convenient to extend the co-action definition to the whole set Act by assuming:

$$\overline{\tau} = \tau \ , \qquad \overline{\overline{a}} = a \ .$$

CCS (Calculus of Communicating Systems) is a minimal set of operators to represent such labelled transition systems enriched with the co-action mechanism; we introduce this formalism and discuss two ways to define its bisimulation semantics which turn out to be equivalent. *CCS* is a *simple* model of concurrent systems and we shall build on it to discuss the notions of *deterministic* (chapter 27), *timed* (chapter 28), and *probabilistic* concurrent system (chapter 29). We shall also consider an extension of *CCS*, known as π-calculus (chapter 30), that allows for a rather direct embedding of higher-order functional programs.

26.1 *CCS*

Actions in *CCS* are defined according to the equation (26.1) above. Besides the nil, prefix and non-deterministic choice operators introduced in chapter 24, *CCS* includes operators to declare a local action (cf. local variable in $\mathsf{Imp}_{\|}$), to put processes in parallel, and to define recursive behaviors:

$$P \ ::= 0 \mid \alpha.P \mid (P + P) \mid (P \mid P) \mid \nu a \ P \mid A(a^*)$$

where $\alpha \in Act$ and $A, B, \ldots$ are process identifiers. An action name is free if it is not in the scope of a local action declaration (a ν). We write a^* for a possibly empty list of action names $a_1, \ldots, a_n$. Similarly, $\nu a^* \ P$ stands for $\nu a_1 \cdots \nu a_n \ P$, and $[b^*/a^*]$ for $[b_1/a_1, \ldots, b_n/a_n]$. It is assumed that each process identifier A is defined by a unique equation $A(b^*) = P$ where

the free names in P are contained in the set of parameters $\{b^*\}$. For instance, A could be a process identifier defined by the equation:

$$A(a,b) = a.\nu c\ (A(a,c) \mid \overline{b}.A(c,b))\ . \tag{26.2}$$

Here the set of variables occurring free in $a.\nu c\ (A(a,c) \mid \overline{b}.A(c,b))$ is $\{a,b\}$ which happens to be included (actually equal) to the set of parameters of the process identifier A. Also notice that an action name, say b, may appear in a prefix as such or in its dual form $\overline{b}$.

Moving towards semantics, the main design decision consists in regarding $a, b, \ldots$ as channel names on which parallel processes synchronize. More precisely, a synchronization may only happen when a process is ready to perform an action and another parallel process is ready to perform its dual action as, *e.g.*, in the process $(a.P \mid \overline{a}.Q)$. Following the synchronization, the process moves to $(P \mid Q)$. *CCS* is an asynchronous model of concurrency where interaction is possible through *rendez-vous* synchronization on *pure* channels. A *rendez-vous* channel is a channel of null capacity where the sender must always wait for a receiver. A channel is *pure* if no message value is exchanged; all that matters is the synchronization.

An important consequence of assuming a synchronization by *rendez-vous* is to offer a better control on the role of the environment. In the $\mathsf{Imp}_{\|}$ model, the environment can modify the (visible part of the) state and these modifications may affect the future computation of the process. In *CCS*, the only way the environment may affect the computation of the process is to perform an action which is dual to an action that the process is *ready* to perform.

Starting from this intuition, we follow *two paths* to define a compositional semantics of *CCS*. The first path consists in associating a labelled transition system with each *CCS* process. Then the equivalences on lts defined in the previous chapter 24, apply to *CCS* processes too and lead to a compositional semantics. The second path consists in looking at *CCS* as a (rudimentary) programming language and define its possible reductions similarly to what we have done for the $\mathsf{Imp}_{\|}$ language in chapter 19. Then what needs to be done is to fix a notion of observable and to derive a notion of compositional equivalence. We work with the notion of (weak) bisimulation introduced in chapter 24 and in the end, we show that the two paths outlined above actually lead to the same compositional equivalence.

As a concrete example illustrating the difference between the two approaches, consider the *CCS* process $P \equiv (a.0 \mid \overline{a}.0)$. In the first approach, we have to consider the labelled transitions:

$$P \xrightarrow{a} (0 \mid \overline{a}.0) \xrightarrow{\overline{a}} (0 \mid 0)\ , \quad P \xrightarrow{\overline{a}} (a.0 \mid 0) \xrightarrow{a} (0 \mid 0)\ , \quad P \xrightarrow{\tau} (0 \mid 0)\ .$$

While in the second, we just have have the reduction:

$$P \to (0 \mid 0)\ .$$

We shall see that the τ transitions correspond to the reductions while the other labelled transitions correspond to interactions with the environment.

26.2 Labelled transition system for *CCS*

Table 26.1 describes a lts for *CCS* processes where the symmetric rules for $\mid$ and $+$ are omitted.

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \{a, \overline{a}\}}{\nu a\ P \xrightarrow{\alpha} \nu a\ P'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{(P \mid Q) \xrightarrow{\tau} (P' \mid Q')}$$

$$\frac{P \xrightarrow{\alpha} P''}{P + P' \xrightarrow{\alpha} P''} \qquad \frac{B(a^*) = P \quad [b^*/a^*]P \xrightarrow{\alpha} P'}{B(b^*) \xrightarrow{\alpha} P'}$$

Table 26.1: Lts for *CCS* (symmetric rules omitted)

Exercise 408 (labelled transitions) *Check that:*

$$\nu b\ (a.P \mid P') \mid \nu c\ \overline{a}.Q \xrightarrow{\tau} \nu b\ (P \mid P') \mid \nu c\ Q\ .$$

Example 409 (an unbounded buffer in *CCS*) *In CCS, the communication is by* rendez-vous *(or handshake, or synchronous). What if we want* channels with buffers*? One approach is to* enrich the model. *Another approach is to show that* buffers can be expressed in *CCS. An unbounded buffer taking inputs on a and producing outputs on b can be written as (up to renaming, this is the same as equation (26.2)):*

$$Buf(a, b) = a.\nu c\ (Buf(a, c) \mid \overline{b}.Buf(c, b))\ .$$

We write more suggestively $a \mapsto b$ for $Buf(a, b)$, assuming $a \neq b$. We would like to show that $a \mapsto b$ works indeed as an unbounded buffer*. Let $\overline{c}^n = \overline{c} \ldots \overline{c}.0$, n times, $n \geq 0$. We should have:*

$$P(n) = \nu a\ (\overline{a}^n \mid a \mapsto b)\ \text{`equivalent to'}\ \overline{b}^n$$

An interesting exercise because $P(n)$ has a non trivial dynamics*. For the time being we just analyze* some *of the labelled transitions of $P(n)$.*

- *For $n = 0$, $P(n)$ cannot reduce.*
- *For $n > 0$, we need to generalize a bit the form of the process $P(n)$. Let $Q(n, m)$ be a process of the form:*

  $$Q(n, m) = \nu a, c_1, \ldots, c_m\ (\overline{a}^n \mid a \mapsto c_1 \mid \cdots \mid c_m \mapsto b)\ ,$$

  for $m \geq 0$. Note that $P(n) = Q(n, 0)$ and $Q(0, k)$ cannot reduce for any k. Moreover, the message can traverse the whole chain so that:

  $$Q(n, m) \stackrel{\overline{b}}{\Rightarrow} Q(n - 1, 2m + 1)\ .$$

  Thus:

  $$P(n) \stackrel{\overline{b}}{\Rightarrow} \cdots \stackrel{\overline{b}}{\Rightarrow} Q(0, 2^n - 1)\ \text{`equivalent to'}\ 0\ ,$$

  where we recall that: $\stackrel{\overline{b}}{\Rightarrow} = (\xrightarrow{\tau})^* \xrightarrow{\overline{b}} (\xrightarrow{\tau})^*$.

Note that there are plenty of reductions we did not consider! Yet, in chapter 27 we shall be able to conclude that this analysis suffices to derive that $P(n)$ is 'equivalent to' $\overline{b}^n$.

We can regard *CCS* as a *labelled transition system* where states are processes. We say that P and Q are *strongly bisimilar* (written $P \sim Q$) if they are bisimilar with respect to the lts we have just defined. We say that they are *weakly bisimilar* (written $P \approx Q$) if they are bisimilar with respect to the derived lts $\stackrel{\alpha}{\Rightarrow}$ where internal actions are 'abstracted'. Obviously $P \sim Q$ implies $P \approx Q$. Next, we consider the issue of compositionality.

Definition 410 (*CCS* context) *A* context C *is a process with a hole* [ ] *(here and in the following we omit the symmetric cases when listing contexts):*

$$C ::= [\,] \mid \alpha.C \mid C + P \mid C \mid P \mid \nu a\ C\ .$$

Proposition 411 *If $P \sim Q$ then $C[P] \sim C[Q]$.*

Proof. We apply the standard technique which amounts to define a relation $\mathcal{R}$ which includes the processes of interest and show that it is a bisimulation.

$$\begin{array}{ll} \text{Prefix} & \mathcal{R} = \{(\alpha.P_1, \alpha.P_2) \mid P_1 \sim P_2\} \cup \sim \\ \text{Sum} & \mathcal{R} = \{(P_1 + Q, P_2 + Q) \mid P_1 \sim P_2\} \cup \sim \\ \text{Parallel} & \mathcal{R} = \{(P_1 \mid Q, P_2 \mid Q) \mid P_1 \sim P_2\} \cup \sim \\ \text{Restriction} & \mathcal{R} = \{(\nu a\ P_1, \nu a\ P_2) \mid P_1 \sim P_2\} \cup \sim\ . \end{array}$$

□

Remark 412 *In general, the non-deterministic sum does* not *preserve weak bisimulation as:*

$$\tau.a \approx a \qquad but \qquad \tau.a + b \not\approx a + b\ .$$

However a *guarded version* of the non-deterministic sum has this property. Denote with D the following contexts:

$$D ::= [\,] \mid \alpha.D \mid \alpha'.D + P \mid (D \mid P) \mid \nu a\ D \qquad \alpha' \neq \tau\ .$$

Proposition 413 *If $P \approx Q$ then $D[P] \approx D[Q]$.*

Proof. Similar to the strong case. □

Remark 414 (on unguarded sum) *There are* two viewpoints *on the non-preservation of weak-bisimulation by the sum:*

1. *One should take the largest congruence which refines $\approx$. Then,* e.g.*, we should distinguish $\tau.a$ from a.*

2. *In most applications one just needs a guarded sum and so it is enough to have a notion of equivalence which is preserved by guarded sums.*

The second viewpoint tends to prevail at least in formalisms like $\mathsf{Imp}_{\|}$ (chapter 19) or the π-calculus (chapter 30) where a guarded form of sum is easily derived from parallel composition.

Exercise 415 (sequentialization in *CCS*) *We consider a fragment of CCS (we drop sum and recursive definitions) extended with an operator ';' for process sequentialization. Thus the* process syntax *is as follows:*

$$P ::= 0 \mid \alpha.P \mid (P \mid P) \mid P;P \mid \nu a\ P \qquad \alpha \in \{a, \overline{a} \mid a \in A\} \cup \{\tau\}\ . \tag{26.3}$$

The labelled transition system *for CCS is extended with the following rules for process sequentialization:*

$$\frac{P \stackrel{\alpha}{\to} P'}{P;Q \stackrel{\alpha}{\to} P';Q} \qquad \frac{P\downarrow \qquad Q \stackrel{\alpha}{\to} Q'}{P;Q \stackrel{\alpha}{\to} Q'}$$

Here the predicate $\downarrow$ denotes proper termination *and it is defined as the least set of processes such that:*[1]

$$\frac{}{0\downarrow} \qquad \frac{P\downarrow \quad Q\downarrow}{(P \mid Q)\downarrow} \qquad \frac{P\downarrow \quad Q\downarrow}{(P;Q)\downarrow} \qquad \frac{P\downarrow}{\nu a\ P\downarrow}\ .$$

If P is a process in the extended language (26.3) and c is a name not occurring in P then $[\![P]\!]c$ is a CCS process (without process sequentialization) defined as follows:

$$\begin{array}{lll}
[\![0]\!]c & = \overline{c}.0 & \\
[\![\alpha.P]\!]c & = \alpha.[\![P]\!]c & \\
[\![P_1 \mid P_2]\!]c & = \nu c_1, c_2\ (([\![P_1]\!]c_1 \mid [\![P_2]\!]c_2) \mid c_1.c_2.\overline{c}.0) & (c_1, c_2 \notin \mathsf{fv}(P_1) \cup \mathsf{fv}(P_2) \cup \{c\}) \\
[\![P_1; P_2]\!]c & = \nu c'\ ([\![P_1]\!]c' \mid c'.[\![P_2]\!]c) & (c' \notin \mathsf{fv}(P_1) \cup \mathsf{fv}(P_2) \cup \{c\}) \\
[\![\nu a\ P]\!]c & = \nu a\ [\![P]\!]c & (c \neq a)
\end{array}$$

(1) Show that if $P\downarrow$ then $[\![P]\!]c \stackrel{\overline{c}}{\Rightarrow} Q$ and $Q \sim 0$, where $\sim$ is the largest strong bisimulation. *(2) Define the notion of* weak simulation up to strong bisimulation *and show that this is a sound technique to prove weak simulation. (3) Show that for all processes P, P is* weakly simulated *by $[\![P]\!]c$ up to strong bisimulation.*

Exercise 416 (bisimulation up-to context) *We introduce a notion of bisimulation up to the contexts. For simplicity let us assume: $D ::= [\ ] \mid \alpha.D \mid (D \mid P) \mid \nu a\ D$. Suppose:*

$$\mathcal{H}(\mathcal{R}) = \{(P', Q') \mid \exists\, D\ \ P' \equiv D[P], P\ \mathcal{R}\ Q, D[Q] \equiv Q'\}\ .$$

Let $\mathcal{F}$ be the monotonic function associated with (one step) weak bisimulation.

1. *Analyze the one-step transitions of a process $D[P]$ as a function of the transitions of P and $D[0]$.*
2. *Show that $\mathcal{H}$ preserves $<_{\mathcal{F}}$.*
3. *Explicit the associated notion of bisimulation up to context.*

Exercise 417 (prime factorization) *Let A be a set of actions with generic elements $a, b, \ldots$ and let P denote a process in the following fragment of CCS:*

$$P ::= \mathbf{0} \mid a.P \mid (P + P) \mid (P \mid P) \qquad a \in A\ .$$

[1] This is related to (but slightly different from) the *immediate termination* predicate introduced for the language $\mathsf{Imp}_{\|}$ (chapter 19).

Notice that there is no notion of co-action and therefore no possibility of synchronization among parallel processes. In this exercise when we speak of a process, we refer to a process in this fragment. We define the size of a process*, say $|P|$, by induction on the structure of P as follows:*

$$|\mathbf{0}| = 0 \ , \quad |a.P| = 1 + |P| \ , \quad |P + Q| = max(|P|, |Q|) \ , \quad |P \mid Q| = |P| + |Q| \ .$$

We say that a process P is irreducible *if $P \sim P_1 \mid P_2$ implies that $P_1 \sim 0$ or $P_2 \sim 0$ and we say that P is* prime *if P is irreducible and moreover $P \not\sim 0$. Prove the following assertions.*

1. *If $P \xrightarrow{a} Q$ then $|P| > |Q|$.*
2. *If $P \sim Q$ then $|P| = |Q|$ (but the converse fails).*
3. *For all processes P, Q, R the following properties hold:*
   - *If $(P \mid R) \sim (Q \mid R)$ then $P \sim Q$ (this is a kind of* cancellation property*).*
   - *If $(P \mid R) \sim (Q \mid R')$ and $R \xrightarrow{a} R'$ then there exists Q' such that $Q \xrightarrow{a} Q'$ and $P \sim Q'$.*
4. *Every process P such that $P \not\sim 0$ can be expressed up to strong bisimulation as the* parallel composition of prime processes*.*
5. *Every process P such that $P \not\sim 0$ has a* unique decomposition *as the parallel composition of prime processes. Unicity here has to be understood in the same sense as the unicity of the prime factorization of a natural number.*

26.3 A reduction semantics for *CCS*

We want to define a reduction semantics for *CCS*. A technical problem is that in the *syntax*, the synchronizing processes can be far away as in: $\nu b \ (a.P \mid P') \mid \nu c \ \overline{a}.Q$. In the lts presented in Table 26.1 we have tackled this problem by keeping track of the *potential* transitions of every sub-process. An *alternative approach* consists in introducing a notion of *structural equivalence* on processes which is strong enough to bring two synchronizing processes in contiguous positions and weak enough to identify only processes that are intuitively equivalent. In particular in the following, we assume a structural equivalence $\equiv$ which is the least congruence which (i) includes renaming, (ii) such that $\mid$ is associative and commutative, (iii) such that $+$ is associative, commutative and idempotent, and (iv):

$$\begin{array}{ll} \nu a \ (P \mid Q) \equiv \nu a \ P \mid Q & \text{if } a \notin \mathsf{fv}(Q) \\ A(b^*) \equiv [b^*/a^*]P & \text{if } A(a^*) = P \ . \end{array}$$

An *evaluation context* E is defined by:

$$E ::= [\ ] \mid \nu a \ E \mid E \mid P \qquad \text{(evaluation contexts).} \tag{26.4}$$

Then the *reduction relation* is

$$\begin{array}{lll} P \to Q & \text{if} & P \equiv E[a.P' + Q' \mid \overline{a}.P'' + Q''] \text{ and } Q \equiv E[P' \mid P''] \\ P \to Q & \text{if} & P \equiv E[\tau.P + Q'] \text{ and } Q \equiv E[P] \ . \end{array}$$

Exercise 418 (reduction up to structural equivalence) *Check that:*

$$\nu b\ (a.P \mid P') \mid \nu c\ \overline{a}.Q \to \nu b\ (P \mid P') \mid \nu c\ Q\ .$$

Internal transitions and reductions can be related as follows (proofs left to the reader).

Proposition 419 *Let P be a CCS process.*

1. *If $P \xrightarrow{\tau} P'$ then $P \to P'$.*
2. *If $P \to P'$ then $P \xrightarrow{\tau} P''$ and $P' \equiv P''$.*

The next step is to introduce some candidates for the notion of *basic observable*. We write:

$P \downarrow a$ if the process P is 'ready to perform' a visible communication action on channel a.

This is called a *strong commitment* (or *barb*). It is a simple exercise to define $\downarrow$ by induction on the structure of P. One may also distinguish the *polarity* of the communication (input or output).

Following the notation adopted for weak transitions, we denote with $\Rightarrow$ the reflexive and transitive closure of the reduction relation $\to$ (elsewhere in these lecture notes the notation $\xrightarrow{*}$ is used). Then we derive a notion of *weak* commitment by writing:

$$P \Downarrow a \text{ if } P \Rightarrow Q \text{ and } Q \downarrow a\ .$$

We also have the following strong and weak commitment predicates which *abstract* the channel on which the process commits:

$$\begin{array}{lll} P \downarrow_\exists & \text{if} & \exists a\ \ P \downarrow a\ , \\ P \Downarrow_\exists & \text{if} & \exists a\ \ P \Downarrow a\ . \end{array}$$

Finally, we introduce the following strong and weak predicates to observe termination:

$$\begin{array}{lll} P \downarrow & \text{if} & P \not\to\ , \\ P \Downarrow & \text{if} & \exists P'\ \ P \Rightarrow P' \text{ and } P' \downarrow\ . \end{array}$$

Definition 420 (static contexts) *We define the static contexts as the contexts of the following shape:*

$$C ::= [\ ] \mid (C \mid P) \mid \nu a\ C\ .$$

Intuitively, they are called *static* because they persist after a transition (unlike a prefix or a sum). It is generally held that a useful equivalence should be preserved at least by static contexts. Incidentally, in the simple case considered, static contexts coincide with evaluation contexts. Next we introduce a notion of compositional equivalence which is based on the notion of bisimulation.

Definition 421 (contextual bisimulation) *A binary relation $\mathcal{R}$ on processes is a* strong contextual bisimulation *if whenever $P\ \mathcal{R}\ Q$ the following conditions hold (and reciprocally for Q):*

(cxt) *For all* static contexts C, $C[P]\ \mathcal{R}\ C[Q]$.

(red) *If $P \to P'$ then for some Q', $Q \to Q'$ and $P' \mathcal{R} Q'$.*

(cmt) *If $P \downarrow a$ then $Q \downarrow a$.*

For the weak *version replace $\to$ by $\Rightarrow$ and $\downarrow$ by $\Downarrow$. For the one-step weak version the replacement only takes place on the right of the implication. Denote with $\sim_C$ ($\approx_C$) the largest contextual (weak) bisimulation.*

Informally, we can say that a contextual bisimulation is a relation that is preserved by static contexts and by reduction, and that is compatible with commitments. If we drop the preservation by static contexts we obtain the following notion.

Definition 422 (barbed bisimulation) *A binary relation $\mathcal{R}$ on processes is a* strong barbed bisimulation *if whenever $P \mathcal{R} Q$ the following conditions hold (and reciprocally for Q):*

(red) *If $P \to P'$ then for some Q', $Q \to Q'$ and $P' \mathcal{R} Q'$.*

(cmt) *If $P \downarrow a$ then $Q \downarrow a$.*

We denote with $\sim_{BB}$ the largest such equivalence, and with $\approx_{BB}$ its weak variant.

Barbed bisimulation distinguishes less processes than contextual bisimulation and it is not preserved by parallel composition.

Exercise 423 (on barbed bisimulation) *In the framework of CCS, show that barbed bisimulation is* not *preserved by parallel composition.*

Because preservation by the operators of the language is essential for compositional reasoning, the notion of barbed bisimulation can be refined as follows.

Definition 424 (barbed equivalence) *We say that two processes are* barbed equivalent *if put in any static context they are barbed bisimilar. We denote such equivalence with $\sim_{BE}$. The weak variant based on weak barbed bisimulation is denoted with $\approx_{BE}$.*

Then the comparison of contextual bisimulation and barbed equivalence arises as an obvious question.

Exercise 425 (on barbed equivalence) *Show that if two processes are contextually bisimilar then they are barbed equivalent.*

The converse can be quite tricky to prove. The characterization of labelled bisimulation we are aiming at is more direct/natural when working with *contextual bisimulation* than with *barbed equivalence.*

Exercise 426 (variations on commitment) *Show that we get an equivalent notion of contextual bisimulation if the condition [cmt] is replaced by: $P \Downarrow_\exists$ implies $Q \Downarrow_\exists$. On the other hand, show that we get an incomparable notion of contextual bisimulation if the condition [cmt] is replaced by: $P \Downarrow$ implies $Q \Downarrow$.*

The labelled bisimulation introduced in section 26.2 is an example of contextual bisimulation.

Proposition 427 *The largest labelled bisimulation is a contextual bisimulation (both in the strong and weak case).*

PROOF. Denote with $\sim$ ($\approx$) the labelled (weak) bisimulation. It has been proved that $\sim$ and $\approx$ are preserved by static contexts. Incidentally, note that an *arbitrary* labelled bisimulation does not need to be saturated by static contexts. Moreover the conditions [red] and [commit] of contextual bisimulation are particular cases of the bisimulation game in the labelled case. □

The previous proposition shows that every labelled bisimulation is contained in contextual bisimulation. The converse is given by the following.

Proposition 428 *The largest contextual bisimulation $\sim_C$ (or $\approx_C$ in the weak case) is a labelled (weak) bisimulation.*

PROOF. We consider directly the *weak case*. An *internal choice*[2] in *CCS* can be defined as follows:

$$P \oplus Q = \nu a\ (a.P \mid a.Q \mid \overline{a}) = \tau.P + \tau.Q\ .$$

If $P \approx_C Q$ and $P \stackrel{\tau}{\Rightarrow} P'$ then $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P' \approx_C Q'$, by condition [red] of contextual bisimulation.

So suppose $P \stackrel{a}{\Rightarrow} P'$. Let o_1, o_2 be two distinct fresh names (not in P and Q) and define the static context:

$$C = [\ ] \mid \overline{a}.(o_1 \oplus (o_2 \oplus 0))\ .$$

By hypothesis, $C[P] \approx_C C[Q]$. Clearly, $C[P] \stackrel{\tau}{\Rightarrow} P' \mid (o_2 \oplus 0)$ and again by hypothesis (condition [red]) $C[Q] \stackrel{\tau}{\Rightarrow} Q''$ and $P'' \equiv P' \mid (o_2 \oplus 0) \approx_C Q''$.

Now we argue that Q'' must be of the shape $Q' \mid (o_2 \oplus 0)$ where $Q \stackrel{a}{\Rightarrow} Q'$. The case $Q'' = C[Q']$ and $Q \stackrel{\tau}{\Rightarrow} Q'$ is impossible because $P'' \Downarrow o_2$ entails $Q' \Downarrow o_2$, and the latter entails $Q' \Downarrow o_1$ which cannot be matched by P''. The cases $Q \stackrel{a}{\Rightarrow} Q'$ and $Q'' = Q' \mid R'$ where $R' \in \{o_1 \oplus (o_2 \oplus 0), o_1, o_2, 0\}$ are also impossible for similar reasons. Thus we must have $Q'' = Q' \mid (o_2 \oplus 0)$ and $P' \mid (o_2 \oplus 0) \approx_C Q''$.

It is easy to argue that since $P' \mid (o_2 \oplus 0) \stackrel{\tau}{\Rightarrow} P' \mid 0$ we must have $Q' \mid (o_2 \oplus 0) \stackrel{\tau}{\Rightarrow} Q_1 \mid 0$ and $Q_1 \mid 0 \approx_C P' \mid 0$. Thus $Q \stackrel{a}{\Rightarrow} Q_1$ and $P' \equiv (P' \mid 0) \approx_C (Q_1 \mid 0) \equiv Q_1$. Strictly speaking, we use an *up to technique*. □

Exercise 429 *Do the previous proof in the strong case. Can you simplify the context C in this case? Show that (weak) labelled bisimulation is a (weak) barbed bisimulation. Show that (weak) labelled bisimulation is a (weak) barbed equivalence.*

[2] It is called internal because the environment has no way of controlling it; by opposition, a choice such as $a.P + b.Q$ is called external.

26.4 Value-passing *CCS*

As already mentioned, in *CCS* communication is pure synchronization. We now consider an extension where *values* can be sent along channels. In the following, values are just basic atomic objects such as booleans or integers which can be tested for equality:

$$v ::= v_0 \mid v_1 \mid v_2 \ldots \qquad \text{(values)}$$

In chapter 30, we shall consider the more complex case where the values are actually channels. Let *Val* be the set of values. Then the collection of actions given by equation (26.1) is revised as follows:

$$Act = \{av, \overline{a}v \mid a \in A, v \in Val\} \cup \{\tau\} \qquad \text{(actions with value passing)} \tag{26.5}$$

Input and output actions are now pairs composed of a channel name and a value. To write value-passing *CCS* processes, we need a notion of *variable* ranging over values which stands for the value read upon communication.

$$id ::= x \mid y \mid \ldots \qquad \text{(variables)}$$

We also need a notion of *term* which is either a value or a variable (we call this *term* by analogy with first-order logic):

$$t ::= v \mid id \qquad \text{(term)}$$

Then the syntax of *CCS* value-passing processes is as follows:

$$P \quad ::= 0 \mid a(id).P \mid \overline{a}t.P \mid [t = t]P, P \mid (P + P) \mid (P \mid P) \mid \nu a\ P \mid A(a^*)$$

where $a(x).P$ is the process that receives a value v on the channel a and becomes $[v/x]P$, $\overline{a}v.P$ is the process that sends v on a and becomes P, and $[v = v']P, Q$ is the process that compares the values v and v' and runs P if they are equal and Q otherwise.

The reduction semantics for *CCS* is easily extended to value passing *CCS*. Omitting the details concerning the evaluation context and the structural equivalence, the synchronization rule with exchange of values is:

$$\overline{a}v.P \mid a(x).Q \to P \mid [v/x]Q\ ,$$

and we add the usual rules for the conditional (cf. chapter 19):

$$[v = v]P, Q \to P \qquad [v = v']P, Q \to Q \quad (v \neq v')\ .$$

Notice that the resulting reduction semantics is supposed to operate on terms *without* free value variables. As a matter of fact, reducing processes with free variables would be a form of *symbolic execution* and requires carrying along with the process a set of constraints which describe the possible values of its free variables.

The labelled semantics of *CCS* with value passing rises some subtle issues concerning the treatment of the input prefix. Consider a process $a(x).P$. The action structure we have given above in equation (26.5) suggests a rule of the shape:

$$\frac{}{a(x).P \xrightarrow{av} [v/x]P} \qquad \text{(early binding)}\ . \tag{26.6}$$

However, by changing a little bit the action structure (26.5), we could also think of a rule that maps a process to a *function* from values to processes:

$$\frac{}{a(x).P \xrightarrow{a} \lambda x.P} \qquad \text{(late binding)} \ . \tag{26.7}$$

This in turn requires defining an obvious notion of bisimulation on functions: two functions $\lambda x.P$ and $\lambda x.Q$ from values to processes are bisimilar if for all values $v \in \mathit{Val}$, $[v/x]P$ and $[v/x]Q$ are bisimilar (cf. chapter 9). The first rule is called *early binding* and formalizes a situation where the communication channel and the value received are selected at the same time. By opposition, the second rule is called *late binding*. It turns out that the late binding approach leads to a labelled bisimulations which is more discriminating than the one based on early binding. For instance, consider the processes:

$$\begin{aligned} P &\equiv a(x).([x = v_0]\overline{b}.0, 0 + [x = v_1]\overline{c}.0, 0) \ , \\ Q &\equiv a(x).[x = v_0]\overline{b}.0, 0 + a(x).[x = v_1]\overline{c}.0, 0 \ . \end{aligned}$$

The processes P and Q are 'early-binding bisimilar' but not 'late-binding bisimilar'. Specifically, by a late-binding input the process P goes to a function that cannot be matched by Q. In this case, the comparison with contextual bisimulation suggests that the early binding semantics is the 'right' one.

We conclude this quick review of value passing *CCS* by mentioning that at the price of an *infinitary* syntax, it is quite simple to reduce it to ordinary *CCS*. This is similar in spirit to transformations from predicate logic to propositional logic where universal and existential quantifications are replaced by infinitary conjunctions and disjunctions, respectively. In our case, the basic idea is to replace the input of a value by the non-deterministic sum of infinitely many inputs:

$$[\![a(x).P]\!] = \Sigma_{v \in \mathit{Val}} a_v.[\![[v/x]P]\!] \ . \tag{26.8}$$

Incidentally, for a finite and small set of values this gives an effective way of programming value passing in basic *CCS*.

26.5 Summary and references

Labelled bisimulation requires: labels, labelled transitions, and labelled bisimulation. The choice of the labels and the rules of the bisimulation game may be hard to justify. On the other hand, contextual bisimulation requires reduction, static contexts, and commitments. This approach is more natural but it may be harder to prove that two processes are contextual bisimilar. For *CCS*, labelled bisimulation coincides with contextual bisimulation. In general this kind of result is a guideline when we are confronted to more complicated models (such as the π-calculus in chapter 30).

CCS is a model of message passing based on redez-vous communication among two processes. Another popular interaction mechanism consists in allowing several parallel processes to synchronize on the same label. This mechanism does not scale so well when we want to add more structure to the actions as, *e.g.*, in value passing synchronization.

CCS has been introduced by Milner in [Mil80]; a revised presentation is in [Mil95]. The reduction semantics of concurrent systems is put forward in [BB92]. The notion of *contextual bisimulation* is studied by [HY95]. The earlier definition of *barbed equivalence* can be found in [MS92]. Exercise 417 is based on [MM93].

Chapter 27

Determinacy and confluence

In automata theory, one can consider various definitions of determinism. For instance, in the framework of *finite* automata, consider the following ones.

1. There is no *word* w that admits two computation paths in the graph such that one leads to an accepting state and the other to a non-accepting state.
2. Each *reachable configuration* admits at most one successor.
3. For each *state*, either there is exactly one outgoing transition labelled with ϵ, or all outgoing transitions are labelled with distinct symbols of the input alphabet.

Thus one can go from *'extensional' conditions* (intuitive but hard to verify) to *'syntactic' conditions* (verifiable but not as general). In the following, we propose a definition of deterministic lts and show that all the equivalences included between trace equivalence and bisimulation collapse on such systems. We also introduce a notion of *confluence* on labelled transition systems. This is a stronger property than determinism which allows for a restricted form of parallel composition and for the representation of deterministic models of parallel computation such as *Kahn networks*. Finally, we consider *reactive* systems, *i.e.*, systems which enjoy a kind of generalized termination property. It turns out that for such system, it is enough to check a *local* form of confluence.

27.1 Determinism in lts

In the first place, it is useful to recall why non-determinism is needed. First, it arises naturally in *race conditions* where two 'clients' request the same service such as:

$$\nu a\ (\overline{a}.P_1 \mid \overline{a}.P_2 \mid a)\ .$$

Second, it is a tool for general specification and portability. It is often the case that we do not want to commit on a particular behavior. For instance, consider:

$$\nu a, b\ (\tau.\overline{a}.\overline{b}.\overline{c}\ \mid\ a.\overline{b}.\overline{d}\ \mid\ b)\ .$$

Depending on the compilation, the design of the virtual machine, the processors timing,... we might always run $\overline{d}$ rather than $\overline{c}$ (or the other way around).

On the other hand, deterministic systems are easier to test, debug, and possibly prove correct. Notice that often the implementation seems 'deterministic' because the scheduler determinizes the program's behavior. However this kind of determinism is *not portable*: running the program in another environment may produce different results.

We now move towards a definition of determinacy. Here are some reasonable requirements:

- If P and P' are *'equivalent'* then one is determinate if and only if the other is.
- If we run an *'experiment'* twice we always get the same 'result'.
- If P is determinate and we run an experiment then *the residual of* P after the experiment should still be determinate.

If we place ourselves in the context of a simple model such as *CCS*, we can interpret *equivalent* as *weak bisimilar* and *experiment* as a finite sequence of *labelled transitions*.

More formally, let us denote with $\mathcal{L}$ the set of *visible actions and co-actions* with generic elements $\ell, \ell', \ldots$ and let us denote with $Act = \mathcal{L} \cup \{\tau\}$ the set of *actions*, with generic elements $\alpha, \beta, \ldots$ Let $s \in \mathcal{L}^*$ denote a finite word over $\mathcal{L}$. Then:

$$\begin{array}{ll} P \stackrel{\epsilon}{\Rightarrow} P' & \text{if } P \stackrel{\tau}{\Rightarrow} P' \\ P \stackrel{\ell_1 \ldots \ell_n}{\Rightarrow} P',\ n \geq 1 & \text{if } P \stackrel{\ell_1}{\Rightarrow} \cdots \stackrel{\ell_n}{\Rightarrow} P' \ . \end{array}$$

If $P \stackrel{s}{\Rightarrow} Q$ we say that Q is a *derivative* of P. As usual we write $|s|$ for the length of the word s.

Definition 430 *A process P is* determinate *if for any $s \in \mathcal{L}^*$, if $P \stackrel{s}{\Rightarrow} P_i$ for $i = 1, 2$ then $P_1 \approx P_2$.*

Remark 431 *This definition relies on the notion of* labelled transition system*. Indeed, the transition $P \stackrel{\ell}{\rightarrow} P'$, ℓ represents a* minimal *interaction with the environment and P' is the* residual *after the interaction.*

Exercise 432 *Are the following CCS processes determinate? (1) $a.(b+c)$. (2) $a.b + ac$. (3) $a + a.\tau$. (4) $a + \tau.a$. (5) $a + \tau$.*

Proposition 433 *The following properties hold:*

1. *If P is determinate and $P \stackrel{\alpha}{\rightarrow} P'$ then P' is determinate.*
2. *If P is determinate and $P \approx P'$ then P' is determinate.*

Proof. Like most of the following proofs, the argument is by *diagram chasing*.

1. Suppose $P \stackrel{\alpha}{\rightarrow} P'$ and $P' \stackrel{s}{\Rightarrow} P_i$ for $i = 1, 2$.
   - If $\alpha = \tau$ then $P \stackrel{s}{\Rightarrow} P_i$ for $i = 1, 2$. Hence $P_1 \approx P_2$.
   - If $\alpha = \ell$ then $P \stackrel{\ell \cdot s}{\Rightarrow} P_i$ for $i = 1, 2$. Hence $P_1 \approx P_2$.
2. Suppose $P \approx P'$ and $P' \stackrel{s}{\Rightarrow} P'_i$ for $i = 1, 2$.

- By definition of *weak bisimulation*: $P \stackrel{s}{\Rightarrow} P_i$ and $P_i \approx P'_i$, for $i = 1, 2$.
- Since P is *determinate*, we have $P_1 \approx P_2$.
- Therefore, we conclude by *transitivity* of $\approx$: $P'_1 \approx P_1 \approx P_2 \approx P'_2$. □

Definition 434 (τ-inertness) *We say that a process P is* τ-inert *if for all its derivatives Q, if $Q \stackrel{\tau}{\Rightarrow} Q'$ then $Q \approx Q'$.*

Proposition 435 *If a process is determinate then it is τ-inert.*

Proof. Suppose $P \stackrel{s}{\Rightarrow} Q$ and $Q \stackrel{\tau}{\Rightarrow} Q'$. Then $P \stackrel{s}{\Rightarrow} Q$ and $P \stackrel{s}{\Rightarrow} Q'$. Thus by determinacy, $Q \approx Q'$. □

Next we introduce a *weak version* of the notion of trace equivalence for lts presented in definition 358.

Definition 436 (traces) *We define the* traces *of a process P as:*

$$tr(P) = \{s \in \mathcal{L}^* \mid P \stackrel{s}{\Rightarrow} \cdot\} \ ,$$

and say that two processes P, Q are trace equivalent *if $tr(P) = tr(Q)$.*

Notice that the traces of a process form a *non-empty*, *prefix-closed* set of *finite words* over $\mathcal{L}$.

Exercise 437 *Are the following equations valid for* trace equivalence *and/or* weak bisimulation*?*

$$a + \tau = a, \quad \alpha.(P + Q) = \alpha.P + \alpha.Q, \quad (P + Q) \mid R = P \mid R + Q \mid R, \quad P = \tau.P \ .$$

Exercise 438 (compositionality of trace semantics) *Show that if P, Q, R are CCS processes and $tr(P) = tr(Q)$ then $tr(P \mid R) = tr(Q \mid R)$.*

The following result entails that on deterministic processes most equivalences (trace, simulation-induced equivalence, bisimulation,...) collapse.

Proposition 439 *Let P, Q be processes.*

1. *If $P \approx Q$ then $tr(P) = tr(Q)$.*
2. *Moreover, if P, Q are* determinate *then $tr(P) = tr(Q)$ implies $P \approx Q$.*

Proof. (1) Suppose $P \approx Q$ and $P \stackrel{s}{\Rightarrow} \cdot$. Then $Q \stackrel{s}{\Rightarrow} \cdot$ by induction on $|s|$ using the properties of weak bisimulation.

(2) Suppose P, Q determinate and $tr(P) = tr(Q)$. We show that:

$$\{(P, Q) \mid tr(P) = tr(Q)\}$$

is a bisimulation.

- If $P \xrightarrow{\tau} P'$ then $P \approx P'$ by *determinacy*. Thus taking $Q \stackrel{\tau}{\Rightarrow} Q$ we have:
$$P' \approx P \qquad tr(P) = tr(Q)\ .$$
By (1), we conclude: $tr(P') = tr(P) = tr(Q)$.

- If $P \xrightarrow{\ell} P'$ then we note that:
$$tr(P) = \{\epsilon\} \cup \{\ell\} \cdot tr(P') \cup \bigcup_{\ell \neq \ell', P \stackrel{\ell'}{\Rightarrow} P''} \{\ell'\} \cdot tr(P'')\ .$$
This is because all the processes P' such that $P \stackrel{\ell}{\Rightarrow} P'$ are bisimilar, hence trace equivalent. A similar reasoning applies to $tr(Q)$. Thus there must be a Q' such that $Q \stackrel{\ell}{\Rightarrow} Q'$ and $tr(P') = tr(Q')$. □

Example 440 (the unbounded buffer reconsidered) *Recall the unbounded buffer in example 409:*
$$\begin{array}{ll} Buf(a,b) & = a.\nu c\ (Buf(a,c) \mid \overline{b}.Buf(c,b)) \\ P(n) & = \nu a\ (\overline{a}^n \mid Buf(a,b)) \end{array}$$
One can show that these processes are deterministic*. In fact one can show that they actually enjoy a stronger property known as* confluence *which is introduced next in section 27.2.*

27.2 Confluence in lts

We introduce a notion of *confluence* that strengthens determinacy and is preserved by some form of communication (parallel composition + restriction). For instance,
$$\nu a\ ((a + b) \mid \overline{a})$$
will be *rejected* because $a + b$ is *not* confluent (while being *deterministic*).

The notion of confluence we consider is reminiscent of *confluence* in rewriting systems (cf. definition 23). By analogy, one calls confluence the related theory in process calculi but bear in mind that: (1) confluence is relative to a *labelled transition system* and (2) we close diagrams *up to equivalence*.

Before introducing formally the notion of confluence for lts we need to define a notion of action difference.

Definition 441 (action difference) *Suppose $\alpha, \beta \in Act$. Their* action difference $\alpha\backslash\beta$ *is defined as:*
$$\alpha\backslash\beta = \begin{cases} \alpha & \text{if } \alpha \neq \beta \\ \tau & \text{otherwise.} \end{cases}$$

We can generalize the notion of action difference to sequences of visible actions $r, s \in \mathcal{L}^*$. To compute the *difference* $r\backslash s$ of r by s we scan r from left to right deleting each label which occurs in s taking into account the multiplicities (cf. difference of *multi-sets*). We abuse notation by writing $\ell \notin s$ to mean that ℓ does not occur in the word s.
$$\begin{array}{ll} (\epsilon\backslash s) & = \epsilon \\ (\ell r\backslash s) & = \begin{cases} \ell \cdot (r\backslash s) & \text{if } \ell \notin s \\ r\backslash(s_1 \cdot s_2) & \text{if } s = s_1 \ell s_2, \ell \notin s_1\ . \end{cases} \end{array}$$
For instance: $aba\backslash ca = ba$ and $ca\backslash aba = c$.

Exercise 442 *Let $r, s, t \in \mathcal{L}^*$. Show that:*

1. $(rs)\backslash(rt) = s\backslash t$.
2. $r\backslash(st) = (r\backslash s)\backslash t$.
3. $(rs)\backslash t = (r\backslash t)(s\backslash(t\backslash r))$.

We now introduce a notion of confluent process.

Definition 443 (confluence) *A process P is* confluent *if* for every derivative Q *of P we have:*

$$\frac{Q \stackrel{\alpha}{\Rightarrow} Q_1 \quad Q \stackrel{\beta}{\Rightarrow} Q_2}{\exists\, Q_1', Q_2' \;\; (Q_1 \stackrel{\beta\backslash\alpha}{\Rightarrow} Q_1', \quad Q_2 \stackrel{\alpha\backslash\beta}{\Rightarrow} Q_2', \;\; \textit{and} \quad Q_1' \approx Q_2')} \quad \textit{[conf 0]} \tag{27.1}$$

The condition in definition 443 is labelled as [conf 0] to distinguish it from two more equivalent conditions that we state below and that are labelled [conf 1] and [conf 2].

- A process P is *confluent 1* if if *for every derivative* Q of P we have:

$$\frac{Q \stackrel{\alpha}{\rightarrow} Q_1 \quad Q \stackrel{\beta}{\Rightarrow} Q_2}{\exists\, Q_1', Q_2' \;\; (\; Q_1 \stackrel{\beta\backslash\alpha}{\Rightarrow} Q_1', \quad Q_2 \stackrel{\alpha\backslash\beta}{\Rightarrow} Q_2', \;\; \text{and} \quad Q_1' \approx Q_2' \;)} \quad \text{[conf 1]} \tag{27.2}$$

- A process P is *confluent 2* if for all $r, s \in \mathcal{L}^*$ we have:

$$\frac{P \stackrel{r}{\Rightarrow} P_1 \qquad P \stackrel{s}{\Rightarrow} P_2}{\exists\, P_1', P_2' \;\; (\; P_1 \stackrel{s\backslash r}{\Rightarrow} P_1', \quad P_2 \stackrel{r\backslash s}{\Rightarrow} P_2', \;\; \text{and} \quad P_1' \approx P_2' \;)} \quad \text{[conf 2]} \tag{27.3}$$

Remark 444 *In conditions [conf 0] and [conf 1] if $\alpha = \beta$ then we close the diagram with τ actions only.*

A first *sanity check* is to verify that the confluent processes are invariant under *transitions* and *equivalence* (cf. proposition 433).

Proposition 445 *The following properties hold:*

1. *If P is confluent and $P \stackrel{\alpha}{\rightarrow} P'$ then P' is confluent.*
2. *If P is confluent and $P \approx P'$ then P' is confluent.*

Proof. (1) If Q is a derivative of P' then it is also a derivative of P.

(2) It is enough to apply the fact that:

$$(P \approx P' \text{ and } P \stackrel{\alpha}{\Rightarrow} P_1) \;\text{ implies }\; \exists\, P_1' \;\; (P' \stackrel{\alpha}{\Rightarrow} P_1' \text{ and } P_1 \approx P_1')$$

and the transitivity of $\approx$. □

Confluence implies τ-inertness, and from this we can show that it implies determinacy too.

Proposition 446 *Suppose P is* confluent. *Then P is: (1) τ-*inert *and (2)* determinate.

Proof. First a reminder. A relation R is a *weak bisimulation up to* $\approx$ if:

$$\frac{P\ R\ Q \quad P \stackrel{\alpha}{\Rightarrow} P'}{\exists Q'\ \ Q \stackrel{\alpha}{\Rightarrow} Q' \text{ and } P'(\approx \circ R \circ \approx)Q'}$$

(and symmetrically for Q). It is important that we work with the *weak moves* on both sides, otherwise the relation R is *not* guaranteed to be contained in $\approx$ (cf. exercise 384). Now we move to the proof.

1. We want to show that $P \stackrel{\tau}{\Rightarrow} Q$ implies $P \approx Q$. We show that:

   $$R = \{(P, Q) \mid P \stackrel{\tau}{\Rightarrow} Q\}$$

   is a weak bisimulation up to $\approx$. It is clear that whatever Q does, P can do too with some extra moves. In the other direction, suppose, *e.g.*, $P \stackrel{\alpha}{\Rightarrow} P_1$ with $\alpha \neq \tau$ (case $\alpha = \tau$ left as exercise). By [conf 0], $Q \stackrel{\alpha}{\Rightarrow} Q_1$, $P_1 \stackrel{\tau}{\Rightarrow} P_2$, and $Q_1 \approx P_2$. That is: $P_1(R \circ \approx)Q_1$.

2. We want to show that if P is confluent then it is determinate. Suppose $P \stackrel{s}{\Rightarrow} P_i$ for $i = 1, 2$ and $s \in \mathcal{L}^*$. We proceed by *induction* on $|s|$. If $|s| = 0$ and $P \stackrel{\tau}{\Rightarrow} P_i$ for $i = 1, 2$ then *by τ-inertness* $P_1 \approx P \approx P_2$. For the inductive case, suppose $P \stackrel{\ell}{\Rightarrow} P_i' \stackrel{r}{\Rightarrow} P_i$ for $i = 1, 2$. By confluence and τ-inertness, we derive that $P_1' \approx P_2'$. By *weak bisimulation*, $P_2' \stackrel{r}{\Rightarrow} P_2''$ and $P_2'' \approx P_1$. By *inductive hypothesis*, $P_2 \approx P_2''$. Thus $P_2 \approx P_2'' \approx P_1$ as required. □

Exercise 447 *We have seen that confluence implies determinacy which implies τ-inertness. Give examples that show that these implications cannot be reversed.*

We now turn to the confluence 1 definition which is 'asymmetric' in that the move from Q to Q_1 just concerns a *single* action.

Proposition 448 (conf 1) *A process P is confluent iff for every derivative Q of P, it satisfies condition [conf 1].*

Proof. The diagrams of [conf 1] are a particular case of [conf 0]. Thus we just have to show that the diagrams of [conf 1] suffice to complete the diagrams of [conf 0].

We may proceed by induction on the length of the transition $Q \stackrel{\alpha}{\Rightarrow} Q_1$. For instance, suppose $\alpha \neq \beta$, $\beta \neq \tau$, and:

$$Q \stackrel{\tau}{\rightarrow} Q_1 \stackrel{\alpha}{\Rightarrow} Q_2\ , \quad Q \stackrel{\beta}{\Rightarrow} Q_3\ .$$

Then we derive:

$Q_1 \stackrel{\beta}{\Rightarrow} Q_4$	$Q_3 \stackrel{\tau}{\Rightarrow} Q_5$	$Q_4 \approx Q_5$	(by [conf 1])
$Q_2 \stackrel{\beta}{\Rightarrow} Q_6$	$Q_4 \stackrel{\alpha}{\Rightarrow} Q_7$	$Q_4 \approx Q_7$	(by inductive hypothesis)
$Q_5 \stackrel{\alpha}{\Rightarrow} Q_8$	$Q_7 \approx Q_8$		(from $Q_4 \approx Q_5$ and $Q_4 \stackrel{\alpha}{\Rightarrow} Q_7$).

Therefore: $Q_2 \stackrel{\beta}{\Rightarrow} Q_6$, $Q_3 \stackrel{\alpha}{\Rightarrow} Q_8$, and $Q_6 \approx Q_8$ as required. □

Exercise 449 *Consider another case of the proof. For instance, when* $Q \stackrel{\alpha}{\to} Q_1 \stackrel{\tau}{\Rightarrow} Q_2$.

We turn to condition [conf 2].

Proposition 450 (conf 2) *A process* P *is* confluent *iff it satisfies [conf 2].*

PROOF. ($\Leftarrow$) It suffices to check that if P has property [conf 2] then its derivatives have it too.

- Suppose $P \stackrel{t}{\Rightarrow} Q$ for $t \in \mathcal{L}^*$.
- Suppose further $Q \stackrel{r}{\Rightarrow} Q_1$ and $Q \stackrel{s}{\Rightarrow} Q_2$.
- By composing diagrams and applying [conf 2] we get:

$$Q_1 \stackrel{(ts\backslash tr)}{\Rightarrow} Q_1' \quad Q_2 \stackrel{(tr\backslash ts)}{\Rightarrow} Q_2' \quad Q_1' \approx Q_2' \ .$$

- By exercise 442, $ts\backslash tr = s\backslash r$ and $tr\backslash ts = r\backslash s$. Then we derive:

$$Q_1 \stackrel{(s\backslash r)}{\Rightarrow} Q_1' \quad Q_2 \stackrel{(r\backslash s)}{\Rightarrow} Q_2' \quad Q_1' \approx Q_2' \ .$$

($\Rightarrow$) We proceed in three steps.

1. By induction on $|s|$ we show that:

$$\frac{P \stackrel{\tau}{\Rightarrow} P_1 \quad P \stackrel{s}{\Rightarrow} P_2}{\exists\, P_1', P_2' \;\; P_1 \stackrel{s}{\Rightarrow} P_1', \quad P_2 \stackrel{\tau}{\Rightarrow} P_2', \text{ and } \quad P_1' \approx P_2'} \ .$$

2. Then, again by induction on $|s|$, we show that:

$$\frac{P \stackrel{\ell}{\Rightarrow} P_1 \quad P \stackrel{s}{\Rightarrow} P_2}{\exists\, P_1', P_2' \;\; P_1 \stackrel{s\backslash \ell}{\Rightarrow} P_1', \quad P_2 \stackrel{\ell\backslash s}{\Rightarrow} P_2', \quad \text{and } P_1' \approx P_2'} \ .$$

3. Finally we prove the commutation of diagram [conf 2] by induction on $|r|$ when $P \stackrel{r}{\Rightarrow} P_1$.

□

Exercise 451 *Complete the proof.*

Next, we return to the issue of building confluent (and therefore determinate) processes.

Proposition 452 (building confluent processes) *If* P, Q *are confluent processes then so are: (1)* 0, $\alpha.P$, *(2)* $\nu a\ P$, *and (3)* σP *where* σ *is an injective substitution on the free names of* P.

PROOF. Routine analysis of transitions (cf. similar statement for determinacy). □

Remark 453 (on sum) *In general,* $a + b$ *is* determinate *but it is* not confluent *for* $a \neq b$.

Definition 454 (sorting) *Let P be a process. We define its* sorting $\mathcal{L}(P)$ *as the set:*

$$\{\ell \in \mathcal{L} \mid \exists\, s \in \mathcal{L}^*\ \ P \stackrel{s}{\Rightarrow} Q \stackrel{\ell}{\rightarrow} \cdot\}\ .$$

Exercise 455 *With reference to exercise 440, show that $\mathcal{L}(a \mapsto b) = \{a, \overline{b}\}$.*

Definition 456 (restricted composition) *A restricted composition is a process of the shape: $\nu a_1, \ldots, a_n\ (P \mid Q)$ where:*

1. *P and Q do not share visible actions: $\mathcal{L}(P) \cap \mathcal{L}(Q) = \emptyset$.*
2. *P and Q may interact only on the restricted names:*

$$\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \subseteq \{a_1, \ldots, a_n\} \cup \{\overline{a}_1, \ldots, \overline{a}_n\}\ .$$

Proposition 457 *Confluence is preserved by restricted composition.*

Proof. We abbreviate $\nu a_1, \ldots, a_n\ (P \mid Q)$ as $\nu a^*\ (P \mid Q)$. First we observe that any derivative of $\nu a^*\ (P \mid Q)$ will have the shape $\nu a^*\ (P' \mid Q')$ where P' is a derivative of P and Q' is a derivative of Q.

Since sorting is preserved by transitions, the two conditions on sorting in definition 454 will be satisfied. Therefore, it is enough to show that the diagrams in [conf 1] commute for processes of the shape $R = \nu a^*\ (P \mid Q)$ under the given hypotheses.

We consider one case. Suppose: $R \stackrel{\ell}{\rightarrow} \nu a^*\ (P_1 \mid Q)$ because $P \stackrel{\ell}{\rightarrow} P_1$. Also assume: $R \stackrel{\ell}{\Rightarrow} \nu a^*\ (P_2 \mid Q_2)$ because $P \stackrel{s\ell r}{\Rightarrow} P_2$ and $Q \stackrel{\overline{s}\cdot\overline{r}}{\Rightarrow} Q_2$ with $s \cdot r \in \{a^*, \overline{a^*}\}^*$ and $\ell \notin \{a^*, \overline{a^*}\}$.

Since P is confluent we have:

$$\frac{P \stackrel{\ell}{\rightarrow} P_1 \quad P \stackrel{s\ell r}{\Rightarrow} P_2}{\exists\, P_1', P_2'\ \ P_1 \stackrel{sr}{\Rightarrow} P_1', \quad P_2 \stackrel{\tau}{\Rightarrow} P_2', \text{ and } \quad P_1' \approx P_2'}\ .$$

Then we have:

$$\nu a^*\ (P_1 \mid Q) \stackrel{\tau}{\Rightarrow} \nu a^*\ (P_1' \mid Q_2) \approx \nu a^*\ (P_2' \mid Q_2)\ ,$$

thus closing the diagram (note that we use the congruence properties of $\approx$). □

Exercise 458 *Consider other cases of the proof, for instance:*

$$\begin{array}{lll} \nu a\ (P \mid Q) \stackrel{\tau}{\rightarrow} \nu a\ (P \mid Q) & \text{as } P \stackrel{a}{\rightarrow} P_1, & Q \stackrel{\overline{a}}{\rightarrow} Q_1\ , \\ \nu a\ (P \mid Q) \stackrel{\tau}{\Rightarrow} \nu a\ (P_2 \mid Q_2) & \text{as } P \stackrel{s}{\rightarrow} P_2, & Q \stackrel{\overline{s}}{\Rightarrow} Q_2\ . \end{array}$$

27.3 Kahn networks

Kahn networks are a deterministic model of parallel computation where communication is point-to-point, *i.e.*, for every channel there is at most one sender and one receiver, and channels are order preserving buffers of unbounded capacity, *i.e.*, sending is non blocking and the order of emission is preserved at the reception.

In this model, each (sequential) process may:

1. perform arbitrary *sequential deterministic computation*,

2. *insert a message in a buffer*,

3. *receive a message from a buffer*. If the buffer is empty then the process *must* suspend,

However, a process *cannot* try to receive a message from several channels at once. In a nutshell Kahn's approach to the semantics of such systems is as follows. First, we regard the unbounded buffers as finite or infinite words over some data domain and second, we model the nodes of the network as functions over words. Kahn observes that the associated system of equations has a least fixed point which defines the semantics of the whole system.

Kahn networks are an important (practical) case where *parallelism* and *determinism* coexist without producing race conditions. For instance, they are frequently used in the *signal processing* community. Our modest goal is to formalize Kahn networks as a fragment of *CCS* and to apply the developed theory to show that the fragment is confluent and therefore deterministic.

We will work with a 'data domain' that contains just one element. The generalization to arbitrary data domains is not difficult, but we would need to formalize determinacy and confluence in the framework of an extended *CCS* where messages carry values (as, *e.g.*, in the value passing *CCS* described in chapter 26). First, let us conclude the analysis of the *unbounded buffers* in *CCS*.

Exercise 459 *With reference to exercises 409, 440, and 455:*

1. *Check that the process $a \mapsto b$ falls in the class of* confluent processes *defined in proposition 457. In particular:*

$$\mathcal{L}(a \mapsto c) \cap \mathcal{L}(\overline{b}.c \mapsto b) = \emptyset \ , \qquad \mathcal{L}(a \mapsto c) \cap \overline{\mathcal{L}(\overline{b}.c \mapsto b)} \subseteq \{c, \overline{c}\} \ .$$

2. *Derive from proposition 439 that to prove $P(n) = \nu a \ (\overline{a}^n \mid a \mapsto b) \approx \overline{b}^n$ it is enough to check:*

$$tr(\overline{b}^n) = \{\epsilon, \overline{b}, \overline{b}\overline{b}, \ldots, \overline{b}^n\} \ . \tag{27.4}$$

3. *Prove property (27.4).*

We define a class of *CCS* processes sufficient to represent Kahn networks.

Definition 460 (restricted processes) *Let KP be the least set of processes such that $0 \in KP$ and if $P, Q \in KP$ and α is an action then:*

1. *$\alpha.P \in KP$,*

2. *$A(b^*) \in KP$ provided the names b^* are all distinct and A is defined by an equation $A(a^*) = P$ and $P \in KP$.*

3. *$\nu a^* \ (P \mid Q) \in KP$ provided $\mathcal{L}(P) \cap \mathcal{L}(Q) = \emptyset$ and $\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \subseteq \{a^*, \overline{a^*}\}$,*

Exercise 461 *Check that $a \mapsto b$ is a KP process and that Kahn processes are confluent.*

Example 462 *Suppose we have a Kahn network with three nodes, and the following ports and behaviors where we use* ! *for output and* ? *for input.*

Node	*Ports*	*Behaviors*
1	$?a, ?b, ?c, !d, !e, !f$	$A_1 = ?a.!d.!e.?b.?c.!f.A_1$
2	$!b, ?d$	$A_2 = ?d.!b.A_2$
3	$!c, ?e$	$A_3 = ?e.!c.A_3$.

The corresponding CCS system relies on the equations for the buffer process plus:

$$\begin{array}{ll} A_1(a,b,c,d,e,f) & = a.\overline{d}.\overline{e}.b.c.\overline{f}.A_1(a,b,c,d,e,f) \\ A_2(b,d) & = d.\overline{b}.A_2(b,d) \\ A_3(c,e) & = e.\overline{c}.A_3(c,e) \ . \end{array}$$

The sorting is easily derived:

$$\begin{array}{ll} \mathcal{L}(A_1(a,b,c,d,e,f)) & = \{a,b,c,\overline{d},\overline{e},\overline{f}\} \\ \mathcal{L}(A_2(b,d)) & = \{\overline{b},d\} \\ \mathcal{L}(A_3(c,e)) & = \{\overline{c},e\} \ . \end{array}$$

To build the system, we have to introduce a buffer before every input channel. Thus the initial configuration is:

$$\begin{array}{l} \nu a', b, b', c, c', d, d', e, e' \\ (a \mapsto a' \mid b \mapsto b' \mid c \mapsto c' \mid d \mapsto d' \mid e \mapsto e' \mid \\ A_1(a', b', c', d, e, f) \mid A_2(b, d') \mid A_3(c, e') \ ) \end{array}$$

It is easily checked that the resulting process belongs to the class KP.

To summarize, to build confluent processes we can use: (i) nil and input prefix, (ii) restricted composition, (iii) injective recursive calls, and (iv) recursive equations $A(a^*) = P$, where P is built according to the rules above. This class of processes is enough to represent Kahn networks. Notice that, via recursion, we can also represent Kahn networks with a dynamically changing number of nodes (see example 409).

27.4 Reactivity and local confluence in lts

We know that a terminating and locally confluent rewriting system is confluent (proposition 47). We present a suitable generalization of this result to confluent lts. First we need to generalize the notion of termination.

Definition 463 (reactivity) *Let P be a process. We say that it is* terminating *(or strongly normalizing) if there is no infinite sequence:*

$$P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \cdots$$

and that it is reactive *(or fully terminating) if all its derivatives are terminating.*

Definition 464 (local confluence) *Let P be a process. We say that it is* locally confluent *if for all its derivatives Q:*

$$\frac{Q \xrightarrow{\alpha} Q_1 \quad Q \xrightarrow{\beta} Q_2}{\exists\, Q_1', Q_2' \ (Q_1 \overset{\beta \backslash \alpha}{\Rightarrow} Q_1', \quad Q_2 \overset{\alpha \backslash \beta}{\Rightarrow} Q_2', \ \textit{and} \quad Q_1' \approx Q_2')} \ .$$

Exercise 465 *Consider again the process:*

$$A(a,b) = a.\nu c\ (A(a,c) \mid \overline{b}.A(c,b))\ .$$

Is the process $A(a,b)$ reactive? Consider the cases $a \neq b$ and $a = b$.

Exercise 466 *Consider the process: $A = a.b+\tau.(a.c+\tau.A)$. Check whether A is: (1) τ-inert, (2) locally confluent, (3) terminating, (4) reactive, (5) determinate, and (6) confluent.*

Suppose P is a *reactive* process and let W be the set of its derivatives. For $Q, Q' \in W$ write $Q > Q'$ if Q rewrites to Q' by a positive number of τ-actions. Then $(W, >)$ is a well founded set.

Proposition 467 *If a process is* reactive *and* locally confluent *then it is* confluent.

Proof. Let B be the relation $\stackrel{\tau}{\rightarrow} \cup (\stackrel{\tau}{\rightarrow})^{-1} \cup \approx$ (restricted to W) and B^* its reflexive and transitive closure. Note that B^* is symmetric too. We take the following steps.

1. For every derivative Q of P it holds:
$$\frac{Q \stackrel{\tau}{\Rightarrow} Q_1, \quad Q \stackrel{\alpha}{\Rightarrow} Q_2}{\exists Q_3\ (\ Q_1 \stackrel{\alpha}{\Rightarrow} Q_3 \text{ and } Q_2 B^* Q_3\ )}\ .$$
2. The relation B^* is a weak-bisimulation.
3. The process P is τ-inert.
4. The process P is confluent.

Note that B^* is a binary relation on W (the derivatives of P).

Step 1 The argument is *by induction* (cf. proposition 30) on the well founded order $(W, >)$.

- If $Q = Q_1$ then the statement holds trivially.
- So assume $Q \stackrel{\tau}{\rightarrow} Q_3 \stackrel{\tau}{\Rightarrow} Q_1$ and consider 2 cases.
  1. If $Q \stackrel{\tau}{\rightarrow} Q_4 \stackrel{\alpha}{\Rightarrow} Q_2$.
     - By local confluence, $Q_3 \stackrel{\tau}{\Rightarrow} Q_5$, $Q_4 \stackrel{\tau}{\Rightarrow} Q_6$, and $Q_5 \approx Q_6$.
     - By inductive hypothesis, $Q_6 \stackrel{\alpha}{\Rightarrow} Q_7$ and $Q_2 B^* Q_7$.
     - By definition of bisimulation, $Q_5 \stackrel{\alpha}{\Rightarrow} Q_8$ and $Q_7 \approx Q_8$.
     - By inductive hypothesis, $Q_1 \stackrel{\alpha}{\Rightarrow} Q_9$ and $Q_8 B^* Q_9$.

     So $Q_2 B^* Q_7 \approx Q_8 B^* Q_9$, and by definition of B, $Q_2 B^* Q_9$.
  2. If $Q \stackrel{\alpha}{\rightarrow} Q_4 \stackrel{\tau}{\Rightarrow} Q_2$ with $\alpha \neq \tau$.
     - By local confluence, $Q_3 \stackrel{\alpha}{\Rightarrow} Q_5$, $Q_4 \stackrel{\tau}{\Rightarrow} Q_6$, $Q_5 \approx Q_6$.
     - By inductive hypothesis, $Q_1 \stackrel{\alpha}{\Rightarrow} Q_7$ and $Q_5 B^* Q_7$.

     So $Q_2 \stackrel{\tau}{\Leftarrow} Q_4 \stackrel{\tau}{\Rightarrow} Q_6 \approx Q_5 B^* Q_7$. Hence $Q_2 B^* Q_7$.

Step 2 The relation B^* is a weak-bisimulation.

Suppose $Q_0 B Q_1 \cdots B Q_n B Q_{n+1}$ and $Q_0 \stackrel{\alpha}{\Rightarrow} Q'_0$. Proceed by induction on n and case analysis on $Q_n B Q_{n+1}$. By inductive hypothesis, we know that $Q_n \stackrel{\alpha}{\Rightarrow} Q'_n$ and $Q'_0 B^* Q'_n$.

1. If $Q_n \approx Q_{n+1}$ then $Q_{n+1} \stackrel{\alpha}{\Rightarrow} Q'_{n+1}$ and $Q'_n \approx Q'_{n+1}$. So $Q'_0 B^* Q'_n \approx Q'_{n+1}$ and we use $B^* \circ \approx \subseteq B^*$.
2. If $Q_n \stackrel{\tau}{\leftarrow} Q_{n+1}$ then $Q_{n+1} \stackrel{\alpha}{\Rightarrow} Q'_n$.
3. If $Q_n \stackrel{\tau}{\rightarrow} Q_{n+1}$ then by Step (1), $Q_{n+1} \stackrel{\alpha}{\Rightarrow} Q'_{n+1}$ and $Q'_n B^* Q'_{n+1}$.
   So $Q'_0 B^* Q'_n B^* Q'_{n+1}$ and we use $B^* \circ B^* \subseteq B^*$.

Step 3 The process P is τ-inert.

By definition, $\stackrel{\tau}{\rightarrow} \subseteq B^*$ and by Step (2), $B^* \subseteq \approx$.

Step 4 The process P is confluent.

By induction on the well-founded order W. We distinguish two cases.

1. Suppose $Q \stackrel{\alpha}{\rightarrow} Q_3 \stackrel{\tau}{\Rightarrow} Q_1$ and $Q \stackrel{\beta}{\rightarrow} Q_4 \stackrel{\tau}{\Rightarrow} Q_2$, with $\alpha, \beta \neq \tau$.
   - By local confluence, $Q_3 \stackrel{\beta \backslash \alpha}{\Rightarrow} Q_5$, $Q_4 \stackrel{\alpha \backslash \beta}{\Rightarrow} Q_6$, and $Q_5 \approx Q_6$.
   - By Step (3), $Q_4 \approx Q_2$, and by weak bisimulation, $Q_2 \stackrel{\alpha \backslash \beta}{\Rightarrow} Q_8$, $Q_6 \approx Q_8$.
   - By Step (3), $Q_3 \approx Q_1$, and by weak bisimulation, $Q_1 \stackrel{\beta \backslash \alpha}{\Rightarrow} Q_7$, $Q_5 \approx Q_7$.

   So we have $Q_8 \approx Q_6 \approx Q_5 \approx Q_7$ as required.
2. Suppose $Q \stackrel{\tau}{\rightarrow} Q_3 \stackrel{\alpha}{\Rightarrow} Q_1$ and $Q \stackrel{\beta}{\Rightarrow} Q_2$.
   - By Step (3), $Q \approx Q_3$, and by weak bisimulation, $Q_3 \stackrel{\beta}{\Rightarrow} Q_5$, $Q_2 \approx Q_5$.
   - By inductive hypothesis, $Q_1 \stackrel{\beta \backslash \alpha}{\Rightarrow} Q_6$, $Q_5 \stackrel{\alpha \backslash \beta}{\Rightarrow} Q_7$, and $Q_6 \approx Q_7$.
   - By weak bisimulation, $Q_2 \stackrel{\alpha \backslash \beta}{\Rightarrow} Q_4$ and $Q_4 \approx Q_7$.

   So $Q_4 \approx Q_7 \approx Q_6$ as required. □

Exercise 468 *Suppose P is a CCS process that is reactive and such that for every derivative Q of P we have:*

$$\frac{Q \stackrel{\tau}{\rightarrow} Q_1 \qquad Q \stackrel{\tau}{\rightarrow} Q_2}{Q_1 \approx Q_2} .$$

Show that this implies that for every derivative Q of P we have:

$$\frac{Q \stackrel{\tau}{\Rightarrow} Q_1 \quad Q \stackrel{\tau}{\Rightarrow} Q_2}{\exists Q'_1, Q'_2 \ (\ Q_1 \stackrel{\tau}{\Rightarrow} Q'_1, \quad Q_2 \stackrel{\tau}{\Rightarrow} Q'_2, \quad \textit{and} \quad Q'_1 \approx Q'_2\ )} .$$

27.5 Summary and references

A process is determinate if it always reacts in the same way to the stimuli coming from the environment. Confluence is a stronger property than determinacy that is preserved by a restricted form of parallel composition. Following [Mil95][chapter 11], we have presented 3 alternative characterizations of confluence. We have seen that a restricted form of parallel composition preserves confluence and as a case study we have shown that this fragment of *CCS* is enough to represent Kahn networks [Kah74]. Synchronous data flow languages such as Lustre [CPHP87] can be regarded as a refinement of this model where buffers have size 0. A rather complete study of the notion of confluence in the more general framework of the π-calculus is in [PW97], which builds on previous work on confluence for *CCS* with value passing. Reactivity is a form of hereditary termination. A process is reactive if it terminates after any sequence of interactions with the environment. The presented generalization of Newman's proposition 47 is described in [GS96].

Chapter 28

Synchronous/Timed models

As mentioned in chapter 19, one important classification criterion in concurrent systems is the *relative speed* of the processes. In particular, in chapter 19 we have contrasted asynchronous and synchronous systems. So far we have considered models ($\mathsf{Imp}_{\|}$, *CCS*) where processes are *asynchronous*, *i.e.*, proceed at independent speeds. In particular, processes can only *synchronize* through an *await* statement or an *input/output* communication. In the following we are going to discuss an enrichment of the *CCS* model where processes are *synchronous* (or *timed*). In first approximation, in a synchronous concurrent system all processes *proceed in lockstep* (at the same speed). In other words, the computation is regulated by a notion of *instant* (or *round*, or *phase*, or *pulse*,...).

Though synchronous circuits are typical examples of synchronous systems, one should *not* conclude that synchronous systems are *hardware*. Notions of synchrony are quite useful in the design of *software systems* too. The programming of many problems in a distributed setting can be 'simplified' or even 'made possible' by a synchronous assumption. Examples include: leader election, minimum spanning tree, and consensus in the presence of failures. In general, the notion of synchrony is a useful *logical concept* that can make *programming easier*.

The formalization of a synchronous model depends on the way the notion of instant is considered. One possibility is to assume that at each instant each (sequential) process performs a locally defined amount of work. For instance, a popular definition found in books on distributed algorithms requires that at each instant each process (1) writes in the output communication channels, (2) reads the contents of the input communication channels, and (3) computes its next state. However, a less constrained viewpoint is possible which consists in assuming that at each instant, each process performs an arbitrary (but hopefully finite) number of actions. The instant ends when each process has either terminated its task for the current instant or it is suspended waiting for events that cannot arise. This is the viewpoint taken by synchronous languages such as *Esterel* and we shall describe next its formalization in the framework of *CCS*. The reader should keep in mind that we select *CCS* because of its simplicity but that the approach can be easily ported to other models of concurrent systems.

28.1 Timed *CCS*

We discuss the definition of a synchronous/timed model on top of *CCS*. Following the terminology in the literature, we call this model *timed CCS* (*TCCS*). As usual, we write $\alpha, \alpha', \ldots$ for the *CCS* actions and we reserve $\ell, \ell', \ldots$ for the *CCS* actions but the τ action. We denote

$$\frac{P \overset{\tau}{\not\to} \cdot}{(P \rhd Q) \overset{\mathsf{tick}}{\to} Q} \qquad \frac{}{0 \overset{\mathsf{tick}}{\to} 0}$$

$$\frac{}{\ell.P \overset{\mathsf{tick}}{\to} \ell.P} \qquad \frac{P_i \overset{\mathsf{tick}}{\to} P_i' \quad i = 1,2 \quad (P_1 \mid P_2) \overset{\tau}{\not\to} \cdot}{(P_1 \mid P_2) \overset{\mathsf{tick}}{\to} (P_1' \mid P_2')}$$

$$\frac{P_i \overset{\mathsf{tick}}{\to} P_i' \quad i = 1,2}{(P_1 + P_2) \overset{\mathsf{tick}}{\to} (P_1' + P_2')} \qquad \frac{P \overset{\mathsf{tick}}{\to} P'}{\nu a\ P \overset{\mathsf{tick}}{\to} \nu a\ P'}$$

Table 28.1: Labelled transition system for the tick action

with $\mu, \mu', \ldots$ the *TCCS* actions. They are obtained by extending the *CCS* actions (chapter 26) with a new tick action which represents the move to the following instant:

$$\mu \quad ::= \alpha \mid \mathsf{tick} \quad (\textit{TCCS} \text{ actions}).$$

We also extend the syntax of *CCS* processes with a new operator 'else-next' which allows to program processes which are time dependent and are able to react to the absence of an event. Intuitively, the process $(P \rhd Q)$ tries to run P in the current instant and if it cannot it runs Q in the following.

$$P \quad ::= \cdots \mid (P \rhd P) \quad (\textit{TCCS} \text{ processes}).$$

The labelled transition system for *TCCS* includes the usual rules for the α actions (Table 26.1) plus:

$$\frac{P \overset{\alpha}{\to} P'}{(P \rhd Q) \overset{\alpha}{\to} P'} \quad \text{(a rule for else-next)}.$$

Moreover, we introduce in Table 28.1 special rules for the tick action describing the passage of time. The intuition is the following:

A process can tick if and only if it cannot perform τ actions.

Incidentally, this is in perfect agreement with the usual feeling that we do not see time passing when we have something to do!

Exercise 469 (on formalising tick actions) *Check that $P \overset{\mathsf{tick}}{\to} \cdot$ if and only if $P \overset{\tau}{\not\to} \cdot$. The lts in Table 28.1 uses the negative condition $P \overset{\tau}{\not\to} \cdot$. Show that this condition can be formalized in a positive way by defining a formal system to derive judgments of the shape $P \downarrow L$ where L is a set of observable actions and $P \downarrow L$ if and only if $P \overset{\tau}{\not\to} \cdot$ and $L = \{\ell \mid P \overset{\ell}{\to} \cdot\}$.*

The following exercise identifies two important choices in the design of *TCCS*.

Exercise 470 (continuations of tick action) *We say that P is a 'CCS process' if it does not contain the else_next operator. Show that:*

1. *If $P \overset{\mathsf{tick}}{\to} Q_1$ and $P \overset{\mathsf{tick}}{\to} Q_2$ then $Q_1 = Q_2$. So the passage of time is* deterministic.
2. *If P is a CCS process and $P \overset{\mathsf{tick}}{\to} Q$ then $P = Q$. So CCS processes are* insensitive to the passage of time.

Exercise 471 (programming a switch) *Let* $\mathsf{tick}.P = (0 \triangleright P)$ *and* $\mathsf{tick}^{\,n}.P = \mathsf{tick} \cdots \mathsf{tick}.P$, *n times.*

1. *Program a* light switch *Switch*$(press, off, on, brighter)$ *that behaves as follows:*
   - *Initially the switch is off.*
   - *If the switch is off and it is pressed then the light turns on.*
   - *If the switch is pressed again in the following 2 instants then the light becomes brighter while if it is pressed at a later instant it turns off again.*
   - *If the light is brighter and the switch is pressed then it becomes off.*
2. *Program a* fast user *Fast*$(press)$ *that presses the switch every 2 instants and a* slow user *Slow*$(press)$ *that presses the switch every 4 instants.*
3. *Consider the systems:*

$$\nu press\ (\ Switch(press, off, on, brighter) \mid Fast(press)\ )$$
$$\nu press\ (\ Switch(press, off, on, brighter) \mid Slow(press)\ )$$

and determine when the light is going to be off, on, and bright.

Definition 472 *The notion of* weak transition *is extended to the* tick *action by defining:*

$$\overset{\mathsf{tick}}{\Rightarrow} \;=\; \overset{\tau}{\Rightarrow} \circ \overset{\mathsf{tick}}{\rightarrow} \circ \overset{\tau}{\Rightarrow} \qquad \textit{(weak } \mathsf{tick} \textit{ action)}\ .$$

Then we denote with $\approx_{\mathsf{tick}}$ *the related largest weak bisimulation.*

Exercise 473 (bisimulation for *TCCS***)** *Show that* $\approx_{\mathsf{tick}}$ *is preserved by parallel composition. Also show that* $((P_1 \triangleright P_2) \triangleright P_3) \approx_{\mathsf{tick}} (P_1 \triangleright P_3)$. *Thus the nesting of else-next operators on the left is useless!*

Exercise 474 (more on congruence of $\approx_{\mathsf{tick}}$ **)** *Suppose* $P_1 \approx_{\mathsf{tick}} P_1$ *and* $Q_1 \approx_{\mathsf{tick}} Q_2$. *Prove or give a counterexample to the following equivalences.*

1. $P_1 + Q_1 \approx_{\mathsf{tick}} P_2 + Q_2$.
2. $((\ell.P_1) \triangleright Q_1) \approx_{\mathsf{tick}} ((\ell.P_2) \triangleright Q_2)$.
3. $(P_1 \triangleright Q_1) \approx_{\mathsf{tick}} (P_2 \triangleright Q_2)$.

We have identified the *CCS* processes with the *TCCS* processes that do not contain an else-next operator. A natural question is whether the equivalences we have on *CCS* are still valid when the *CCS* processes are placed in a timed environment. A basic observation is that a diverging computation does not allow time to pass. Thus if we denote with Ω the diverging process $\tau.\tau.\tau \cdots$ we have $0 \not\approx_{\mathsf{tick}} \Omega$ while in the ordinary (termination insensitive) bisimulation for *CCS* we have $0 \approx \Omega$. The situation is more pleasant for *reactive* processes cf. chapter 27).

Proposition 475 (*CCS* vs. *TCCS*) *Suppose* P, Q *are CCS processes.*

1. *$P \approx_{\mathsf{tick}} Q$ implies $P \approx Q$.*
2. *If moreover, P, Q are reactive then $P \approx Q$ implies $P \approx_{\mathsf{tick}} Q$.*

PROOF. (1) *TCCS* bisimulation is stronger than *CCS* bisimulation and α-derivatives of *CCS* are again *CCS* processes.

(2) First notice that for a *CCS* process being reactive w.r.t. *CCS* actions is the same as being reactive w.r.t. *TCCS* actions. For α actions, the condition $P \approx Q$ suffices. Otherwise, suppose $P \stackrel{\mathsf{tick}}{\Rightarrow} P'$. By exercises 469 and 470(2), this means $P \stackrel{\tau}{\Rightarrow} P' \stackrel{\mathsf{tick}}{\rightarrow} P'$. By definition of *CCS* bisimulation, $Q \stackrel{\tau}{\Rightarrow} Q_1$, $P' \approx Q_1$. By reactivity, $Q_1 \stackrel{\tau}{\Rightarrow} Q' \stackrel{\mathsf{tick}}{\rightarrow}$. Again by definition of *CCS* bisimulation, $P' \approx Q'$, Hence $Q \stackrel{\mathsf{tick}}{\Rightarrow} Q'$ and $P' \approx Q'$. □

Exercise 476 (termination sensitive bisimulation) *Rather than restricting the attention to reactive processes, another possibility is to consider a bisimulation for CCS which is sensitive to termination. We write $P \downarrow$ if $P \stackrel{\tau}{\not\rightarrow} \cdot$ and $P \Downarrow$ if $P \stackrel{\tau}{\Rightarrow} Q$ and $Q \downarrow$. Show that on CCS processes the bisimulation $\approx_{\mathsf{tick}}$ can be characterized as the largest relation $\mathcal{R}$ which is a weak labelled bisimulation (in the usual CCS sense) and such that if $P \mathcal{R} Q$ and $P \downarrow$ then $Q \Downarrow$.*

28.2 A deterministic calculus based on signals

As a case study, we consider a variant of the *TCCS* model where processes interact through *signals* (rather than channels). A signal is either emitted or not. Once it is emitted it *persists* during the instant and it is *reset* at the end of it. Thus the collection of emitted signals grows *monotonically* during each instant.

The presented calculus is named *SL* (*synchronous language*). We describe it as a fragment of timed *CCS* where we write $s, s', \ldots$ for *signal* names. The syntax of *SL* processes is as follows:

$$P ::= 0 \mid s.P, P \mid (\mathsf{emit}\ s) \mid (P \mid P) \mid \nu s\ P \mid A(s^*) \quad (SL \text{ processes}).$$

The newly introduced operators can be understood in terms of those of *TCCS* as follows:

$$\begin{array}{ll} s.P, Q & = (s.P \triangleright Q) \\ (\mathsf{emit}\ s) & = (\overline{s}.Emit(s) \triangleright 0) \\ & \text{where: } Emit(s) = (\overline{s}.Emit(s) \triangleright 0)\ . \end{array}$$

Notice that in *SL* there is no sum and no prefix for emission (cf. asynchronous π-calculus, chapter 30). The input is a specialized form of the input prefix and the else-next operator. The derived synchronization rule is:

$$(\mathsf{emit}\ s) \mid s.P, Q \stackrel{\tau}{\rightarrow}\stackrel{\tau}{\rightarrow} (\mathsf{emit}\ s) \mid P\ .$$

The second τ transition is just recursion unfolding and we will ignore it in the following. Notice that:

$$(\mathsf{emit}\ s) \mid s.P_1, Q_1 \mid s.P_2, Q_2 \stackrel{\tau}{\Rightarrow} (\mathsf{emit}\ s) \mid P_1 \mid P_2\ .$$

The tick action can be expressed as:

$$\mathsf{tick}\ .P = \nu s\ s.0, P \qquad s \notin \mathsf{fv}(P)\ .$$

A *persistent input* (as in *TCCS*) is expressed as:

$$\mathsf{await}\ s.P = A(s^*), \quad \text{where: } A(s^*) = s.P, A(s^*),\ \mathsf{fv}(P) \cup \{s\} = \{s^*\}\ .$$

Exercise 477 *Re-program in SL the light switch seen in exercise 471. Compare the solution with the one based on TCCS.*

The *SL* calculus enjoys a *strong* form of confluence where one can close the diagram in *at most one step* and *up to* α*-renaming.*

Proposition 478 (strong confluence) *For all SL programs* P *the following holds:*

$$\frac{P \xrightarrow{\tau} P_1 \qquad P \xrightarrow{\tau} P_2}{P_1 \equiv P_2 \ \textit{or}\ \exists Q\ \ (P_1 \xrightarrow{\tau} Q, P_2 \xrightarrow{\tau} Q)}$$

Proof. Internal reductions are due either to *unfolding* or to *synchronization.* The only possibility for a *superposition* of the redexes is:

$$(\mathsf{emit}\ s) \mid s.P_1, Q_1 \mid s.P_2, Q_2\ .$$

And we exploit the fact that emission is *persistent.* □

The bisimulation $\approx_{\mathsf{tick}}$ developed for *TCCS* can be applied to *SL* too. However, because of the restricted form of *SL* processes, one can expect additional equations to hold. For instance:

$$s.(\mathsf{emit}\ s), 0 \text{ should be 'equivalent' to } 0\ . \tag{28.1}$$

A similar phenomenon arises with asynchronous communication in the π-calculus (cf. chapter 30). More generally, because *SL* is deterministic (cf. proposition 478) one can expect a collapse of the *bisimulation* and *trace* semantics (cf. proposition 439).

Exercise 479 (on *SL* equivalence) *Check that the equation (28.1) does not hold in the TCCS embedding. Also, prove or disprove the following equivalences:*

1. $s.(s.P, Q), Q \approx_{\mathsf{tick}} s.P, Q.$
2. $(\mathsf{emit}\ s) \mid s.P, Q \approx_{\mathsf{tick}} (\mathsf{emit}\ s) \mid P.$

28.3 Summary and references

Time, in the sense we have described it here, is derived from the notion of computation and as such it is a *logical* notion rather than a concept we attach on top of the computational model. Time passes when no computation is possible. Moving from an asynchronous to a synchronous model means enriching the language with the possibility to react to the absence of computation, *i.e.*, to the passage of time. The distinction between synchronous and asynchronous models is standard in the analysis of distributed algorithms (see, *e.g.*, [Lyn96]). In the framework of process calculi, a notion of 'timed' *CCS* is introduced in [Yi91]. This calculus has a $\mathsf{tick}(x)$ operator that describes the passage of x time units where x is a non-negative real. A kind of *else_next* operator is proposed in [NS94]. A so called *testing semantics* of a

process calculus very close to the one presented here is given in [HR95]. However, it seems fair to say that all these works generalize to *CCS* ideas that were presented for the *Esterel* programming language [BC84, BG92]. Two basic differences in the *Esterel* approach are that processes interact through *signals* and that the resulting calculus is *deterministic*.

Another important difference is that in the *Esterel* model it is actually possible to *react immediately* (rather than at the end of the instant) to the absence of a signal. This requires some semantic care, to avoid writing *paradoxical programs* such as $s.0, (\mathsf{emit}\ s)$ which are supposed to emit s when s is not there (cf. stabilization problems in the design of synchronous circuits). It also requires some clever *compilation techniques* to determine whether a signal is not emitted. In fact these techniques (so far!) are specific to *finite state models*.

The *SL* model [BdS96] we have described is a *relaxation* of the *Esterel* model where the *absence of a signal* can only be detected at the end of the instant. If we forget about *name generation*, then the *SL* model essentially defines a kind of *monotonic Mealy machine*. Monotonic in the sense that output signals can only depend *positively* on input signals (within the same instant). The monotonicity restriction allows to *avoid the paradoxical programs* as monotonic boolean equations do have a least fixed point! The *SL* model has a natural and *efficient implementation model* that works well for *general programs* (not just finite state machines). The model has been adapted to several *programming environments* (*C* [Bou91], *Scheme* [SBS04], *ML* [MP05]) and it has been used to program significant applications.

The *Esterel*/*TCCS*/*SL* models described here actually follow an earlier attempt at describing synchronous/timed systems in the framework of *CCS* known as *SCCS*/*Meije* model [Mil83, AB84]. The basic idea of these models is that the actions of the system live in an abelian (commutative) group freely generated from a collection of basic actions. At each instant, each (sequential) process must perform exactly one action and the observable result of the computation is the group composition of the actions performed by each process. This gives rise to a model with pleasant algebraic properties but whose implementation and generalization to a full scale programming languages appear to be problematic.

Finally, let us mention *timed automata* as another popular formalism for describing 'timed' systems [AD94]. This is an enrichment of finite state automata with *timing constraints* which still enjoys decidable model-checking properties. This is more a *specification language* for finite control systems than a *programming language*.

Chapter 29

Probability and non-determinism

We discuss the modelling of systems which exhibit both non-deterministic and probabilistic behaviors. We focus on three basic ideas. First, we should not confuse non-deterministic and probabilistic choice. Second, because of probabilistic choice, we need to lift relations on states (such as bisimulations) to relations on distributions. Third, again because of probabilistic choice, we need to revisit the notion of weak transition so as to formalize the notion that the system can evolve from one state to another with probability 1. In the following, these ideas are formalized in the framework of *CCS*, but the reader should keep in mind that the notion of probabilistic computation is an 'orthogonal' feature that can be added to a variety of models of concurrent systems (a similar consideration holds for the notion of synchronous/timed computation).

29.1 Preliminaries

Probabilities arise in several areas of system design and analysis. For instance, one may want to *analyze* programs or protocols that toss coins at some point in the computation, *e.g.*, the probability that a test for number primality returns the correct answer. In another direction, one may want to evaluate the reliability of a system given some probability of failure of its components. And yet in another direction, one may be interested in evaluating the performance of a system in terms of, say, the average waiting time of its users.

As already mentioned, in concurrent systems, non-determinism arises to account for *race conditions* and also as a *specification device*. It is then natural to *lift* methods for (deterministic) probabilistic systems to non-deterministic ones.

To fix the ideas, we define some standard notions. Let S be a *countable* set of *states*. A *(discrete) distribution* on the states S is a function $\Delta : S \to [0,1]$ such that $\Sigma_{s \in S} \Delta(s) = 1$. We denote with $Dist(S)$ the collection of distributions on S. If $S' \subseteq S$ we denote with $\Delta[S']$ the sum $\Sigma_{s' \in S'} \Delta(s')$. It is a well known fact that the value of the sum *does not* depend on the enumeration of the states. We may represent a distribution as a *formal sum* $\Sigma_{i \in I}[p_i]s_i$ where $\Sigma_{i \in I} p_i = 1$. The *binary* version is also written as $s_1 +_p s_2$ which stands for $[p]s_1 + [1-p]s_2$. The *unary* version is shortened to s which stands for $[1]s$.

The notion of *discrete time Markov chain* is standard in probability theory. It is based on: (i) a discrete notion of time $t = 0, 1, 2, \ldots$ and (ii) a transition 'matrix':

$$P : S \to Dist(S) \qquad \text{(Markov chain)}, \tag{29.1}$$

where $p(s)(s')$ is the probability that the system being at state s at time t moves into state s' at time $t+1$ (for any t). This is a beautiful theory connecting linear algebra to probability theory.

The notion of *Markov decision process* is an elaboration on Markov chains popularized in the 50's. We add to a Markov chain a countable set of *actions Act* and redefine the transition matrix as a partial function:

$$P : (S \times Act) \rightharpoonup Dist(S) \qquad \text{(Markov decision process)}, \tag{29.2}$$

where $P(s,a)(s')$ is the *probability* of moving from s to s' given that the decision to perform action a has been taken. Possibly, we may consider a *reward function* $R : (S \times Act) \to (S \to \mathbf{R})$ too, where $R(s,a)(s')$ is the *reward* the decision maker gets if being in s and taking the decision a the system moves into s'. A *typical problem* in this area is to determine a *policy* for the decision maker that maximizes some cumulative function of the rewards.

Following the introduction of labelled transition systems, it is natural to consider a related notion of *probabilistic* lts. The *key point* is that being in state s and taking the action a the system may end up in *different* distributions of states. We now have a transition *relation*:

$$\to \subseteq \; S \times Act \times Dist(S) \qquad \text{(Probabilistic lts)}. \tag{29.3}$$

If all the distributions are trivial (Dirac) then we are back to *labelled transition systems* (cf. definition 356). If for a given state and action there is at most one distribution then we are back to *Markov Decision Processes*.

29.2 Probabilistic *CCS*

We discuss how the process calculus framework *CCS* adapts to the move from *lts* to *probabilistic lts*. In particular we look at:

1. The definition of *strong bisimulation* on probabilistic lts.
2. How to associate a probabilistic lts with a *Probabilistic CCS*, namely a *CCS* enriched with a probabilistic choice operator.

Our first problem is to find a way of lifting a relation on states to a relation on distributions. Let $\mathcal{R} \subseteq S \times S$ be an equivalence relation on states and let $[s]_R$ denote the equivalence class of s and S/R the set of equivalence classes.

Definition 480 (lumping equivalence) *We lift $\mathcal{R}$ to an equivalence relation $\mathcal{D}(R)$ on $Dist(S)$ as follows:*

$$\Delta \; \mathcal{D}(\mathcal{R}) \; \Delta' \quad \textit{if} \quad \forall\, s \in S \;\; \Delta[\,[s]_{\mathcal{R}}\,] = \Delta'[\,[s]_{\mathcal{R}}\,] \; .$$

Thus two distributions are *equivalent* with respect to $\mathcal{R}$ if they are the same modulo the equivalence induced by $\mathcal{R}$. This is also called *lumping equivalence* in Markov processes literature (lumping=aggregate). We can then introduce a notion of bisimulation for probabilistic lts.

Definition 481 (probabilistic bisimulation) *Let* $(S, Act, \to)$ *be a* probabilistic *lts. An* equivalence relation $\mathcal{R}$ *over* S *is a* bisimulation *if* $s\ \mathcal{R}\ s'$ *and* $s \xrightarrow{\alpha} \Delta$ *implies:*

$$\exists\, \Delta'\ \ s' \xrightarrow{\alpha} \Delta' \quad \text{and} \quad \Delta\ \mathcal{D}(R)\ \Delta'\ .$$

We denote with $\sim_P$ *the largest bisimulation.*[1]

Example 482 (bisimilar states) *The following states* A *and* A' *are bisimilar:*

$$\begin{array}{ll} A = in.([0.8]B + [0.2]C) & A' = in.([0.8]B + [0.1]D + [0.1]E) \\ B = out.[1]A & D = err.[1]A \\ C = err.[1]A & E = err.[1]A\ . \end{array}$$

We may represent this system as a bipartite graph *where Nodes = States ∪ Distributions. The edges from States to Distributions are labelled with (CCS)* actions *and the labels in the other direction with* probabilities.

Example 483 (non-bisimilar states) *The following states* A *and* A' *are* not *bisimilar.*

$$\begin{array}{l|l} A = in.\Delta_1 + in.\Delta_2 & \Delta_1 = (B +_{0.9} C) \\ & \Delta_2 = (B +_{0.5} C) \\ B = out.[1]A & \\ C = err.[1]A & \\ A' = in.\Delta_1 + in.\Delta_2 + in.\Delta_3 & \Delta_3 = (B +_{0.7} C)\ . \end{array}$$

Moreover, one may argue that Δ_3 *is a* convex combination *of* Δ_1 *and* Δ_2*. Indeed taking* $\lambda = 0.5$*:*

$$\Delta_3 = \lambda \cdot \Delta_1 + (1 - \lambda) \cdot \Delta_2\ .$$

There exists a more relaxed *definition of bisimulation that takes this into account.*

Example 484 (communication protocol) *We introduce some TCCS notation (cf. chapter 28):*

$$a.P \rhd_n Q = a.P \rhd (a.P \rhd \cdots (a.P \rhd Q) \cdots)\ .$$

We model a communication medium that may lose messages (but acknowledgments are never lost):

$$\begin{array}{lll} S & = send.S' & \textit{(sender)} \\ S' & = \overline{in}.(ack.S \rhd_2 S') & \\ M & = in.(M +_{0.1} \mathsf{tick}\ .(\overline{out}.M) & \textit{(medium)} \\ R & = out.\overline{rec}.\overline{ack}.R & \textit{(receiver)}\ . \end{array}$$

The medium transmits at most 1 message/instant and whenever the message is lost 2 instants pass without any message being transmitted.

Exercise 485 (vending machine) *Here is an* unreliable vending machine *with* slow *and* fast *users where we write* P *for* $[1]P$*.*

$$\begin{array}{ll} VM & = coin.(\,VM +_{0.1} VM') \\ VM' & = ((\overline{tea}.VM + \overline{coffee}.VM) \rhd VM) \\ SlowU & = \overline{coin}.\mathsf{tick}\ .tea.0 \\ FastU & = \overline{coin}.tea.0\ . \end{array}$$

Consider $(VM \mid SlowU)$ *and* $(VM \mid FastU)$*. Who may get the tea?*

[1] For the sake of simplicity, we assume the relations under consideration are *equivalence* relations. However, it is possible to develop a notion of probabilistic bisimulation without this assumption.

RULES

$$\frac{}{\Sigma_{i\in I}\alpha_i.\Delta_i \stackrel{\alpha_j}{\to} \Delta_j} \qquad \frac{P \stackrel{a}{\to} \Delta \quad P' \stackrel{\overline{a}}{\to} \Delta'}{(P \mid P') \stackrel{\tau}{\to} (\Delta \mid \Delta')}$$

$$\frac{P \stackrel{\alpha}{\to} \Delta}{(P \mid P') \stackrel{\alpha}{\to} \Delta \mid [1]P'} \qquad \frac{P \stackrel{\alpha}{\to} \Delta \quad a, \overline{a} \neq \alpha}{\nu a\ P \stackrel{\alpha}{\to} \nu a\ \Delta}$$

$$\frac{[b^*/a^*]P \stackrel{\alpha}{\to} \Delta}{A(b^*) \stackrel{\alpha}{\to} \Delta} \quad \text{if } A(a^*) = P\ .$$

NOTATION FOR DISTRIBUTIONS

$$\begin{array}{rcl} \nu a\ \Sigma_{i\in I}[p_i]P_i & \equiv & \Sigma_{i\in I}[p_i]\nu a\ P_i \\ \Sigma_{i\in I}[p_i]P_i \mid \Sigma_{j\in J}[q_j]Q_j & \equiv & \Sigma_{(i,j)\in I\times J}[p_i \cdot q_j](P_i \mid Q_j)\ . \end{array}$$

Table 29.1: Probabilistic lts for *PCCS*

We refine the definition of *CCS* to account for probabilistic computation. To this end, we distinguish *processes* (states) and (formal) *distributions*:

$$\begin{array}{lll} P ::= & 0 \mid \Sigma_{i\in I}\alpha_i.\Delta_i \mid (P \mid P) \mid \nu a\ P \mid A(a^*) & \text{(processes)} \\ \Delta ::= & \Sigma_{i\in I}[p_i]P_i & \text{(distributions)} \end{array}$$

Table 29.1 presents the *rules for the probabilistic lts*. They are quite similar to those for *CCS* (cf. Table 26.1) modulo the introduction of a suitable notation for distributions.

Exercise 486 *Apply the rules to derive:* $a.(P_1 +_{0.5} P_1) \mid P_3 \stackrel{a}{\to} (P_1 \mid P_3) +_{0.5} (P_2 \mid P_3)$.

29.3 Measuring transitions

We discuss the problem of defining a notion of *weak* transition in a probabilistic setting. Consider the following probabilistic lts:

$$P_0 \ = \tau.(P_0 +_{0.5} P_1)\ , \qquad P_1 \ = a.[1]0\ . \tag{29.4}$$

It seems reasonable to regard P_0 as equivalent to P_1 in that we expect P_0 to end up in P_1, but the notion of *weak* transition as transitive closure is not quite adequate. Indeed, by a finite iteration of τ moves we can only reach a distribution of the shape $P_0 +_p P_1$, where $p = 1/2^n$. It is only to the 'limit' that the process P_0 reaches the process P_1. Notice that a similar issue arises when observing the termination of a (deterministic) probabilistic program. In this case, we do not want to distinguish between a program that terminates and a program that terminates with probability 1. For the sake of simplicity, we shall work with Markov decision processes, *i.e.*, we assume:

$$P : (S \times Act) \rightharpoonup Dist(S)\ .$$

We shall comment at the end of the section on the generalization to probabilistic lts.

We assume the reader has been exposed to a course in discrete probability and we just recall a few elementary definitions and facts. A σ-algebra is a triple $(\Omega, \mathcal{A}, P)$ where Ω is a set, $\mathcal{A} \subseteq 2^\Omega$ is a non-empty set of events (subsets of Ω) which is closed under countable unions and complement, and $P : \mathcal{A} \to [0,1]$ is a function that assigns a probability to each event so that $P(\Omega) = 1$ and $P(\bigcup_{i \in \mathbf{N}} A_i) = \Sigma_{i \in \mathbf{N}} P(A_i)$ if the events A_i are pairwise disjoint.

If Ω is finite or countable, it is often possible to define a function $P : \Omega \to [0,1]$ such that $\Sigma_{\omega \in \Omega} P(\omega) = 1$. Then we can take the set of events as the parts of Ω, *i.e.*, $\mathcal{A} = 2^\Omega$, and extend P to $\mathcal{A}$ by defining for $A \in \mathcal{A}$:

$$P(A) = \Sigma_{\omega \in A} P(\omega) \ .$$

However, this approach is not always viable. One important and famous example concerns the interval $[0,1]$ of real numbers. In this case, it is possible to show that there is no reasonable probability measure on $2^{[0,1]}$ (the subsets of the real interval $[0,1]$). On the other hand, assuming $0 \leq a \leq b \leq 1$, we expect the probability of the interval $[a,b]$ to be $(b-a)$. We can then build the *least* σ-algebra which includes the intervals (this is also called the Borel algebra). It turns out that we can rely on this construction to build a σ-algebra over the possible executions of a Markov decision process.

Definition 487 *An execution of a Markov decision process $P : S \times Act \rightharpoonup Dist(S)$ is a sequence:*

$$\sigma = s_0 \stackrel{a_1}{\to} s_1 \stackrel{a_2}{\to} \cdots \stackrel{a_n}{\to} s_n \tag{29.5}$$

such that $P(s_i, a_{i+1})$ is defined and $P(s_i, a_{i+1})(s_{i+1}) > 0$ for $i = 0, \ldots, n-1$.

We define $trace(\sigma) = a_1 \cdots a_n$, $first(\sigma) = s_0$, and $last(\sigma) = s_n$. Let $Ex(s)$ be the collection of executions σ such that $first(\sigma) = s$. We write $\sigma \leq \sigma'$ if σ is a prefix of σ' and denote with $\uparrow \sigma$ the *cone* generated by σ, *i.e.*, the collection of executions with prefix σ:

$$\uparrow \sigma = \{\sigma' \mid \sigma \leq \sigma'\} \ . \tag{29.6}$$

Then we build the least σ-algebra on $Ex(s)$ which includes the cones $\uparrow \sigma$ for $\sigma \in Ex(s)$. If σ is an execution with the shape (29.5) above, we let the probability of the generated cone to be:

$$P(\uparrow \sigma) = \Pi_{i=0,\ldots,n-1} P(s_i, a_{i+1})(s_{i+1}) \ .$$

Let Act^* be the set of finite words over Act. If $A \subseteq Act^*$ and $C \subseteq S$ then we define the set of executions, starting in s, going through C, while producing a trace in A:

$$Ex(s, A, C) = \bigcup \{\uparrow \sigma \mid \sigma \in Ex(s), last(\sigma) \in C, trace(\sigma) \in A\} \ .$$

Since $Ex(s, A, C)$ is a countable union of cones this set is an event in the least (Borel) σ-algebra defined above and its probability is defined.

Exercise 488 *Let $\mathbf{N}^*$ be the set of finite words over the natural numbers with generic elements $w, w', \ldots$. Denote with $\uparrow w = \{w' \mid w \leq w'\}$ the* cone *generated by w. Let $p_i = 1/2^{i+1}$ and notice that $\Sigma_{i=0,\ldots,\infty} p_i = 1$. For $w = i_1 \cdots i_n \in \mathbf{N}^*$, define the probability of the generated cone as:*

$$P(\uparrow w) = p_{i_1} \cdots p_{i_n} \ .$$

Also define a segment *$[w, w']$ as the (finite) set $\{w'' \mid w \leq w'' \leq w'\}$. Show that the events of the least σ-algebra generated by the the cones consist of countable disjoint unions of cones and segments and that the probability of a segment is 0.*

Before defining a notion of *weak* bisimulation, it is convenient to characterize the notion of (strong) bisimulation introduced in definition 481.

Proposition 489 *An equivalence relation R on the states S of a Markov decision process $P : S \times Act \rightharpoonup Dist(S)$ is a bisimulation if and only if the following holds: for all $s, s' \in S$, $\alpha \in Act$, and $C \in S/R$ if $(s, s') \in R$ then*

$$P(Ex(s, \{\alpha\}, C)) = P(Ex(s', \{\alpha\}, C)) \ .$$

Proof. If $P(s, \alpha)$ is undefined then the set of executions is empty and the probability is 0. Otherwise, notice that $P(Ex(s, \{\alpha\}, [s'']_R)) = \Delta([s'']_R)$ where $\Delta = P(s, \alpha)$. □

The notion of weak bisimulation then amounts to replace the singleton set $\{\alpha\}$ with the sets corresponding to the regular expressions τ^* and $\tau^*\alpha\tau^*$, for $\alpha \neq \tau$; in the definition below, we shall actually abuse notation by denoting these sets by the corresponding regular expressions.

Definition 490 *An equivalence relation R on the states S of a Markov decision process $P : S \times Act \rightharpoonup Dist(S)$ is a bisimulation if for all states $s, s' \in S$, word $\alpha \in Act \cup \{\epsilon\}$, and equivalence class $C \in S/R$ if $(s, s') \in R$ then*

$$P(Ex(s, \tau^*\alpha\tau^*, C)) = P(Ex(s', \tau^*\alpha\tau^*, C)) \ .$$

Example 491 *The probabilistic lts (29.4) can be regarded as a Markov decision process with transitions $s_0 \xrightarrow{\tau} s_0 +_{1/2} s_1$ and $s_1 \xrightarrow{a} [1]s_2$. Let us check that the system goes from state s_0 to state s_2 with probability 1 while producing a trace whose only observable action is a:*

$$P(Ex(s_0, \tau^*a\tau^*, \{s_2\})) = 1 \ .$$

To this end, consider the executions: $\sigma_n = s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_0 \xrightarrow{\tau} s_1 \xrightarrow{a} s_2$, where the state s_0 is repeated $n \geq 1$ times. Notice that $\uparrow \sigma_n \cap \uparrow \sigma_m = \emptyset$ if $n \neq m$. Thus:

$$P(\bigcup_{n \geq 1} \uparrow \sigma_n) = \Sigma_{n \geq 1} P(\uparrow \sigma_n) = \Sigma_{n \geq 1} \frac{1}{2^n} = 1 \ .$$

The definition of a σ-algebra on executions can be extended to probabilistic lts by introducing a notion of *scheduler* (or adversary). A scheduler is a function that associates with every execution the next non-deterministic move. The idea is then to measure executions relatively to a given scheduler.

29.4 Summary and references

Probabilistic lts are labelled relations from processes to distributions on processes. Taking this approach one distinguishes between non-deterministic and probabilistic choice. A first key construction consists in *lifting* a relation on processes to a relation on distributions (the notion of strong bisimulation is derived). A second key construction concerns the introduction of a σ-algebra on the collection of possible executions. This leads to a framework where we can measure the probability of a weak transition and thus define a suitable notion of weak bisimulation.

Vitali [Vit05] shows that certain subsets of the real line are not measurable. Rabin [Rab63] introduces a notion of *probabilistic automata* as a generalization of finite *deterministic* automata. The notion of Markov decision problem has been developed around 1950 (the book [Put94] is a standard reference). The notion of *probabilistic lts* has been put forward in [Var85] under the name of *concurrent Markov chains* in the framework of model-checking and it has been later revisited by [HJ90] in the framework of *TCCS*. The notion of *bisimulation* is introduced for Markov Decision Processes in [LS89] while the weaker notion of 'convex' bisimulation for probabilistic lts is introduced in [SL95]. The book [Pan09] covers the situation where distributions are on *continuous state spaces* and requires a non-trivial amount of measure theory.

Chapter 30

π-calculus

CCS provides a basic model of *communication* and *concurrency* while ignoring the mechanisms of *procedural* and *data abstraction* which are at the heart of sequential programming (in this respect, *CCS* is close to Turing machines). We can contrast *CCS* with a basic model of sequential programming such as the (typed) λ-calculus. A basic question is: how can we *integrate* the λ-calculus and *CCS*?

One standard approach, supported both by theory and by practice, is to take the λ-calculus as the *backbone* of the programming language and to add on top a few features for communication and concurrency. The resulting language provides a comfortable programming environment but one may question whether this is the *simplest model* one can hope for (Ockham's razor). It turns out that the superposition of the concepts of function and process leads to some *redundancy* and that it is possible to reduce to simpler languages such as the π-calculus.

There are *two main ways* to look at the π-calculus. On one hand, it can be regarded as an extension of *CCS* where channels exchange values that are themselves channel names. As such it inherits from *CCS* a relatively simple and tractable theory including labelled transition systems and bisimulation proof methods. This viewpoint is developed below. On the other hand, it can be regarded as a concurrent extension of one of the *intermediate functional languages* studied in the chapter 14 on the compilation of functional languages. As such it has an expressive power comparable (up to some encoding!) to the one of modern programming languages. This viewpoint is elaborated in chapter 31.

30.1 A π-calculus and its reduction semantics

The basic idea is that the π-calculus is an extension of value passing *CCS* where processes exchange *channel names* as in:

$$(\; x(y).P \mid \nu z \; \overline{x}z.Q \;) \xrightarrow{\tau} \nu z \; ([z/y]P \mid Q) \; .$$

Quoting the authors who introduced the π-calculus:

> It will appear as though we reduce all concurrent computation to something like a cocktail party, in which the only purpose of communication is to transmit (or to receive) a name which will admit further communications.

The abstract syntax of *a possible* π-calculus is defined as follows.

$$\begin{array}{lll} id & ::= x \mid y \mid \ldots & \text{(names)} \\ P & ::= 0 \mid id(id).P \mid \overline{id}id.P \mid (P \mid P) \mid \nu id\ P \mid [id = id]P \mid !(id(id).P) & \text{(processes).} \end{array}$$

The informal semantics is as follows: 0 does nothing, $x(y).P$ waits for a name z on the channel x and then becomes $[z/y]P$, $\overline{x}y.P$ sends y on the channel x and becomes P, $(P \mid Q)$ runs P and Q in parallel, $\nu x\ P$ creates the new name x and runs P, $[x = y]P$ compares x and y and becomes P if they are equal (otherwise it is stuck), $!(x(y).P)$ waits for a name z on the channel x and then becomes $[z/y]P \mid !(x(y).P)$ (thus the operator '$!$' replicates an input and allows to generate infinite recursive behaviors). In processes, the formal parameter of an input and the ν bind names. We define $\mathsf{fv}(P)$ as the set of names occurring free in a process P. As usual, bound names can be renamed according to the rules of α-conversion.

Remark 492 (definable operators) *In the presented version of the π-calculus, we have dropped two operators which are present in CCS: non-deterministic choice and recursive definitions. The reason is that, up to some restrictions, both can be encoded in the presented calculus. The encoding of non-deterministic choice is related to the one we have already considered in example 307. The encoding of recursive definitions amounts to replace, say,* $\mathsf{letrec}\ A(x) = P\ \mathsf{in}\ Q$ *by* $\nu A\ (!(A(x).P') \mid Q')$ *where P' and Q' are obtained from P and Q, respectively, by replacing each (tail) recursive call, say $A(y)$, with a message $\overline{A}y$.*

Remark 493 (name renaming and substitution) *The names of the π-calculus can be split in two categories: those on which we can just perform α-renaming and those on which we can perform both α-renaming and general (non-injective) substitutions. In particular, names bound by the ν operator fall in the first category while names bound by the (replicated) input operator fall in the second one. It makes sense to regard the first category of names as* constants *and the second one as* variables*, and indeed some authors distinguish two syntactic categories and add a third one which is the union of the first two.*

If we start with a process whose free names are constants then this property is preserved by reduction and all substitutions replace a variable by a constant. This is true of the labelled transitions described in the following section 30.2 too, assuming that all the actions are built out of constant names. A consequence of this remark is that it is possible to suppose that the reduction rules and the labelled transitions are given on processes where all free names are constants. This is in line with the usual practice in operational semantics where the reduction rules are defined on 'closed' programs (as, e.g.*, in chapter 8).*

In order to define a compositional semantics for the π-calculus we follow the approach presented for *CCS* in chapter 26 which amounts to define a contextual bisimulation and a labelled bisimulation and show that they coincide. However, because the notion of label for the π-calculus is not obvious, this time we shall start with contextual bisimulation. Table 30.1 defines the static contexts, a structural equivalence, and the reduction rules for the π-calculus. The related notions of commitment and contextual (weak) bisimulation are inherited directly from *CCS* (section 26.3).

Exercise 494 *Reduce the processes $\nu y\ \overline{x}y.P \mid x(z).Q$ and $\nu y\ \overline{x}y.P \mid x(z).(y(w).Q)$.*

SYNTAX

$$C \quad ::= [\,] \mid C \mid P \mid P \mid \nu x\ C \quad \text{(static contexts)}$$

STRUCTURAL EQUIVALENCE

$$\begin{array}{ll} P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 & \text{(associativity)} \\ P_1 \mid P_2 \equiv P_2 \mid P_1 & \text{(commutativity)} \\ \nu x\ P_1 \mid P_2 \equiv \nu x\ (P_1 \mid P_2) \quad x \notin \mathsf{fv}(P_2) & \text{(extrusion)} \end{array}$$

REDUCTION

$$\frac{}{\overline{x}y.P_1 \mid x(z).P_2 \to P_1 \mid [y/z]P_2} \qquad \frac{}{\overline{x}y.P_1 \mid !(x(z).P_2) \to (P_1 \mid [y/z]P_2) \mid !(x(z).P_2)}$$

$$\frac{}{[x = x]P \to P} \qquad \frac{P \equiv C[P'] \quad P' \to Q' \quad C[Q'] \equiv Q}{P \to Q}$$

Table 30.1: Reductions for the π-calculus

30.2 A lts for the π-calculus

We now consider the problem of defining a labelled transition system for the π-calculus. This is a rather technical exercise. As a first step, we distinguish four types of actions:

Action α	**Example**
τ	$\overline{x}y.P \mid x(z).Q \xrightarrow{\tau} P \mid [y/z]Q$
input xz	$x(y).P \xrightarrow{xz} [z/y]P$
output $\overline{x}y$	$\overline{x}y.P \xrightarrow{\overline{x}y} P$
bound output $\overline{x}(y)$	$\nu y\ \overline{x}y.P \xrightarrow{\overline{x}(y)} P$

We remark that the first three cases would arise naturally in an extension of *CCS* with ground values too (section 26.4). The real novelty is the *bound output case*. Note that an effect of the bound output action is to *free* the restricted name y. The bound output action carries a bound name and one has to be careful to *avoid conflicts*. Here are some typical situations.

$\nu y\ x(y).P$	The name which is input must not conflict with the fresh one.
$\nu y\ \overline{x}y.P$	The name which is output should become free.
$\nu y\ \overline{x}y.P \mid x(z).Q$	The scope of νy should extend to Q.
$\nu y\ \overline{x}y.P \mid x(z).(y(w).Q)$	The fresh y and the one on the recipient side are distinct.

We fix some conventions concerning free and bound names in actions. (1) All occurrences of a name in an action are free except y in a bound output action $\overline{x}(y)$. (2) Define $\mathsf{fv}(\alpha)$ ($\mathsf{bv}(\alpha)$) as the set of names occurring free (bound) in the action α. (3) Let $n(\alpha) = \mathsf{fv}(\alpha) \cup \mathsf{bv}(\alpha)$.

Based on this, Table 30.2 defines a labelled transition system for the π-calculus. Rules apply up to α-renaming and symmetric rules are omitted.

Exercise 495 (on the lts) *Apply the definition to compute the labelled transitions of the processes discussed above.*

$$\frac{}{x(y).P \xrightarrow{xz} [z/y]P} \qquad \frac{}{\overline{x}y.P \xrightarrow{\overline{x}y} P}$$

$$\frac{P \xrightarrow{\alpha} P' \quad x \notin n(\alpha)}{\nu x\ P \xrightarrow{\alpha} \nu x\ P'} \qquad \frac{P \xrightarrow{\overline{y}x} P' \quad x \neq y}{\nu x\ P \xrightarrow{\overline{y}(x)} P'}$$

$$\frac{P \xrightarrow{xy} P' \quad Q \xrightarrow{\overline{x}y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{xy} P' \quad Q \xrightarrow{\overline{x}(y)} Q' \quad y \notin \mathsf{fv}(P)}{P \mid Q \xrightarrow{\tau} \nu y\ (P' \mid Q')}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \mathsf{bv}(\alpha) \cap \mathsf{fv}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{x(y).P \xrightarrow{\alpha} P'}{!(x(y).P) \xrightarrow{\alpha} P' \mid !(x(y).P)}$$

$$\frac{}{[x = x]P \xrightarrow{\tau} P}$$

Table 30.2: A lts for the π-calculus

Exercise 496 (τ-transitions vs. reduction) *In Table 30.1, we have defined a reduction relation $\rightarrow$ on the π-calculus. Show that reduction and τ-transitions are the same up to structural equivalence. Namely:*

1. *If $P \rightarrow Q$ then $P \xrightarrow{\tau} Q'$ and $Q \equiv Q'$.*
2. *If $P \xrightarrow{\tau} Q$ then $P \rightarrow Q'$ and $Q \equiv Q'$.*

For instance, consider: $(\nu y\ \overline{x}y.P_1 \mid P_2) \mid x(z).P_3$.

The next step is to define a notion of bisimulation on the labelled transition system. This is not completely obvious. Suppose we want to show that P and Q are 'labelled' bisimilar. Further, suppose $P \xrightarrow{\overline{x}(y)} P'$ makes a bound output. What is the condition on Q? We note that bisimilar processes may have different sets of free names. For instance, suppose $P \equiv \nu y\ \overline{x}y.P'$ and $Q \equiv \nu y\ \overline{x}y.Q' \mid R$, with $y \in \mathsf{fv}(R)$. Then the transition $P \xrightarrow{\overline{x}(y)} \cdot$ cannot be matched (literally) by Q because y is free in Q. This leads to the following definition.

Definition 497 (labelled bisimulation) *A binary relation $\mathcal{R}$ on processes is a strong labelled bisimulation if:*

$$\frac{P\ \mathcal{R}\ Q, \qquad P \xrightarrow{\alpha} Q, \qquad \mathsf{bv}(\alpha) \cap \mathsf{fv}(Q) = \emptyset}{\exists\, Q' \ \ Q \xrightarrow{\alpha} Q' \qquad P'\ \mathcal{R}\ Q'} \ .$$

and, as usual, a symmetric condition holds for Q. For the weak case, we replace $\xrightarrow{\alpha}$ by $\stackrel{\alpha}{\Rightarrow}$. We denote with $\sim_L$ ($\approx_L$) the largest strong (weak) labelled bisimulation.

Proposition 498 *Strong (weak) labelled bisimulation is preserved by static contexts.*

Proof. We want to show that strong (weak) labelled bisimulation is preserved by static contexts. Let us abbreviate with νy^* a *possibly empty* list $\nu y_1, \ldots, \nu y_n$. We define a binary relation:

$$\mathcal{R} = \{(\nu y^*\ P \mid R, \nu y^*\ Q \mid R) \mid P \sim_L Q\} \ .$$

We show that $\mathcal{R}$ is a labelled bisimulation. First let us see what goes wrong with the relation one would define for *CCS*:

$$\mathcal{R}' = \{(P \mid R, Q \mid R) \;\mid\; P \sim_L Q\} \ .$$

We can have $P \mid R \stackrel{\tau}{\to} \nu y\ P' \mid R'$ because $P \stackrel{\overline{x}(y)}{\to} P'$ and $R \stackrel{xy}{\to} R'$. Then we just have $Q \mid R \stackrel{\tau}{\to} \nu y\ Q' \mid R'$ and P' is bisimilar to $Q' \ldots$

Let us look again at this case when working with the larger relation $\mathcal{R}$. Suppose $\nu y^*\ P \mid R \stackrel{\tau}{\to} \nu y\ \nu y^*\ P' \mid R'$ because $P \stackrel{\overline{x}(y)}{\to} P'$ and $R \stackrel{xy}{\to} R'$. Then $Q \stackrel{\overline{x}(y)}{\to} Q'$ and $P' \sim_L Q'$. Therefore $\nu y^*\ Q \mid R \stackrel{\tau}{\to} \nu y\ \nu y^*\ Q' \mid R'$ and now:

$$\nu y\ \nu y^*\ P' \mid R'\ \mathcal{R}\ \nu y\ \nu y^*\ Q' \mid R' \ .$$

It is actually possible to develop a little bit of 'bisimulation-up-to-context' techniques (cf. exercise 416) to get rid once and for all of these technicalities. □

Exercise 499 *Complete the proof that labelled bisimulation is preserved by static contexts. Then generalize the proof to the weak case.*

Since labelled bisimulation is preserved by static contexts, we can easily conclude that the largest labelled bisimulation is a contextual bisimulation. We are then left to show that the largest contextual bisimulation is a labelled bisimulation.

Proposition 500 *Contextual bisimulation is a labelled bisimulation.*

Proof. With reference to the proof for *CCS* (chapter 26), the static contexts we have to build are now slightly more elaborate.

Action	Static Context
xy	$[\,] \mid \overline{x}y.o_1 \oplus (o_2 \oplus 0)$
$\overline{x}y$	$[\,] \mid x(z).[z = y](o_1 \oplus (o_2 \oplus 0))$
$\overline{x}(y)$	$[\,] \mid x(z).([z = y_1]w_1 \mid \cdots \mid [z = y_k]w_k \mid (o_1 \oplus (o_2 \oplus 0)))$

where $y_1, \ldots, y_k$ are the free names of the pair of processes under consideration and $w_1, \ldots, w_k$, o_1, o_2 are fresh names. □

Exercise 501 *Complete the proof that the largest contextual bisimulation is a labelled bisimulation.*

Remark 502 (input prefix) *Strictly speaking, the contextual/labelled bisimulation we have studied is not preserved by the input prefix in the sense that it is not true that $P \sim Q$ implies $x(y).P \sim x(y).Q$. For instance, take $P \equiv [z = y]\nu w \overline{z}w.0$ and $Q \equiv 0$. The point is that our semantics compares processes by (implicitly) assuming that all free names are 'constants' in the sense of remark 493. However, the comparison of processes with 'variable' names (like y in the example above) can be reduced to the problem of comparing processes with constant names by considering all possible substitutions of the variable name. While in principle this leads to infinitely many cases, a little analysis shows that it is enough to substitute all names which are free in the processes under consideration plus a fresh one.*

More sophisticated analyses are possible by defining labelled transition systems which perform a symbolic execution of processes. However, these technical developments appear to be of limited interest as they are rarely used in applications. Moreover, there are interesting fragments of the π-calculus where they become useless because the obvious notion of bisimulation we have considered is actually preserved by the input prefix.

30.3 Variations

We have considered a particular variety of the π-calculus with the aim of having a relatively simple labelled transition system and labelled bisimulation. However, a number of variations are possible. We mention a few and discuss their impact on the characterization of labelled bisimulation as contextual bisimulation.

Polyadic channels This is an extension where several names can be transmitted at once. We shall see in chapter 31 that this extension is quite natural when looking at the π-calculus as an intermediate language. The extension calls for some form of typing to guarantee that in a synchronization the sender and the receiver agree on the number of names to be exchanged (cf. Table 31.6). Moreover, in the formalization of the labelled transition system, the structure of the actions is a bit more complicated as several names can be extruded as the result of a communication. The syntax of the actions becomes:

$$\alpha ::= \tau \mid x(y_1, \ldots, y_n) \mid \nu z_1, \ldots, z_m\ \overline{x}(y_1, \ldots, y_n)$$

with the requirement for the output that $z_1, \ldots, z_m$ is a (possibly empty) subsequence of $y_1, \ldots, y_n$ composed of distinct names.

Asynchronous communication This is actually a restriction that requires that an output action cannot prefix another action. Again we shall see that this restriction is suggested by looking at the π-calculus as an intermediate language. From a semantic viewpoint, this restriction entails that an input action is not directly observable and calls for a modification of the bisimulation condition, or equivalently, for a modification of the labelled transition system. In the adapted semantics, one can show, *e.g.*, that the processes $x(y).\overline{x}y$ and 0 are weakly bisimilar.

Other restrictions We may consider restricted communication patterns. For instance, in the context of asynchronous communication we may make the additional assumption that each name has a *unique* receiver. Then in a process $P \equiv !(x(y).P) \mid \overline{x}z$ the output on the x channel is not observable from the environment because the capability of receiving on x is attributed to the process P itself. In another direction, we may want to drop the operation for name comparison. Then, modulo some additional hypotheses, it may be *impossible* to distinguish two names y and z. For instance, imagine y and z are two names for the same service and that the only way the observer can use y and z is to send a message on them. In this case, a theory of bisimulation will have to consider names modulo an equivalence relation.

30.4 Summary and references

The π-calculus is an extension of *CCS* where processes exchange channel names. As in *CCS*, it is possible to define the notions of contextual and labelled bisimulation and show that they coincide. In particular, this comes as a justification of the definition of the labelled bisimulation which is quite technical. The π-calculus is introduced by Milner *et al.* in [MPW92] following earlier work in [EN86]. The books [Hen07, SW01] explore its theory. The encoding of non-deterministic choice is studied in [Nes00] while the notion of bisimulation for asynchronous communication is analyzed in [ACS98].

Chapter 31

Processes vs. functions

We develop the view that the π-calculus is a concurrent extension of an intermediate language used in the compilation of languages of the *ML* family. In chapter 14, we have seen that a standard polyadic, call-by-value, λ-calculus can be put in CPS (continuation passing style), value named form. It turns out that, modulo a simple change of notation, this language corresponds to a deterministic and sequential fragment of the π-calculus. From chapter 15, we know that the CPS and value named transformations are type preserving. Thus modulo the change of notation, the π-calculus inherits the propositional typing discipline of the λ-calculus. As a second step, we consider an extension of the restricted π-calculus, called λ_j-calculus, which allows to express recursive, parallel, and concurrent behaviors and to which the ordinary π-calculus can be compiled. To summarize, we have the following diagram:

$$\lambda \overset{\mathcal{C}_{cps}}{\rightarrow} \lambda_{cps} \overset{\mathcal{C}_{vn}}{\rightarrow} \lambda_{cps,vn} \cong \pi_{restricted} \subset \lambda_j \overset{[\![\_]\!]}{\leftarrow} \pi \ ,$$

which is read from left to right as follows. The call-by-value λ-calculus is put in CPS form (λ_{cps}) and then in CPS value named form ($\lambda_{cps,vn}$). This is equivalent to a restricted form of π-calculus ($\pi_{restricted}$). When this restricted π-calculus is extended with recursive definitions, parallel composition, and a form of *join definition*, it becomes sufficiently expressive to represent the ordinary π-calculus.

31.1 From λ to π notation

Table 31.1 recalls the *value named* λ-calculus in CPS form introduced in chapter 14. In this λ-calculus, all values are named and when we apply the name of a λ-abstraction to the name of a value we create a new copy of the body of the function and replace its formal parameter name with the name of the argument as in:

$$\mathsf{let}\ y = V\ \mathsf{in}\ \mathsf{let}\ f = \lambda x.M\ \mathsf{in}\ @(f,y)\ \rightarrow\ \mathsf{let}\ y = V\ \mathsf{in}\ \mathsf{let}\ f = \lambda x.M\ \mathsf{in}\ [y/x]M\ .$$

We also recall that in the value named λ-calculus the evaluation contexts are sequences of let definitions associating values to names. We can move from λ to π with a simple change of notation which is summarized in Table 31.2.

Example 503 (compilation and reduction simulation) *The following example illustrates the compilation of a λ-term and how the compiled term simulates the original one.*

$$I = \lambda x.x, \quad I' = \lambda x,k.@(k,x), \quad I(z) = !z(x,k).@(k,x),$$
$$K = \lambda x.@(halt,x), \quad K(k) = !k(x).@(halt,x)$$

λ	$@(I,y)$	$\to y$
λ_{cps}	$@(I',y,K)$	$\to @(K,y)$ $\to @(halt,y)$
$\lambda_{cps,vn}$	$\mathsf{let}\ z = I'\ \mathsf{in}\ \mathsf{let}\ k = K\ \mathsf{in}\ @(z,y,k)$	$\to \mathsf{let}\ z = I'\ \mathsf{in}\ \mathsf{let}\ k = K\ \mathsf{in}\ @(k,y)$ $\to \mathsf{let}\ z = I'\ \mathsf{in}\ \mathsf{let}\ k = K\ \mathsf{in}\ @(halt,y)$
π	$\nu z\ (I(z) \mid \nu k\ (K(k) \mid \overline{z}(y,k)))$	$\to \nu z\ (I(z) \mid \nu k\ (K(k) \mid \overline{k}y))$ $\to \nu z\ (I(z) \mid \nu k\ (K(k) \mid \overline{halt}\ y))$

We stress that the π-terms obtained from the compilation are *highly constrained*; the following section 31.2 discusses the restrictions and some possible relaxations.

We can lift the correspondence between λ and π terms to types. In chapter 15, we have shown that the CPS and value named transformations preserve (propositional) types. Modulo the change of notation presented above, this provides a propositional type system for the π-calculus. We recall that *tid* is the syntactic category of *type variables* with generic elements $t, s, \ldots$, and A is the syntactic category of *types* with generic elements $A, B, \ldots$ A *type context* is written $x_1 : A_1, \ldots, x_n : A_n$ with the usual conventions. We also write $x^* : A^*$ for a possibly empty sequence $x_1 : A_1, \ldots, x_n : A_n$, and $\Gamma, x^* : A^*$ for the context resulting from Γ by adding the sequence $x^* : A^*$. Table 31.3 recalls the typing rules for the CPS, value named calculus and presentes the very same rules formulated in the π-notation. For the sake of brevity, we shall omit the type of a term since this type is always the type of results R and write $\Gamma \vdash^{vn} M$ rather than $\Gamma \vdash^{vn} M : R$. A type of the form $A^+ \to R$ corresponds to a channel type $Ch(A^+)$ in π-calculus notation. As expected, one can show that typing in this system is preserved by reduction.

31.2 Adding concurrency

The typed $\lambda_{cps,vn}$-calculus which is the target of the compilation chain is restricted in several ways. We show how these restrictions can be relaxed to obtain a calculus which can represent

Syntax

$$\begin{array}{lll} V & ::= \lambda id^+.M & \text{(values)} \\ M & ::= @(id, id^+) \mid \mathsf{let}\ id = V\ \mathsf{in}\ M & \text{(CPS terms)} \\ E & ::= [\,] \mid \mathsf{let}\ id = V\ \mathsf{in}\ E & \text{(evaluation contexts)} \end{array}$$

Reduction Rule

$$E[@(x, z_1, \ldots, z_n)] \quad \to \quad E[[z_1/y_1, \ldots, z_n/y_n]M] \quad \text{if } E(x) = \lambda y_1, \ldots, y_n.M$$

$$\text{where: } E(x) = \begin{cases} V & \text{if } E = E'[\mathsf{let}\ x = V\ \mathsf{in}\ [\,]] \\ E'(x) & \text{if } E = E'[\mathsf{let}\ y = V\ \mathsf{in}\ [\,]], x \neq y \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Table 31.1: A value named, CPS λ-calculus: $\lambda_{cps,vn}$

λ-interpretation	λ-syntax	π-syntax	π-interpretation
function application	$@(x, y^+)$	$\overline{x}y^+$	calling a service
function definition	$\mathsf{let}\ x = \lambda y^+.M\ \mathsf{in}\ N$	$\nu x\ (!x(y^+).M \mid N)$	service definition

Table 31.2: Changing notation from λ to π

Typing rules with λ notation

$$A \quad ::= tid \mid (A^+ \to R) \quad \text{(types)}$$

$$\frac{\Gamma, y : A^+ \to R \vdash^{vn} N \quad \Gamma, x^+ : A^+ \vdash^{vn} M}{\Gamma \vdash^{vn} \mathsf{let}\ y = \lambda x^+.M\ \mathsf{in}\ N} \qquad \frac{x : A^+ \to R, y^+ : A^+ \in \Gamma}{\Gamma \vdash^{vn} @(x, y^+)}$$

Typing rules with π notation

$$A \quad ::= tid \mid Ch(A^+) \quad \text{(types)}$$

$$\frac{\Gamma, y : Ch(A^+) \vdash^{\pi} N \quad \Gamma, x^+ : A^+ \vdash^{\pi} M}{\Gamma \vdash^{\pi} \nu y\ (!y(x^+).M \mid N)} \qquad \frac{x : Ch(A^+), y^+ : A^+ \in \Gamma}{\Gamma \vdash^{\pi} \overline{x}y^+}$$

Table 31.3: Isomorphic type systems in λ and π notation

the computations of the π-calculus in a rather direct way. The restrictions and the related relaxations concern the possibility of: (1) defining processes by *general recursion*, (2) running processes in *parallel*, and (3) having a concurrent access to a resource. The first two relaxations are rather standard while the third one lends itself to some discussion.

General recursion It is easily shown that in the *typed* $\lambda_{cps,vn}$-calculus all computations terminate; this is a consequence of the termination of the corresponding typed λ-calculus. To allow for infinite computations we introduce recursive definitions, that is in $\mathsf{let}\ x = M\ \mathsf{in}\ N$ we allow M to depend recursively on x. For instance, we can write a non-terminating term $\mathsf{let}\ x = \lambda y.@(x, y)\ \mathsf{in}\ @(x, z)$.

Parallelism The computations in the $\lambda_{cps,vn}$-calculus are essentially sequential since at any moment there is at most one function call which is active. To allow for some parallelism we allow for function calls to be put in parallel as in $@(x, y^+) \mid @(z, w^+)$. We notice that while the resulting calculus allows for parallel computations it fails to represent concurrent computations (cf. discussion in chapter 19). The reason is that two parallel calls to the same function such as:

$$\mathsf{let}\ x = \lambda y.M\ \mathsf{in}\ @(x, z) \mid @(x, w)$$

can be executed in an arbitrary order without affecting the overall behavior of the process. In other terms, the reductions of the calculus are (strongly) *confluent* (yet another example of parallel and deterministic system, cf. chapters 27 and 28).

Concurrency There are several possibilities to introduce concurrent behaviors in the calculus; we consider 3 of them. One possibility is to introduce a mechanism to define a function

which can be called *at most once*. Then two parallel calls to such a function would be concurrent as in:

$$\mathsf{letonce}\ x = \lambda y.M\ \mathsf{in}\ @(x,z) \mid @(x,w)\ .$$

Here the first call that reaches the definition *consumes* it and the following ones are stuck. Another possibility is to associate *multiple definitions* to the same name. This situation is dual to the previous one in that the definitions rather than the calls are concurrent as in:

$$\mathsf{letmlt}\ x = \lambda y.M_1\ \mathsf{or}\ x = \lambda y.M_2\ \mathsf{in}\ @(x,z)\ .$$

Here the first definition that captures the call is executed and the remaining ones are stuck. A third and final possibility consists in introducing and *joining two names* in a definition which is written as:

$$\mathsf{letjoin}\ (x,y)\ \mathsf{in}\ M\ .$$

As usual in a definition, we assume the names x and y are bound in M. The effect of joining the names x and y is that any function transmitted on x can be applied to any argument transmitted on y. In first approximation, the reduction rule for a joined definition is:

$$\mathsf{letjoin}\ (x,y)\ \mathsf{in}\ E[@(x,z) \mid @(y,w)] \to \mathsf{letjoin}\ (x,y)\ \mathsf{in}\ E[@(z,w)]\ .$$

Thus a joined definition allows for a *three-way synchronization* among two function calls and a definition. In turn, this synchronization mechanism allows to simulate a situation where several threads compete to access the same communication channel. An advantage of this approach with respect to those sketched above is that the calculus keeps a *standard* definition mechanism where each name introduced is defined once and for all. At the same time, this relatively modest extension suffices to express the other more elaborate extensions sketched above.

The λ_j-calculus

In Table 31.4, we define the $\lambda_{cps,vn}$-calculus extended with recursive definitions, parallel calls, and join definitions. For the sake of brevity we call the resulting calculus the λ_j-calculus. The λ_j-calculus includes a notion of *structural equivalence* (cf. section 26.3) which is defined by a collection of equations presented in Table 31.4. These equations allow to commute definitions and parallel composition and can be applied in any context. Relying on these equations it is always possible to transform a term into a list of definitions followed by the parallel composition of function calls. Optionally, one can add equations to remove useless definitions such as $\mathsf{let}\ x = V\ \mathsf{in}\ M \equiv M$ if $x \notin \mathsf{fv}(M)$ and equations to commute let/letjoin definitions under suitable conditions on the occurrences of the defined variables. The fact that we look at terms up to structural equivalence, allows to simplify the definition of the evaluation contexts and the reduction rules. Finally, notice that at the bottom of Table 31.4, the rules for visiting the evaluation context are adapted to letjoin definitions.

Example 504 *A closed term* nil *that cannot reduce can be defined as follows:*

$$\mathsf{nil} \equiv \mathsf{letjoin}\ (x,y)\ \mathsf{in}\ @(x,y)\ . \tag{31.1}$$

A letonce *definition can be represented as follows:*

$$\mathsf{letonce}\ x = \lambda z^{+}.M\ \mathsf{in}\ N\ \equiv\ E[@(y,x') \mid N] \tag{31.2}$$

SYNTAX

$$
\begin{array}{lll}
V & ::= \lambda id^{+}.M & \text{(values)} \\
M & ::= \mathsf{let}\ id = V\ \mathsf{in}\ M \mid \mathsf{letjoin}\ (id, id)\ \mathsf{in}\ M \mid (M \mid M) \mid @(id, id^{+}) & \text{(terms)} \\
E & ::= \mathsf{let}\ id = V\ \mathsf{in}\ E \mid \mathsf{letjoin}\ (id, id)\ \mathsf{in}\ E \mid (E \mid M) \mid [\,] & \text{(evaluation contexts)}
\end{array}
$$

STRUCTURAL EQUIVALENCE

Parallel composition is associative and commutative and assuming $x, x' \notin \mathsf{fv}(N)$ we have:

$$\mathsf{let}\ x = V\ \mathsf{in}\ M \mid N \equiv \mathsf{let}\ x = V\ \mathsf{in}\ (M \mid N) \quad \mathsf{letjoin}\ (x, x')\ \mathsf{in}\ M \mid N \equiv \mathsf{letjoin}\ (x, x')\ \mathsf{in}\ (M \mid N)$$

REDUCTION RULES

$$
\begin{array}{rcll}
E[@(x, z_1, \ldots, z_n)] & \to & E[[z_1/y_1, \ldots, z_n/y_n]M] & \text{if } E(x) = \lambda y_1, \ldots, y_n.M \\
E[@(x, y) \mid @(x', z^{+})] & \to & E[@(y, z^{+})] & \text{if } E(x, x') = \mathit{join}
\end{array}
$$

$$
E(x) = \begin{cases}
V & \text{if } E = E'[\mathsf{let}\ x = V\ \mathsf{in}\ [\,]] \\
E'(x) & \text{if } E = E'[\mathsf{let}\ y = V\ \mathsf{in}\ [\,]] \text{ or } E = E'[\mathsf{letjoin}\ (y, y')\ \mathsf{in}\ [\,]] \text{ and } x \neq y, y' \\
E'(x) & \text{if } E = E'[[\,] \mid M] \\
\text{undefined} & \text{otherwise}
\end{cases}
$$

$$
E(x, x') = \begin{cases}
\mathit{join} & \text{if } E = E'[\mathsf{letjoin}\ (x, x')\ \mathsf{in}\ [\,]] \\
E'(x, x') & \text{if } E = E'[\mathsf{let}\ y = V\ \mathsf{in}\ [\,]] \text{ or } E = E'[\mathsf{letjoin}\ (y, y')\ \mathsf{in}\ [\,]] \text{ and } \{x, x'\} \cap \{y, y'\} = \emptyset \\
E'(x, x') & \text{if } E = E'[[\,] \mid M] \\
\text{undefined} & \text{otherwise.}
\end{cases}
$$

Table 31.4: The value named CPS λ-calculus with join: λ_j

$$
\begin{array}{ll}
[\![0]\!] & = \mathsf{nil} \\
[\![\overline{x}y.P]\!] & = \mathsf{letonce}\ k = \lambda\_.[\![P]\!]\ \mathsf{in}\ @(x_\uparrow, y_\downarrow, y_\uparrow, k) \\
[\![x(y).P]\!] & = \mathsf{letonce}\ k = \lambda y_\downarrow, y_\uparrow, k'.([\![P]\!] \mid @(k', \_))\ \mathsf{in}\ @(x_\downarrow, k) \\
[\![!x(y).P]\!] & = \mathsf{let}\ k = \lambda y_\downarrow, y_\uparrow, k'.([\![P]\!] \mid @(k', \_) \mid @(x_\downarrow, k))\ \mathsf{in}\ @(x_\downarrow, k) \\
[\![P \mid Q]\!] & = [\![P]\!] \mid [\![Q]\!] \\
[\![\nu x\ P]\!] & = \mathsf{letjoin}\ (x_\downarrow, x_\uparrow)\ \mathsf{in}\ [\![P]\!]
\end{array}
$$

Table 31.5: Encoding of the π-calculus in λ_j

where: $E = \mathsf{let}\ x' = \lambda z^+.M\ \mathsf{in}\ \mathsf{letjoin}\ (y, x)\ \mathsf{in}\ [\ ]$. *Assuming* $N \equiv @(x, w^+)$, *we have the following reductions:*

$$E[@(y, x') \mid @(x, w^+)] \quad \rightarrow \quad E[@(x', w^+)] \quad \rightarrow \quad E[[w^+/z^+]M]\ .$$

Since the call $@(y, x')$ *is not regenerated and the name* y *remains local, further invocations of* x *will be stuck. Thus if we define:* $\mathsf{letzero}\ x = \lambda z^+.M\ \mathsf{in}\ N\ =\ E[N]$, *we have:*

$$\mathsf{letonce}\ x = \lambda z^+.M\ \mathsf{in}\ @(x, w^+) \stackrel{*}{\rightarrow} \mathsf{letzero}\ x = \lambda z^+.M\ \mathsf{in}\ [w^+/z^+]M\ .$$

A letmlt *definition can be defined as follows:*

$$\mathsf{letmlt}\ x = \lambda y^+.M_1\ \mathsf{or}\ x = \lambda y^+.M_2\ \mathsf{in}\ N \quad = \quad E[@(x', x_1) \mid @(x', x_2) \mid N] \tag{31.3}$$

where $E = \mathsf{letjoin}\ (x', x)\ \mathsf{in}\ \mathsf{let}\ x_1 = \lambda y^+.(M_1 \mid @(x', x_1))\ \mathsf{in}\ \mathsf{let}\ x_2 = \lambda y^+.(M_2 \mid @(x', x_2))\ \mathsf{in}\ [\ ]$. *Assuming that* $N \equiv @(x, w^+)$ *and that the selected function definition is the first one, we have the following reductions:*

$$
\begin{array}{lll}
E[@(x', x_1) \mid @(x', x_2) \mid @(x, w^+)] & \rightarrow & E[@(x_1, w^+) \mid @(x', x_2)] \\
 & \rightarrow & E[[w^+/y^+]M_1 \mid @(x', x_1) \mid @(x', x_2)]\ .
\end{array}
$$

Notice that each call to x_i *regenerates a call* $@(x', x_i)$ *and that this requires a recursive definition of* x_i *where the body of the function associated to* x_i *depends on* x_i *itself.*

Encoding of the (monadic) π-calculus in the λ_j-calculus

Table 31.5 presents an encoding of the monadic π-calculus described in chapter 30 (without name equality) into the λ_j-calculus. The encoding assumes that for each name x in the term of the π-calculus to be encoded we reserve a pair of names $x_\downarrow, x_\uparrow$ in the λ_j-calculus. The intuition is that an input on x is transformed into a call to the name $x_\downarrow$ while an output on x becomes a call to the name $x_\uparrow$. Then the names $x_\downarrow, x_\uparrow$ are joined so that a reduction in λ_j is possible whenever there is at least one call to $x_\downarrow$ and one call to $x_\uparrow$. Not surprisingly, recursive definitions are needed in the encoding of the replicated input. In the translation of the output we use _ as a dummy variable, *i.e.*, a variable which is not used in the body of the function. Correspondingly, in the translation of the input we use the notation $@(k', \_)$ for a call to k' with a dummy argument. By convention, we define $@(k', \_) \equiv \mathsf{letjoin}\ (x, x')\ \mathsf{in}\ @(k', x')$.

Example 505 *The following reduction in the* π*-calculus:*

$$R \equiv \nu x\ (x(y).P \mid \overline{x}z.Q) \rightarrow \nu x\ ([z/y]P \mid Q)\ ,$$

Syntax types and type contexts

$$\begin{array}{lll} A & ::= tid \mid Ch(A^+) & \text{(types)} \\ \Gamma & ::= id : A, \ldots, id : A & \text{(type contexts)} \end{array}$$

Typing rules for the polyadic π-calculus

$$\frac{}{\Gamma \vdash^\pi 0} \qquad \frac{x : Ch(A^+), y^+ : A^+ \in \Gamma \quad \Gamma \vdash^\pi P}{\Gamma \vdash^\pi \overline{x}y^+.P}$$

$$\frac{x : Ch(A^+) \in \Gamma \quad \Gamma, y^+ : A^+ \vdash^\pi P}{\Gamma \vdash^\pi x(y^+).P} \qquad \frac{x : Ch(A^+) \in \Gamma \quad \Gamma, y^+ : A^+ \vdash^\pi P}{\Gamma \vdash^\pi !(x(y^+).P)}$$

$$\frac{\Gamma \vdash^\pi P \quad \Gamma \vdash^\pi Q}{\Gamma \vdash^\pi P \mid Q} \qquad \frac{\Gamma, x : Ch(A^+) \vdash^\pi P}{\Gamma \vdash^\pi \nu x\ P}$$

Typing rules for the λ_j-calculus

$$\frac{\Gamma, x : Ch(A^+) \vdash^j N \quad \Gamma, x : Ch(A^+), y^+ : A^+ \vdash^j M}{\Gamma \vdash^j \mathsf{let}\ x = \lambda y^+.M\ \mathsf{in}\ N} \qquad \frac{\Gamma, x : Ch(Ch(A^+)), y : Ch(A^+) \vdash^j M}{\Gamma \vdash^j \mathsf{letjoin}\ (x, y)\ \mathsf{in}\ M}$$

$$\frac{x : Ch(A^+), y^+ : A^+ \in \Gamma}{\Gamma \vdash^j @(x, y^+)} \qquad \frac{\Gamma \vdash^j M \quad \Gamma \vdash^j N}{\Gamma \vdash^j M \mid N}$$

Table 31.6: Typing rules for the (polyadic) π-calculus and the λ_j-calculus

is simulated as follows by the λ_j-term $E[T] \equiv [\![R]\!]$ where:

$$\begin{array}{ll} E & = \mathsf{letjoin}\ (x_\downarrow, x_\uparrow)\ \mathsf{in}\ \mathsf{letonce}\ k = \lambda y_\downarrow, y_\uparrow, k'.([\![P]\!] \mid @(k', \_))\ \mathsf{in}\ \mathsf{letonce}\ k' = \lambda\_.[\![Q]\!]\ \mathsf{in}\ [\ ] \\ E' & = \mathsf{letjoin}\ (x_\downarrow, x_\uparrow)\ \mathsf{in}\ \mathsf{letzero}\ k = \lambda y_\downarrow, y_\uparrow, k'.([\![P]\!] \mid @(k', \_))\ \mathsf{in}\ \mathsf{letonce}\ k' = \lambda\_.[\![Q]\!]\ \mathsf{in}\ [\ ] \\ E'' & = \mathsf{letjoin}\ (x_\downarrow, x_\uparrow)\ \mathsf{in}\ \mathsf{letzero}\ k = \lambda y_\downarrow, y_\uparrow, k'.([\![P]\!] \mid @(k', \_))\ \mathsf{in}\ \mathsf{letzero}\ k' = \lambda\_.[\![Q]\!]\ \mathsf{in}\ [\ ] \\ T & = @(x_\downarrow, k) \mid @(x_\uparrow, z_\downarrow, z_\uparrow, k') \end{array}$$

$$E[T] \to E[@(k, z_\downarrow, z_\uparrow, k')] \to E'[[z_\downarrow/y_\downarrow, z_\uparrow/y_\uparrow][\![P]\!] \mid @(k', \_)] \to E''[[z_\downarrow/y_\downarrow, z_\uparrow/y_\uparrow][\![P]\!] \mid [\![Q]\!]]\ .$$

Typing the encoding

The type system for the $\lambda_{cps,vn}$-calculus presented in Table 15.2 can be easily adapted to the π-calculus and the λ_j-calculus. Following the notation in Table 31.3, we write the type $A^+ \to R$ as $Ch(A^+)$. Then Table 31.6 spells out the syntax and typing rules for the polyadic π-calculus and the λ_j-calculus. To accommodate recursive definitions, we allow the name introduced by a let definition to occur in the body of the associated function. The rules for the monadic π-calculus are just a special case where the channel type constructor is always applied to exactly one type.

Exercise 506 *Determine the derived typing rules for the terms* nil *(31.1),* letonce *(31.2), and* letmlt *(31.3).*

We can extend to *types* the translation from the monadic π-calculus to the λ_j-calculus. We use $D = Ch(t)$ as the type of the dummy argument in $@(k', \_)$. Then we define a translation

on types and type contexts as follows:

$$\begin{array}{lcl} [\![t]\!] & = & t\ , \\ [\![Ch(A)]\!] & = & Ch(Ch([\![A]\!]), [\![A]\!], Ch(D))\ , \\ [\![\Gamma, x : A]\!] & = & [\![\Gamma]\!], x_{\downarrow} : Ch([\![A]\!]), x_{\uparrow} : [\![A]\!]\ . \end{array}$$

And we can formulate the following type preservation property whose proof is left to the reader.

Proposition 507 *If P is a term of the monadic π-calculus and $\Gamma \vdash^{\pi} P$ then $[\![\Gamma]\!] \vdash^{j} [\![P]\!]$.*

This exercise can be continued by establishing a kind of barbed bisimulation (cf. definition 422) between a (typed) π-calculus process and its translation in the λ_j-calculus.

31.3 Summary and references

The π-calculus can be regarded as a concurrent relaxation of a functional language in CPS, value named form. This fact explains its ability to encode a variety of features of high-level programming languages. Milner in [Mil92] is the first to discuss a translation from the λ-calculus to the π-calculus. Since then a variety of translations have appeared in the literature. The notion of multiple synchonization is commonly found in Petri nets (see, *e.g.*, [Reu90]). In the framework of the π-calculus, the notion of *join definition* is put forward in [FG96]. This work contains a more elaborate definition mechanism than the one we have described here and it sketches a number of sophisticated encodings.

Chapter 32

Concurrent objects

In this chapter, we reconsider *shared memory* concurrency. The $\mathsf{Imp}_{\|}$ model introduced in chapter 19 has a pedagogical value in that it allows to illustrate many interesting problems that arise in concurrency in a relatively simple setting. On the negative side, it is clear that its modelling power is rather limited. First, it does not support the introduction of data structures such as lists, queues, trees, graphs, ... and the related operations on them. Second, the memory model does not allow for the dynamic allocation, manipulation, and possibly disposal of memory locations which is typical of imperative programming. (Incidentally, these considerations are similar to those motivating the move from CCS to the π-calculus.)

Research on concurrent programming in the shared memory model has focused on the issue of programming data structures that allow for concurrent access, *i.e.*, for the concurrent execution of several operations on the data structure, while providing an observable behavior which is 'equivalent' to that of a data structure where the execution of the operations is sequential, *i.e.*, each operation is run from the beginning to the end without interference from the other operations. In a certain technical setting, this property is called *linearizability* and because the technical setting corresponds roughly to that of a *Java*-like concurrent object-oriented programming language one speaks of concurrent and sequential *objects* rather than concurrent and sequential *data structures*, respectively. Also, the *operations* of the data structures correspond to the *methods* of the object.

An important result in this field is the existence of *universal constructions* that transform any 'sequential' object into a 'concurrent' one without introducing locks. Instead of locks, one relies on relatively simple atomic operations such as *compare and set* (cf. example 310). Such concurrent data structures are called *lock-free*. Because of the absence of locks such data structures guarantee a form of collective progress, *i.e.*, there is a guarantee that some operations will be completed while others may be delayed indefinitely. In fact one can go one step further and produce *wait-free* data structures where each operation is guaranteed to terminate in a bounded number of steps. Unfortunately, such universal transformations tend to be rather inefficient and research has focused on both ways to have more efficient constructions in some special cases and on ways to relax the correctness conditions so as to allow for some efficient implementation techniques. Our goal in this chapter is to discuss these issues in a (fragment of a) state of the art programming language (*Java*) and to hint to their formalization. We build on chapter 18 and the reader is supposed to have a superficial knowledge of the *Java* programming language.

```
public class PingPong extends Thread {
    private String word;
    public PingPong(String w){word = w;}                      //constructor
    public void run(){for (;;){System.out.print(word+" ");}} //redefine run
    public static void main (String[] args){
        (new PingPong("ping")).start();
        (new PingPong("pong")).start();} }
```

Table 32.1: Creating threads in *Java*

```
public class WrongCounter {
    private long value = 0;
    public long getValue() { return value; }
    public void increment() {value=value+1; return; } }
```

Table 32.2: A wrong counter

32.1 Review of concurrent programming in *Java*

We review a few basic notions of concurrent programming in *Java* and provide a few examples of concurrent objects.

We start by describing a basic method to create threads (processes) in *Java*. *Java* has a predefined class Thread which in turn has predefined methods start and run. By invoking start on a Thread object, we invoke the run method on it and return immediately. By default the run method does nothing; so to have some interesting behavior one needs to create a class which extends the Thread class and redefines the run method. As an example, in Table 32.1 we define a class PingPong which extends the Thread class and redefines the run method. What the redefined run method does is to print the String value which constitutes the internal state of a PingPong object. The main method creates two PingPong objects, one writing "ping" and the other writing "pong" and starts them in parallel.

Threads running in parallel may share a common object. For instance, suppose the shared object is a *counter* with methods getValue to read the contents of the counter and increment to increment by one its contents. Table 32.2 gives a preliminary (and wrong!) description of a counter class.

Suppose two threads invoke once the increment method on an object of the WrongCounter class. Reading a value from memory, incrementing it, and storing the result back into memory is not an atomic operation in *Java*. As a result, it is quite possible that the final value of the counter is 1 rather than 2. In fact, the *Java* specification does not even guarantee that reading or writing a long variable is an atomic operation. Indeed, a long variable can be stored in two consecutive memory words and the access to such two words does not need to be atomic. In principle, it could happen that by reading a long variable we get a value which is a 'mix' of values written by concurrent threads.

A simple way to solve these issues is to specify that all the methods of the counter are synchronized as in Table 32.3. A thread that invokes a synchronized method on an object implicitly acquires a lock that guarantees exclusive access to the state of the object and releases the lock upon returning from the method. Other threads invoking a synchronized method on the same object at the same time will be delayed. This is a reformulation of an older synchronization mechanism in concurrent programming known as *monitor*. This

```
public class SyncCounter {
    private long value = 0;
    public synchronized long getValue() {return value;}
    public synchronized void increment() {value=value+1; return;} }
```

Table 32.3: A synchronized counter

```
public class CASCounter {
    private AtomicInteger value= new AtomicInteger(0);
    public int getValue(){return value.get();}
    public void increment(){
        int v;
        do { v = value.get(); }
         while (!value.compareAndSet(v, v + 1));
        return;} }
```

Table 32.4: A counter with compareAndSet

approach obviously guarantees linearizability but it can be inefficient.

An alternative approach consists in reducing the granularity of the operations to that of compareAndSet operations. In *Java*, compareAndSet is actually a method which can be invoked on an object of a special 'Atomic' class and which returns true if the comparison is successful and false otherwise. This implementation of the counter is described in Table 32.4 which relies on a AtomicInteger class.

Notice that this time the implementation of the increment method is significantly different. First the value of the counter is read and incremented and then atomically the current value of the counter is compared to the value read and if they are equal then the counter is incremented. In case of contention, this solution relies on *busy waiting* while the previous one relies on a *context switch*. [1]

Table 32.5 presents an implementation of a concurrent stack object using compareAndSet which is known as Treiber's algorithm. The implementation of the push and pop method follows the approach we have already presented for the increment method. First the methods do some speculative work on the side and then they make it visible with a compareAndSet method provided no interference has occurred so far. In this example, we work on objects of the AtomicReference class and the stack is implemented as a linked list of objects of the Node class.

It should be noticed that compareAndSet can only manage a single pointer atomically. More complex operations such as inserting an element in a queue represented as a linked list, may require the (virtual) update of more pointers at once. In this case more sophisticated programming techniques are needed.

[1] The general wisdom is that if the probability of contention is low then busy waiting may be more efficient than context switch. In case of significant contention, *exponential back-off* is a general strategy to improve a busy waiting solution which is used, *e.g.*, to handle collisions in the Ethernet protocol. In our case, it consists in introducing a *delay* after each iteration of the while loop. The delay is chosen randomly from an interval which increases exponentially with the number of iterations.

```
public class Node{
    int value;
    Node next;
    public Node(int v){value=v; next=this;} }  // Node constructor
public class CASStack {
    AtomicReference<Node> head = new AtomicReference<Node>();
    public void push(int v) {
        Node oldHead;
        Node newHead = new Node(v);
        do {oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));}
    public int pop() {
        Node oldHead;
        Node newHead;
        do {oldHead = head.get();
            if (oldHead == null) return -1;   // -1 default value for empty stack
            newHead = oldHead.next;
        } while (!head.compareAndSet(oldHead,newHead));
        return oldHead.value;}}
```

Table 32.5: A stack with compareAndSet

32.2 A specification of a fragment of concurrent *Java*

We introduce the syntax, reduction rules, and typing rules of a tiny imperative and concurrent object-oriented language (called cJ) which is an extension of the (imperative) J language formalized in chapter 18.

We recall that an *object* value is composed of the name of a *class* and a list of references which correspond to the object's *fields*. A *class* is a declaration where we specify how to build and manipulate the objects of the class. In particular, we specify the fields of each object and the methods that allow their manipulation.

As usual, we assume a class Object without fields and methods. Every other class declaration extends a previously defined class and, in particular, we assume a class Thread which extends the Object class with a method start with no arguments and returning an object of the Object class. The effect of invoking the start method on an object of the Thread class is to spawn in parallel the invocation of the run method on the object (if any). In this chapter we assume all fields are *modifiable* (we stick to the imperative version of the language) and denote with R an infinite set of references (cf. chapter 18) with elements $r, r', \ldots$ A reference is a pointer to an object. The *value* v of an object has the shape: $C(r_1, \ldots, r_n)$, for $n \geq 0$, where C is the name of the class to which the object belongs and $r_1, \ldots, r_n$ are the references associated with the modifiable fields of the object. A *heap memory* h is a partial function with finite domain from references to values.

Table 32.6 defines the syntactic category of *expressions* for cJ. As usual (cf. chapter 18), to define the reduction rules, it is convenient to include values in the syntactic category of expressions. However, it is intended that expressions in a source program do *not* contain values. As in chapter 18, a *program* is composed of a list of class declarations and a distinguished expression where the computation starts. The final value of the distinguished expression can be taken as the output of the program. Among the variables, we reserve this to refer to the object on which a method is invoked. Also, we reserve the names start and run for methods of

e	::=	id	|	(variable)
		v	|	(value)
		new $C(e_1, \ldots, e_n)$	|	(object generation)
		$e.f$	|	(field read)
		$e.m(e_1, \ldots, e_n)$	|	(method invocation)
		$(C)(e)$	|	(casting)
		$e.f := e$	|	(field write)
		$e; e$	|	(sequentialization)
		atomic(e)		(atomicity)

Table 32.6: Expressions in cJ

objects of the Thread class. A *well-formed* program must satisfy certain conditions concerning fields and methods which are specified in chapter 18.

In order to define the reduction rules, it is convenient to introduce the syntactic category of sequential *evaluation contexts* which correspond to a *call-by-value, left to right* reduction strategy and which are defined as follows:

$$\begin{array}{rl} E ::= & [\,] \mid \text{new } C(v^*, E, e^*) \mid E.f \mid E.m(e^*) \mid v.m(v^*, E, e^*) \mid \\ & (C)(E) \mid E.f := e \mid v.f := E \mid E; e \qquad \text{(evaluation contexts)} \end{array} \tag{32.1}$$

Along with the notion of evaluation context, we introduce a notion of *redex*, namely an expression which (up to some type checks) is ready to reduce.

$$\Delta ::= \text{new } C(v^*) \mid v.f \mid v.m(v^*) \mid (D)(v) \mid v.f := v \mid v; e \mid \text{atomic}(e) \qquad \text{(redexes)} \tag{32.2}$$

The reduction of an expression involving the start method may produce the spawn of an expression to be evaluated in parallel. Consequently, we consider a judgment of the shape:

$$(e, h) \xrightarrow{\mu} (e', h')$$

where μ is a (possibly empty) finite multi-set of expressions of the shape $v.\text{run}()$. Equivalently, μ can be regarded as a finite sequence of expressions where the order is irrelevant. As usual, we denote with $\emptyset$ the empty multi-set and moreover we write $\rightarrow$ as an abbreviation for $\xrightarrow{\emptyset}$. Table 32.7 introduces the rules for reducing expressions and configurations (the last two rules). The first 8 rules are driven by the shape of the redexes specified in grammar (32.2). In the rule for atomic, we write:

$$(e_1, h_1) \xrightarrow{\mu}{}^{*} (e_n, h_n) \text{ for } (e_1, h_1) \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_{n-1}} (e_n, h_n) \text{ and } \mu = \mu_1 \cup \cdots \cup \mu_{n-1} \ .$$

The rule for an atomic expression may spawn several threads, but their actual reduction may only start once the evaluation of the atomic expression is completed. Also note that in the proposed semantics $\text{atomic}(v) \rightarrow v$. The following rule allows to cross an evaluation context. The last two rules, explain how to reduce a *configuration* which is a triple (e, μ, h) composed of a main expression, a multi-set of secondary expressions (initially empty), and a heap (initially empty too). This amounts to select non-deterministically one of the expressions and reduce it according to the rules above.

Recall that at the beginning of the computation we can assume that the multi-set of expressions μ contains no references and that the heap h is empty. Then the reduction rules

$$\frac{r^* \text{ distinct and } \{r^*\} \cap dom(h) = \emptyset}{(\mathsf{new}\ C(v^*), h) \to (C(r^*), h[v^*/r^*])} \quad \text{(object generation)}$$

$$\frac{\mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n \quad 1 \leq i \leq n}{(C(r_1, \ldots, r_n).f_i, h) \to (h(r_i), h)} \quad \text{(field read)}$$

$$\frac{\mathit{mbody}(m, C) = \lambda x_1 \cdots x_n.e}{(C(r^*).m(v_1, \ldots, v_n), h) \to ([v_1/x_1, \ldots, v_n/x_n, C(r^*)/\mathsf{this}]e, h)} \quad \text{(method invocation)}$$

$$\frac{C \leq \mathsf{Thread}}{(C(r^*).\mathsf{start}(), h) \overset{C(r^*).\mathsf{run}()}{\to} (\mathsf{Object}(), h)} \quad (\mathsf{start}\ \text{invocation})$$

$$\frac{C \leq D}{((D)(C(r^*)), h) \to (C(r^*), h)} \quad \text{(casting)}$$

$$\frac{\mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n \quad 1 \leq i \leq n}{(C(r_1, \ldots, r_n).f_i := v, h) \to (\mathsf{Object}(), h[v/r_i])} \quad \text{(field write)}$$

$$\frac{}{(v; e, h) \to (e, h)} \quad \text{(sequentialization)}$$

$$\frac{(e, h) \overset{\mu}{\to}{}^* (v, h')}{(\mathsf{atomic}(e), h) \overset{\mu}{\to} (v, h')} \quad \text{(atomicity)}$$

$$\frac{(e, h) \overset{\mu}{\to} (e', h')}{(E[e], h) \overset{\mu}{\to} (E[e'], h')} \quad \text{(evaluation context)}$$

$$\frac{(e, h) \overset{\mu'}{\to} (e', h')}{(e, \mu, h) \to (e', \mu \cup \mu', h')} \quad \text{(main expression)}$$

$$\frac{(e', h) \overset{\mu'}{\to} (e'', h')}{(e, \{\!|e'|\!\} \cup \mu, h) \to (e, \{\!|e''|\!\} \cup \mu \cup \mu', h')} \quad \text{(secondary expression)}$$

Table 32.7: Small step reduction rules for cJ

are supposed to maintain the following invariant: for all reachable configurations (e, μ, h), all the references in e, μ and all the references that appear in a value in the codomain of the heap h are in the domain of definition of the heap ($dom(h)$). This guarantees that whenever we look for a fresh reference it is enough to pick a reference which is not in the domain of definition of the current heap.

Following the discussion in chapter 18 (notably on the typing of casting), we present a type system for cJ. As usual, a type environment Γ has the shape $x_1 : C_1, \ldots, x_n : C_n$ and we consider typing judgments of the shape: $\Gamma \vdash e : C$. Table 32.8 specifies the rules to type expressions that do not contain values or references (as source programs do). The rules governing the typing of class declarations and programs are those specified for the sequential fragment J in chapter 18. The typed language, but for the atomic operator, can be regarded as a fragment of the *Java* programming language. General, but not very efficient, methods to compile the atomic operator have been proposed. The basic idea is to follow an *optimistic strategy* such as the one described in chapter 22. Unlike in the $\mathsf{Imp}_{\|}$ language however, in cJ,

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C} \qquad \frac{\begin{array}{c}\mathit{field}(C) = f_1 : D_1, \ldots, f_n : D_n \\ \Gamma \vdash e_i : C_i, \quad C_i \leq D_i, \quad 1 \leq i \leq n\end{array}}{\Gamma \vdash \mathsf{new}\ C(e_1, \ldots, e_n) : C}$$

$$\frac{\Gamma \vdash e : C \quad \mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n}{\Gamma \vdash e.f_i : C_i} \qquad \frac{\begin{array}{c}\Gamma \vdash e : C \quad \mathit{mtype}(m, C) = (C_1, \ldots, C_n) \to D \\ \Gamma \vdash e_i : C_i' \quad C_i' \leq C_i \quad 1 \leq i \leq n\end{array}}{\Gamma \vdash e.m(e_1, \ldots, e_n) : D}$$

$$\frac{\Gamma \vdash e : C \quad C \leq \mathsf{Thread}}{\Gamma \vdash e.\mathsf{start}() : \mathsf{Object}} \qquad \frac{\Gamma \vdash e : D}{\Gamma \vdash (C)(e) : C}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : C \quad \mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n \\ \Gamma \vdash e' : D_i \quad D_i \leq C_i\end{array}}{\Gamma \vdash e.f_i := e' : \mathsf{Object}} \qquad \frac{\Gamma \vdash e_1 : C_1 \qquad \Gamma \vdash e_2 : C_2}{\Gamma \vdash e_1; e_2 : C_2}$$

$$\frac{\Gamma \vdash e : C}{\Gamma \vdash \mathsf{atomic}(e) : C}$$

Table 32.8: Typing rules for cJ program expressions

and more generally in *Java*, it is not possible to determine statically the collection of object's fields which will be affected by the atomic transaction. In first approximation, the atomic execution of an expression e is compiled into a *speculative* execution of the expression e which maintains a list of object's fields which are read and/or written along with their updated values. At the end of the speculative execution, if certain coherence conditions are met, the computation is *committed*, and otherwise the computation is re-started.

Exercise 508 *Building on proposition 305, formulate and prove a subject reduction property for the typed* cJ *language.*

32.3 Summary and references

We have reviewed some basic notions of concurrent programming in *Java* and formalized the reduction and typing rules of a tiny fragment of it. *Java*'s synchronization builds on the notion of *monitor* which was described in [Hoa74, Han75]. The notion of *linearizability* is introduced in [HW90]. In a nutshell, linearizability means that the execution of the body of the methods of the concurrent object looks atomic. The introduction of a universal construction to transform any 'sequential' object into a 'concurrent' one without introducing locks is due to [Her91]. The book [HS08] is a good survey of the state of the art in this area.

Building a comprehensive model and proof methodology for concurrent object-oriented languages is the subject of ongoing research and seems to require the combination of insights coming from different directions. We mention a few of them. The original work on *linearizability* is described in a rather *ad hoc* model. Recent work, see, *e.g.*, [FORY10] connects this notion with that of observational refinement. Also proofs of linearizability can be quite complex and research is active on developing proof methods and on mechanizing them. A problem related to linearizability is that of finding *general and efficient* ways of compiling the atomic execution of a sequence of statements (see, *e.g.*, [DSS06] for a proposal). The work on so called *separation logic* [Rey02] has focused on developing scalable methods to reason in some

kind of Hoare logic on programs with pointers. However most of the work has been devoted to a simple model which roughly corresponds to the Imp model extended with operators to allocate, modify, and dispose memory locations (cf. chapter 23). Much remains to be done to port the approach to object oriented and/or concurrent programs as it is witnessed by the rather undisciplined proliferation of 'separation logics'. The whole enterprise of (object-oriented) concurrent programming with shared memory is on shaky foundations because it may rely on optimistic or plainly wrong hypotheses on the *memory model*. For instance, see [MPA05] for a tentative definition of a realistic *Java* memory model and [SVN+13] for the implications on compilers' correctness.

Bibliography

[AB84] Didier Austry and Gérard Boudol. Algèbre de processus et synchronisation. *Theor. Comput. Sci.*, 30:91–131, 1984.

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[AC98] Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda Calculi*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[ACS98] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Theor. Comput. Sci.*, 195(2):291–324, 1998.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[Ama09] Roberto M. Amadio. On stratified regions. In *APLAS, Springer LNCS 5904*, pages 210–225, 2009.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies, Spring Joint Computing Conference*, pages 483–485, 1967.

[AN01] André Arnold and Damian Niwinski. *Rudiments of μ-calculus*, volume 146 of *Studies in logic and the foundations of mathematics*. North Holland, 2001.

[BA84] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst.*, 6(3):333–344, 1984.

[Bar84] Hendrik Pieter Barendregt. *The lambda calculus; its syntax and semantics*. North-Holland, 1984.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.

[BC84] Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Springer LNCS 197*, pages 389–448, 1984.

[BC92] Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[BdS96] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *IEEE Trans. Software Eng.*, 22(4):256–266, 1996.

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, , and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.

[Bou91] Frédéric Boussinot. Reactive C: An extension of C to program reactive systems. *Softw., Pract. Exper.*, 21(4):401–428, 1991.

[Bou10] Gérard Boudol. Typing termination in a higher-order concurrent imperative language. *Inf. Comput.*, 208(6):716–736, 2010.

[Bra96] Julian C. Bradfield. The modal mu-calculus alternation hierarchy is strict. In *CONCUR, Springer LNCS 1119*, pages 233–246, 1996.

[Bro96] Stephen D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Inf. Comput.*, 76(2/3):138–164, 1988.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[CHL96] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2):362–397, 1996.

[Chl10] Adam Chlipala. A verified compiler for an impure functional language. In *ACM POPL*, pages 93–106, 2010.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[Cob64] Alan Cobham. The intrinsic computational difficulty of functions. In *Proc. of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science, Y. Bar-Hillel ed.*, pages 24–30. North Holland, 1964.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *ACM POPL*, pages 178–188, 1987.

[dB72] Nicolaas G. de Brujin. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[DG94] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *ACM POPL*, pages 70–83, 1994.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[DSS06] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC, Springer LNCS 4167*, pages 194–208, 2006.

[EN86] Uffe Engberg and Mogens Nielsen. A calculus of communicating systems with label passing. Technical report, DAIMI PB 208, University Aarhus, 1986.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *ACM POPL*, pages 372–385, 1996.

[Flo67] Robert W. Floyd. Assigning meaning to programs. In *Proc. Symp. on Applied Maths*, volume 19, pages 19–32. American Math. Soc., 1967.

[FORY10] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. *Proc. of the Second Scandinavian Logic Symposium*, 63:63–92, 1971.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[Gri77] David Gries. An exercise in proving parallel programs correct. *Commun. ACM*, 20(12):921–930, 1977.

[GS96] Jan Friso Groote and M. P. A. Sellink. Confluence for process verification. *Theor. Comput. Sci.*, 170(1-2):47–81, 1996.

[Gun92] Carl Gunter. *Semantics of programming languages.* MIT Press, 1992.

[Han75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Trans. Software Eng.*, 1(2):199–207, 1975.

[Hen07] Matthew Hennessy. *A distributed Pi-calculus.* Cambridge University Press, 2007.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[Hig52] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2(7):326336, 1952.

[Hin69] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:2960, 1969.

[HJ90] Hans Hansson and Bengt Jonsson. A calculus for communicating systems with time and probabilities. In *IEEE Real-Time Systems Symposium*, pages 278–287, 1990.

[HM92] Maurice Herlihy and J. Eliot B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 3(3):304–311, 1992.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ACM-ISCA*, pages 289–300, 1993.

[Hoa69] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[How96] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124(2):103–112, 1996.

[HR95] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Inf. Comput.*, 117(2):221–239, 1995.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[HY95] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theor. Comput. Sci.*, 151(2):437–486, 1995.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[KÖ26] Dés König. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae*, 8:114–134, 1926.

[Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[KB70] Donald Knuth and Peter Bendix. Simple word problems in universal algebra. In *Universal Algerbras, J. Leech (ed.)*, pages 263–297. Pergamon Press, 1970.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[KR81] Hsiang-Tsung Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[KR90] Richard Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of theoretical computer science: algorithms and complexity, vol. A, J. van Leeuven (ed.).* Elsevier, 1990.

[Kru60] Joseph Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.

[KTU90] Assaf Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is dexptime-complete. In *CAAP*, volume 431 of *Springer LNCS*, page 206220, 1990.

[Lan64] Peter Landin. The mechanical evaluation of expressions. *The Computer Journal (British Computer Society)*, 6(4):308–320, 1964.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *ACM POPL*, pages 144–154, 1993.

[Len96] Giacomo Lenzi. A hierarchy theorem for the μ-calculus. In *ICALP, Springer LNCS 1099*, pages 87–97, 1996.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM POPL*, pages 42–54, 2006.

[Ler09] Xavier Leroy. Mechanized semantics, with applications to program proof and compiler verification. Technical report, Marktoberdorf Summer School, 2009.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM POPL*, pages 47–57, 1988.

[LM82] Damas Luis and Robin Milner. Principal type-schemes for functional programs. In *ACM POPL*, pages 207–212, 1982.

[LMWF94] Nancy Lynch, Michael Merritt, William Weil, and Alan Fekete. *Atomic transactions*. Morgan Kaufmann Publishers Inc., 1994.

[LS89] Kim Guldstrand Larsen and Arne Skou. Bisimulation through probabilistic testing. In *ACM POPL*, pages 344–352, 1989.

[Lyn96] Nancy Lynch. *Distributed algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[Mai90] Harry Mairson. Deciding ML typability is complete for deterministic exponential time. In *ACM POPL*, pages 382–401, 1990.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–374, 1978.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25:267–310, 1983.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mil95] Robin Milner. *Communication and concurrency*. Prentice Hall International, 1995.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2/3):211–249, 1988.

[Mit96] John C. Mitchell. *Foundations for programming languages*. MIT Press, 1996.

[Mit03] John C. Mitchell. *Concepts in programming languages*. Cambridge University Press, 2003.

[MM93] Robin Milner and Faron Moller. Unique decomposition of processes. *Theor. Comput. Sci.*, 107(2):357–363, 1993.

[MMH96] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *ACM POPL*, pages 271–283, 1996.

[MP67] John McCarthy and James Painter. *Correctness of a compiler for arithmetic expressions*, volume 19 of *Mathematical aspects of computer science, Symposia in Applied Mathematics*. North Holland, 1967.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[MP05] Louis Mandel and Marc Pouzet. ReactiveML: a reactive extension to ML. In *ACM PPDP*, pages 82–93, 2005.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *ACM POPL*, pages 378–391, 2005.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[MS92] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *ICALP*, pages 685–695, 1992.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.

[MWCG99] Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[NE00] Leonor Prensa Nieto and Javier Esparza. Verifying single and multi-mutator garbage collectors with Owicki-Gries in Isabelle/HOL. In *Mathematical Foundations of Computer Science, Springer LNCS 1893*, pages 619–628, 2000.

[Nes00] Uwe Nestmann. What is a "good" encoding of guarded choice? *Inf. Comput.*, 156(1-2):287–319, 2000.

[New42] Maxwell Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[Nie03] Leonor Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In *ESOP, Springer LNCS 2618*, pages 348–362, 2003.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.

[NS94] Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, ATP: theory and application. *Inf. Comput.*, 114(1):131–178, 1994.

[NW63] Crispin Nash-Williams. On well-quasi-ordering finite trees. *Proc. Of the Cambridge Phil. Soc.*, 59(04):833–883, 1963.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[Pan09] Prakash Panangaden. *Labelled Markov Processes*. Imperial College Press, 2009.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Conference in Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981.

[PCG+15] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2015.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004. First appeared in 1981.

[Put94] Martin L. Puterman. *Markov decision process. Discrete stochastic dynamic programming*. John Wiley & Sons, 1994.

[PW97] Anna Philippou and David Walker. On confluence in the pi-calculus. In *ICALP, Springer LNCS 1256*, pages 314–324, 1997.

[Rab63] Michael O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963.

[Reu90] Christophe Reutenauer. *The mathematics of Petri nets*. Prentice Hall, 1990.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, 1974.

[Rey98] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. First appeared in 1972.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Ros84] Harvey Rose. *Subrecursion. Functions and hierarchies*, volume 9 of *Oxford logic guides*. Oxford University Press, 1984.

[RS83] Neil Robertson and Paul Seymour. Graph minors I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):3961, 1983.

[SBS04] Manuel Serrano, Frédéric Boussinot, and Bernard P. Serpette. Scheme fair threads. In *ACM PPDP*, pages 203–214, 2004.

[SL95] Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *ACM PODC*, pages 204–213, 1995.

[Sta79] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theor. Comput. Sci.*, 9:67–72, 1979.

[Sti88] Colin Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theor. Comput. Sci.*, 58:347–359, 1988.

[SVN+13] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[SW01] Davide Sangiorgi and David Walker. *The pi-calculus: a theory of mobile processes*. Cambridge University Press, 2001.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[TvD88] Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in mathematics. An introduction. Volume I*. North-Holland, 1988.

[Var85] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *IEEE FOCS*, pages 327–338, 1985.

[vdS87] Jan L. A. van de Snepscheut. Algorithms for on-the-fly garbage collection, revisited. *Inf. Process. Lett.*, 24(4):211–216, 1987.

[Vit05] Giuseppe Vitali. Sul problema della misura dei gruppi di punti di una retta. *Tipografia Gamberini et Parmeggiani*, 1905.

[vR01] Femke van Raamsdonk. On termination of higher-order rewriting. In *Rewriting Techniques and Applications, Springer LNCS 2051*, pages 261–275, 2001.

[Wel99] Joe B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1-3):111–156, 1999.

[Win89] Glynn Winskel. A note on model checking the modal nu-calculus. In *ICALP, Springer LNCS 372*, pages 761–772, 1989.

[Win93] Glynn Winskel. *The formal semantics of programming languages*. MIT Press, 1993.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

[XdRH97] Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.

[Yi91] Wang Yi. CCS + time = an interleaving model for real time systems. In *ICALP, Springer LNCS 510*, pages 217–228, 1991.

Index