Polymorphic Regular Tree Types and Patterns

Jérôme Vouillon
CNRS and Université Paris 7
Jerome.Vouillon@pps.jussieu.fr

Abstract

We propose a type system based on regular tree grammars, where algebraic datatypes are interpreted in a structural way. Thus, the same constructors can be reused for different types and a flexible subtyping relation can be defined between types, corresponding to the inclusion of their semantics. For instance, one can define a type for lists and a subtype of this type corresponding to lists of even length. Patterns are simply types annotated with binders. This provides a generalization of algebraic patterns with the ability of matching arbitrarily deep in a value. Our main contribution, compared to languages such as XDuce and CDuce, is that we are able to deal with both polymorphism and function types.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – Patterns, Polymorphism, and Data Types and Structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs – Type Structure

General Terms Algorithms, Design, Languages, Theory.

Keywords Polymorphism, Subtyping, Tree Automata.

1. Introduction

We investigate an ML-style type system, with variants, function types and prenex polymorphism, but trade type inference for a powerful subtyping relation. Our main motivation for this work is the design of an extension of XDuce [7] with function types and polymorphism, which are crucial for programming in the large.

The type system is based on regular tree grammars. Types are similar to the algebraic datatypes of functional languages. For instance, we can define a list `Cons (1, Cons (2, Nil))`. A type for this value can be specified using the following definition of a type `list` with two variants.

```
type list =
    Cons (int, list)
  | Nil
```

This definition is usually interpreted as specifying an opaque type `list` with two constructors `Cons` and `Nil` of respective types `int→list→list` and `list`. We rather interpret it in a structural way: a value has type `list` if it is the variant `Nil` or if it is a variant of constructor `Cons` with one argument of type `int` and one argument recursively of type `list`. But we can also have a value `Cons (1, 2)`, of type `Cons (int, int)`, with the same constructor but arguments of different types.

Reusing the same constructor for different types make it possible to define refinements of a given type. For instance, here are the type of non empty lists and the type of lists of even length.

```
type non_empty_list = Cons (int, list)

type even_length_list =
    Cons (int, Cons (int, even_length_list))
  | Nil
```

e	$::=$	x	variable
		$\lambda\pi.e$	abstraction
		$e\ e$	application
		$\text{let } x = e \text{ in } e$	let construction
		$c(e,e)$	variant
		$()$	unit
		$\text{match } e \text{ with } \overline{\pi_i \rightarrow e_i}$	pattern matching
		$\Lambda\alpha.e$	type abstraction
		$e\ [\tau]$	type application

Figure 1. Syntax of Expressions e

There is a flexible subtyping relation between types. In particular, both even lists and non-empty lists are subtype of the list type. Type inference seems intractable. Thus, the type of function parameters must be explicitly given. Type application is also explicit, though it can sometimes be inferred.

Patterns are simply types annotated with binders. This provides a generalization of algebraic patterns which the ability of matching arbitrarily deep in a value. For instance, this is a recursive pattern which extracts the last element of a non-empty list:

```
last = Cons(int, last) | Cons(x : int, Nil)) .
```

The paper is organized as follows. We start by describing a calculus with a typeful semantics. We then present some interesting semantic properties of the subtyping relation. As a typeful semantics can be difficult to implement efficiently, we finally isolate a subset of the calculus which does not require runtime type operations and should be easier to implement efficiently.

2. Typed Semantics

We provide a standard presentation of our calculus: syntax, small-step reduction semantics, typing rules, and soundness. For the sake of clarity, we start from high-level constructions and progressively present the details.

2.1 Expressions

The grammar of expressions e is given in figure 1. The syntax of the calculus is a blend of a mini ML (let construction, variants, unit and pattern matching) and System F (explicit type abstraction and application). Most constructions are standard. We comment the most interesting ones below.

We assume given infinite sets of *variables* x, *type variables* α and *constructors* c. Expressions are considered up to renaming of bound variables. The symbol π denotes a *pattern* while the symbol τ denotes a *type*. These two concepts are described in section 2.3.

The abstraction construction $\lambda\pi.e$ is a generalization of the usual construction $\lambda(x : \tau).e$, which can be encoded here by using the pattern $x : \tau$ as parameter (this pattern matches values of type

$$\text{let } x = v \text{ in } e \longrightarrow e[v/x] \qquad (\Lambda\alpha.e)\ [\tau] \longrightarrow e[\tau/\alpha]$$

$$\frac{v \lhd \pi \rightsquigarrow \theta}{(\lambda\pi.e)\ v \longrightarrow \theta(e)} \qquad \frac{v \lhd \pi_j \rightsquigarrow \theta}{\text{match } v \text{ with } \overline{\pi_i \to e_i} \longrightarrow \theta(e_j)}$$

$$\frac{e \longrightarrow e'}{F[e] \longrightarrow F[e']}$$

Figure 2. Reduction Relation $e \longrightarrow e$

τ and binds them to the variable x). For the sake of simplicity, we only consider variants $c(e, e)$ of arity two. Our work can be easily generalized with variants of arbitrary arity[1]. The pattern matching construction is composed of an input expression e and a finite number of branches $\pi_i \to e_i$. When branches are written explicitly, we separate them by a vertical bar. For instance, we write

$$\text{match } e \text{ with } \pi_1 \to e_1 \mid \pi_2 \to e_2$$

for a pattern with two branches. As the type system features subtyping, we believe type inference is not tractable. Hence, type abstraction and application are explicit. Partial inference of type applications should be possible in simple cases using the algorithm proposed by Hosoya, Frisch and Castagna [6]. But, it is not always possible to infer a best type due to function type contravariance.

2.2 Semantics

We give a small step semantics to the calculus. *Values* v are a subgrammar of expressions.

$$\begin{array}{lcl} v & ::= & x \\ & & c(v, v) \\ & & () \\ & & \lambda\pi.e \\ & & \Lambda\alpha.e \end{array}$$

Often, this category of expressions is called *non-expansive expressions* and values do not include variables x. In practice, there is no inconvenience in using this larger category as values and this avoid the need for two similar definitions.

The *reduction relation* $e \longrightarrow e$ is defined in figure 2. The two rules for the let construction and the type application are standard. (The notation $e[v/x]$ stands for the usual capture avoiding *substitution* of the variable x by the value v in the expression e, while the notation $e[\tau/\alpha]$ stands for the substitution of the type variable α by the type τ in the expression e.)

The rules for the application and the pattern matching construction bear some similarity. Both make use of a matching relation $v \lhd \pi \rightsquigarrow \theta$ (defined in section 2.4). This relation asserts that the value v matches the pattern π producing a substitution θ (that is, a finite mapping from variables x to values v). We write $\theta(e)$ for the simultaneous substitution in an expression e of these variables by the corresponding values. In the case of the application, the applied value v must match the input pattern π of the function, producing a substitution θ. The application then reduces to the body e of the function where variables bound in the pattern have been substituted. For the pattern matching construction, the input value v must match the pattern π_j of one of the branches. The construction then reduces to the body e_j of this branch where variables bound in the pattern have been substituted. Note that the pattern matching semantics is not deterministic.

[1] For instance, by encoding an n-ary variant $c(e_1, \ldots, e_n)$ using variants of arity two and the unit expression: $c((), *(e_1, *(e_2, \ldots *(e_n, ()) \ldots)))$, where $*$ is a distinguished constructor.

π	::=	$c(\pi, \pi)$	variant pattern
		$()$	unit pattern
		$\pi \to \pi$	function pattern
		$\pi \cup \pi$	pattern union
		$\bot$	empty pattern
		$\mu(\alpha)\pi$	recursive pattern
		α	pattern variable
		$x : \pi$	binder pattern
		$_$	wildcard

Figure 3. Syntax of Patterns π

$$c(\pi_1, \pi_2) \xrightarrow{c} (\pi_1,\pi_2) \qquad \pi_1 \cup \pi_2 \xrightarrow{\epsilon} \pi_1 \qquad \pi_1 \cup \pi_2 \xrightarrow{\epsilon} \pi_2$$

$$\mu(\alpha)\pi \xrightarrow{\epsilon} \pi[\mu(\alpha)\pi/\alpha] \qquad x : \pi \xrightarrow{\epsilon} \pi$$

Figure 4. Transition Relations $\pi \longrightarrow^c (\pi,\pi)$ and $\pi \longrightarrow^\epsilon \pi$

The last rule specifies where the reduction may occur deeper in an expression. It is defined using *frames* F (that is, one-hole expression pieces) defined by the following grammar.

$$\begin{array}{lcl} F & ::= & c(_, e) \\ & & c(v, _) \\ & & _\ e \\ & & v\ _ \\ & & _\ [\tau] \\ & & \text{let } x = _ \text{ in } e \\ & & \text{match } _ \text{ with } \overline{\pi_i \to e_i} \end{array}$$

Often, the reduction relation is specified using contexts (that is, full expressions with one hole), rather than frames. However, frames are sufficient for our needs and are simpler to reason with as they are not defined inductively.

2.3 Patterns and Types

The syntax of *patterns* π is given in figure 3. Their semantics will be given in section 2.4. Variant patterns $c(\pi, \pi)$, the unit pattern $()$ and function patterns $\pi \to \pi$ match the corresponding values. The binary union of patterns $\pi \cup \pi$ and the empty pattern $\bot$ (empty union) make it possible to define arbitrary unions of patterns. Recursive patterns $\mu(\alpha)\pi$ can be used to explore a value arbitrarily deeply. A pattern variable α is either introduced by this construction or by a type abstraction $\Lambda\alpha.e$. A binder pattern $x : \pi$ extracts a subvalue and binds it to a variable x. Finally, the wildcard pattern $_$ matches anything.

Patterns are considered up to renaming of bound variables. We write $\pi[\pi'/\alpha]$ for the substitution of the variable α by the pattern π' in the pattern π.

We find it convenient to define two relations $\pi \longrightarrow^c (\pi,\pi)$, between three patterns (a source pattern and two target patterns) and a constructor, and $\pi \longrightarrow^\epsilon \pi$, between two patterns (figure 4). These relations can be understood as the transition relations of a tree automaton. Indeed, looking at the semantics of patterns in section 2.4, one can see that if first we have a *transition* $\pi \longrightarrow^c (\pi_1,\pi_2)$, second the value v_1 matches the pattern π_1, and third the value v_2 matches the pattern π_2, then the value $c(v_1, v_2)$ matches the pattern π. Likewise, if we have an *epsilon transition* $\pi \longrightarrow^\epsilon \pi'$ and the value v matches the pattern π', then this value also matches the pattern π. As an example, the transition relations for a simple pattern are depicted in figure 5. Note that a transition $\pi \longrightarrow^c (\pi_1,\pi_2)$ is represented by two arrows starting from the same state π and labelled by the constructor c. The two relations

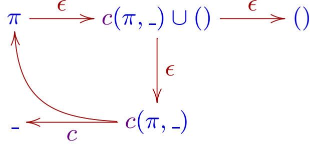

Figure 5. Transitions for Pattern $\pi = \mu(\alpha)(c(\alpha, _) \cup ())$

do not completely characterize the behavior of a pattern. Indeed, some kind of acceptance relation would be required to describe the semantics of the unit pattern, the function patterns and the wildcard pattern. Furthermore, binders are not taken into account. Still, these relations are sufficient for our needs.

The semantics of some patterns is unclear. For instance, should the recursive pattern $\mu(\alpha)\alpha$ matches all values, no value, or something in-between? For this pattern, a reasonable choice can be either the minimal or the maximal interpretation. However, the meaning of a pattern such as $\mu(\alpha)(\alpha \cup (\alpha \rightarrow ()))$ is not clear at all, as there does not exist a minimal nor a maximal interpretation for this pattern. Thus, we choose to disallow this kind of patterns. We say that a type variable α is not *guarded* in a pattern π if it is reachable from the pattern by a sequence of epsilon transitions:

$$\pi \stackrel{\epsilon}{\longrightarrow} \ldots \stackrel{\epsilon}{\longrightarrow} \alpha \ .$$

Then, a pattern is *well-formed* if, for all its subpatterns $\mu(\alpha)\pi$, the type variable α is *guarded* in the pattern π. For instance, the pattern $\mu(\alpha)(x : (() \cup \alpha))$ is ill-formed as we have the transition sequence:

$$x : (() \cup \alpha) \stackrel{\epsilon}{\longrightarrow} () \cup \alpha \stackrel{\epsilon}{\longrightarrow} \alpha \ .$$

On the other hand, the patterns $\mu(\alpha)(c(\alpha, _))$ and $\mu(\alpha)(\alpha \rightarrow \bot)$ are both well-formed. Indeed, the occurrence of the type variable α is "guarded" by a pattern constructor in both cases. In the remainder of the paper, we only consider well-formed patterns.

Well-formedness is preserved by substitution. It is also closed under the transition relations:

REMARK 1 (Transition and Well-Formedness). *Well-formedness is preserved by transition: if π is well-formed and $\pi \longrightarrow^{c} (\pi_1,\pi_2)$, then both π_1 and π_2 are well-formed; likewise, if π is well-formed and $\pi \longrightarrow^{\epsilon} \pi'$, then π' is well-formed.*

Another significant point is that one can reason by induction on the length of sequences of epsilon transitions starting from a pattern π, thanks to the following remark.

REMARK 2 (Finite Epsilon Sequences). *There is no infinite sequence of epsilon transitions $\pi_1 \longrightarrow^{\epsilon} \pi_2 \longrightarrow^{\epsilon} \pi_3 \longrightarrow^{\epsilon} \ldots$*

Types are a simple restriction of patterns: a *type* τ is a (well-formed) pattern which contains no binder $x : \pi$ and no wildcard $_$. The reason for not allowing the wildcard in types is given in section 2.7.

2.4 Pattern Semantics

The matching relation $v \lhd \pi \rightsquigarrow \theta$, which specifies the semantics of patterns, is given in figure 6. As mentioned earlier, this relation asserts that the value v matches the pattern π producing a substitution θ. We write $v \lhd \pi$ when there exists a substitution θ such that $v \lhd \pi \rightsquigarrow \theta$.

We use a list-like notation for substitutions. The empty substitution $[]$, when applied to an expression e, leaves it unchanged. The one-variable substitution $[v/x]$ replaces all occurrences of the variable x by the value v. The concatenation $\theta_1 + \theta_2$ of two substitutions θ_1 and θ_2 performs simultaneously the two substitutions. When the substitutions operate on a common variable, the second

$$\frac{v_1 \lhd \pi_1 \rightsquigarrow \theta_1 \qquad v_2 \lhd \pi_2 \rightsquigarrow \theta_2}{c(v_1, v_2) \lhd c(\pi_1, \pi_2) \rightsquigarrow \theta_1 + \theta_2} \qquad () \lhd () \rightsquigarrow []$$

$$\frac{v \lhd \pi_1 \rightsquigarrow \theta}{v \lhd \pi_1 \cup \pi_2 \rightsquigarrow \theta} \qquad \frac{v \lhd \pi_2 \rightsquigarrow \theta}{v \lhd \pi_1 \cup \pi_2 \rightsquigarrow \theta}$$

$$\frac{v \lhd \pi[\mu(\alpha)\pi/\alpha] \rightsquigarrow \theta}{v \lhd \mu(\alpha)\pi \rightsquigarrow \theta} \qquad \frac{v \lhd \pi \rightsquigarrow \theta}{v \lhd x : \pi \rightsquigarrow \theta + [v/x]}$$

$$v \lhd _ \rightsquigarrow [] \qquad \frac{\emptyset \vdash v \ \# \ \pi_2 \rightarrow \pi_1}{v \lhd \pi_2 \rightarrow \pi_1 \rightsquigarrow []}$$

Figure 6. Matching Relation $v \lhd \pi \rightsquigarrow \theta$

substitution θ_2 takes precedence. For instance, applying the substitution $[v_1/x] + [v_2/x]$ to the expression $c(x, x)$ results in the expression $c(v_2, v_2)$, rather than $c(v_1, v_1)$ or even $c(v_2, v_1)$. This choice is actually not important, as the type system enforce the *linearity* of patterns, which ensures that, whenever two substitutions are concatenated during the evaluation of a well-typed term, they operate on distinct sets of type variables (see remark 4).

The first matching rules are straightforward. A variant value $c(v_1, v_2)$ matches a variant pattern $c(\pi_1, \pi_2)$ if the constructors are identical and the values v_1 and v_2 respectively match the patterns π_1 and π_2. The resulting substitution is the concatenation of the substitutions of each submatch. The unit value $()$ matches the unit pattern $()$, producing an empty substitution. A value matches a union pattern $\pi_1 \cup \pi_2$ if it matches either the pattern π_1 or the pattern π_2. A value matches a recursive pattern $\mu(\alpha)\pi$ if it matches its unrolling $\pi[\mu(\alpha)\pi/\alpha]$. A value v matches a binder pattern $x : \pi$ if it matches the pattern π. The resulting substitution is extended with the substitution of the binder variable x by the whole value v. All values match the wildcard pattern $_$.

Clearly, one cannot match a functional value against a function pattern $\pi_2 \rightarrow \pi_1$ by decomposing it into two values v_1 and v_2 and matching these values against the corresponding patterns π_1 and π_2. Thus, we resort to using the type of the value to perform the matching: a value v matches a function pattern $\pi_2 \rightarrow \pi_1$ if it has type $\pi_2 \rightarrow \pi_1$ in the empty environment (written $\emptyset \vdash v \ \# \ \pi_2 \rightarrow \pi_1$, see section 2.5). There is no rule for the empty pattern $\bot$ nor for pattern variables α. Indeed, no value should match the empty pattern. In the case of pattern variables, we make the conservative assumption that they match no value, as we do not know what they stand for. The typing rules actually ensure that when a matching is performed during the reduction, the pattern contains no free variable. For instance, we can have the following reduction for the identity applied to the unit value:

$$(\Lambda\alpha.\lambda(x : \alpha).x) \ [()] \ () \longrightarrow (\lambda(x : ()).x) \ () \longrightarrow ()$$

Though the pattern initially contained a type variable, this variable is substituted before the matching is performed. On the other hand, the expression $(\lambda(x : (() \cup \alpha)).x) \ ()$, even though it reduces to the unit value $()$, is ill-typed as the type variable α is not bound.

Note that binders are allowed in the subpatterns π_1 and π_2 of a function pattern $\pi_2 \rightarrow \pi_1$ but are ignored. It would certainly be cleaner to disallow them in a actual implementation. However, from a specification point of view, it is simpler to just ignore them.

There is a strong connection between the definition of the matching relation $v \lhd \pi \rightsquigarrow \theta$ and the transition relations $\pi \longrightarrow^{\epsilon} \pi$ and $\pi \longrightarrow^{c} (\pi,\pi)$. Indeed, for each rule with a non-empty set of premises, there is a transition relating the pattern in the conclusion and the patterns in the premises. For instance, the transition

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash c(e_1, e_2) : c(\tau_1, \tau_2)} \qquad \Gamma \vdash () : () \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma; \lfloor \pi \rfloor \vdash e : \tau}{\Gamma \vdash \lambda\pi.e : \lceil \pi \rceil \to \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

$$\frac{\Gamma; \alpha \vdash e : \sigma}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\sigma} \qquad \frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e\ [\tau] : \sigma[\tau/\alpha]}$$

$$\frac{\Gamma \vdash e_2 : \sigma \qquad \Gamma; x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \text{let } x = e_2 \text{ in } e_1 : \tau}$$

$$\frac{\Gamma \vdash e \# \bigcup_i \pi_i \qquad \text{For all } i, \quad \Gamma; \lfloor \pi_i \rfloor \vdash e_i : \tau}{\Gamma \vdash \text{match } e \text{ with } \overline{\pi_i \to e_i} : \tau}$$

$$\frac{\Gamma \vdash e : \sigma' \qquad \sigma' <: \sigma}{\Gamma \vdash e : \sigma} \qquad \frac{\Gamma \vdash e : \tau \qquad \tau <: \pi}{\Gamma \vdash e \# \pi}$$

Figure 7. Typing Judgments $\Gamma \vdash e : \sigma$ and $\Gamma \vdash e \# \pi$

$$\begin{array}{lcl}
\lceil c(\pi_1, \pi_2) \rceil & = & c(\lceil \pi_1 \rceil, \lceil \pi_2 \rceil) \\
\lceil () \rceil & = & () \\
\lceil \pi_1 \to \pi_2 \rceil & = & \lceil \pi_1 \rceil \to \lceil \pi_2 \rceil \\
\lceil \pi_1 \cup \pi_2 \rceil & = & \lceil \pi_1 \rceil \cup \lceil \pi_2 \rceil \\
\lceil \bot \rceil & = & \bot \\
\lceil \alpha \rceil & = & \alpha \\
\lceil \mu(\alpha)\pi \rceil & = & \mu(\alpha)\lceil \pi \rceil \\
\lceil x : \pi \rceil & = & \lceil \pi \rceil
\end{array}$$

Figure 8. Type of a Pattern $\lceil \pi \rceil$

$$\frac{\pi \xrightarrow{c} (\pi_1, \pi_2)}{\pi \prec \pi_1} \qquad \frac{\pi \xrightarrow{c} (\pi_1, \pi_2)}{\pi \prec \pi_2} \qquad \frac{\pi \xrightarrow{\epsilon} \pi'}{\pi \prec \pi'}$$

$$\pi \prec \pi \qquad \frac{\pi_1 \prec \pi_2 \qquad \pi_2 \prec \pi_3}{\pi_1 \prec \pi_3}$$

Figure 9. Reachability Relation $\pi \prec \pi$

$c(\pi_1, \pi_2) \longrightarrow^c (\pi_1, \pi_2)$ corresponds to the first rule and the transition $\pi_1 \cup \pi_2 \longrightarrow^\epsilon \pi_1$ corresponds to the third rule. Conversely, for each possible transition, there is a corresponding matching rule.

The semantics of patterns is not deterministic. This makes it simpler to formalize the calculus and reason about it. An actual implementation may choose a particular strategy, for instance, a first match policy. Clearly, our soundness results would apply, but our analyses may be cruder as they are not tuned to any particular strategy.

2.5 Typing Judgments

We define two typing judgments. The first one, $\Gamma \vdash e : \sigma$, is the usual typing judgment. It asserts that an expression e has type scheme σ in an environment Γ. The second one, $\Gamma \vdash e \# \pi$, is used specifically for typing patterns. It asserts that an expression e is *accepted* by the pattern π in an environment Γ. A *type scheme* is a type preceded by universal quantifiers:

$$\begin{array}{lcl}
\sigma & ::= & \tau \\
 & & \forall\alpha.\sigma
\end{array}$$

A *typing environment* is a sequence of bindings:

$$\begin{array}{llll}
\Gamma & ::= & \emptyset & \text{empty environment} \\
 & & \Gamma; x : \sigma & \text{variable binding} \\
 & & \Gamma; \alpha & \text{type variable binding}
\end{array}$$

We write $\Gamma; \Gamma'$ for the concatenation of two environments Γ and Γ'.

We adopt standard *well-formedness* conditions for patterns, type schemes, expressions and environments: a pattern π, a type scheme σ or an expression e are well-formed in a environment Γ if all their free variables are bound in the environment. An environment is well-formed if for all its variable bindings $\Gamma; x : \sigma$ the type scheme σ is well-formed in the environment Γ.

Typing judgments are defined in figure 7. For any occurrence of an assertion $\Gamma \vdash e : \sigma$ (resp. $\Gamma \vdash e \# \pi$) in these rules, we implicitly require that the environment Γ, the expression e and the type scheme σ (resp. the pattern π) are all well-formed. Most of the rules are straightforward. We comment the most interesting ones.

Given a pattern π, we write $\lceil \pi \rceil$ for the type representing what is accepted by the pattern and $\lfloor \pi \rfloor$ for the piece of environment corresponding to the variables bound in the pattern. These two operations are specified in section 2.6. In order to type an abstraction $\lambda\pi.e$, its body e must have some type τ in the environment extended with the bindings $\lfloor \pi \rfloor$ of the pattern π. Then, the type of the abstraction is a function type with argument type $\lceil \pi \rceil$ and return type τ. For pattern matching, the input expression e must be accepted by the union of the patterns π_i (this ensures that the pattern matching is *exhaustive*). Then, the body e_i of each branch is typed in an environment extended with the corresponding pattern bindings $\lfloor \pi_i \rfloor$. The type of the whole construction is a type τ common to all these bodies. In this typing rule, we use n-ary unions of patterns, which can be defined in term of the empty union $\bot$ and the binary union $\pi_1 \cup \pi_2$. Though there is no canonical definition, one can show that, thanks to subtyping, all possible definitions yield the same definition of the typing judgements (see lemma 8).

The judgement for patterns is introduced by the last rule: we rely on subtyping to define when an expression is accepted by a pattern. Note that the two typing judgments coincide on types: for any type τ, one have $\Gamma \vdash e : \tau$ if and only if $\Gamma \vdash e \# \tau$.

Though there is no explicit construction for recursion, a fixpoint operator can be typed using recursive types.

2.6 Type and Bindings of a Pattern

The type $\lceil \pi \rceil$ of a pattern π is defined in figure 8. This type is actually just the pattern where binders have been erased. Note that it is only defined when the pattern contains no wildcard $_$. In particular, there can be no wildcard in a function input pattern. For instance, the function $\lambda(x : _).()$ is ill-typed.

The definition of the bindings $\lfloor \pi \rfloor$ of a pattern π is more involved. We first define a reachability relation $\pi \prec \pi$ (figure 9). We say that the pattern π_2 is *below* the pattern π_1 when $\pi_1 \prec \pi_2$. The set of patterns below a given pattern π is the set of patterns reachable from the pattern π through the transitions. Hence, in the definition of the relation, there are three rules corresponding to the transitions. The last two rules ensure that the relation is closed under reflexivity and transitivity. As patterns can be recursive, the reachability relation is not in general an order, but only a preorder. It satisfies the following property.

REMARK 3 (Regularity). *There is only a finite number of patterns below a given pattern: for an pattern π, the set $\{\pi' \mid \pi \prec \pi'\}$ is finite.*

$$\frac{\pi[\mu(\alpha)\pi/\alpha] <: \pi'}{\mu(\alpha)\pi <: \pi'} \qquad \frac{\pi' <: \pi[\mu(\alpha)\pi/\alpha]}{\pi' <: \mu(\alpha)\pi}$$

$$\frac{\pi_1 <: \pi \qquad \pi_2 <: \pi}{\pi_1 \cup \pi_2 <: \pi} \qquad \frac{\pi <: \pi_1}{\pi <: \pi_1 \cup \pi_2} \qquad \frac{\pi <: \pi_2}{\pi <: \pi_1 \cup \pi_2}$$

$$\frac{\pi <: \pi'}{x : \pi <: \pi'} \qquad \frac{\pi' <: \pi}{\pi' <: x : \pi} \qquad \frac{\pi_1 <: \pi_1' \qquad \pi_2' <: \pi_2}{\pi_2 \to \pi_1 <: \pi_2' \to \pi_1'}$$

$$\bot <: \pi \qquad () <: () \qquad \alpha <: \alpha \qquad _ <: \pi \qquad \pi <: _$$

$$\frac{(\pi_1 \setminus \bot, \pi_2 \setminus \bot) <: \mathrm{expose}_c(\pi)}{c(\pi_1, \pi_2) <: \pi}$$

$$\frac{\begin{array}{c}(\pi_1 \setminus (\pi_1' \cup \pi_1''), \pi_2 \setminus \pi_2') <: \mathrm{L} \\ (\pi_1 \setminus \pi_1', \pi_2 \setminus (\pi_2' \cup \pi_2'')) <: \mathrm{L}\end{array}}{(\pi_1 \setminus \pi_1', \pi_2 \setminus \pi_2') <: [(\pi_1'', \pi_2'')] + \mathrm{L}}$$

$$\frac{\pi_1 <: \pi_1'}{(\pi_1 \setminus \pi_1', \pi_2 \setminus \pi_2') <: []} \qquad \frac{\pi_2 <: \pi_2'}{(\pi_1 \setminus \pi_1', \pi_2 \setminus \pi_2') <: []}$$

Figure 10. Subtyping Relation $\pi <: \pi$

The set of binders of a pattern π is obtained by collecting the variables x of all the binder patterns $x : \pi'$ below the pattern π.

$$\mathrm{binders}(\pi) = \{x \mid \exists \pi'.\pi \prec (x : \pi')\}$$

Note that binders inside a function pattern $\pi' \to \pi$ are ignored, which is consistent with the semantics of patterns.

We then define a notion of linearity for patterns: a pattern π is *linear* when:

- if $\pi \prec c(\pi_1, \pi_2)$, then $\mathrm{binders}(\pi_1) \cap \mathrm{binders}(\pi_2) = \emptyset$;
- if $\pi \prec \pi_1 \cup \pi_2$, then $\mathrm{binders}(\pi_1) = \mathrm{binders}(\pi_2)$;
- if $\pi \prec x : \pi_1$, then $x \notin \mathrm{binders}(\pi_1)$.

REMARK 4. *Linearity ensures that a variable is bound at most once when an input value is matched again a pattern, and that the set of bound variables is the same for all input values.*

Finally, given a linear pattern π, we define its bindings $\lfloor \pi \rfloor$ as follows (we leave $\lfloor \pi \rfloor$ undefined when the pattern π is not linear). Suppose that $\mathrm{binders}(\pi) = \{x_1, \ldots, x_n\}$ (where $x_i \neq x_j$ for $i \neq j$). Then,

$$\lfloor \pi \rfloor = x_1 : \tau_1; \ldots; x_n : \tau_n$$

where

$$\tau_i = \bigcup_{\pi \prec (x_i : \pi')} \lceil \pi' \rceil \; .$$

Note that $\lfloor \pi \rfloor$ is not actually defined for all linear patterns, as the type $\lceil \pi' \rceil$ is only defined for patterns π' containing no wildcard $_$. For instance, $\lfloor x : _ \rfloor$ is not defined as the type associated to the variable x would be $\lceil _ \rceil$, which is undefined.

The definition above uses unions of a finite sets of types, As noted above, this union is not uniquely defined. Besides, though environments are *a priori* ordered, the order of bindings in $\lfloor \pi \rfloor$ is left unspecified. This is not an issue as it can be shown that the definition of the judgments do not depend on the specific definition of the union chosen nor on the ordering of the bindings.

2.7 Subtyping

The *subtyping relation* $\pi <: \pi$ is defined in figure 10 using *coinductive rules* [1, 5]. A recursive pattern $\mu(\alpha)\pi$ is in subtyping

$$\begin{array}{lcll}
\mathrm{expose}_c(\pi_1 \cup \pi_2) & = & \mathrm{expose}_c(\pi_1) + \mathrm{expose}_c(\pi_2) & \\
\mathrm{expose}_c(x : \pi) & = & \mathrm{expose}_c(\pi) & \\
\mathrm{expose}_c(\mu(\alpha)\pi) & = & \mathrm{expose}_c(\pi[\mu(\alpha)\pi/\alpha]) & \\
\mathrm{expose}_c(c(\pi_1, \pi_2)) & = & [(\pi_1, \pi_2)] & \\
\mathrm{expose}_c(\pi) & = & [] & \text{otherwise}
\end{array}$$

Figure 11. Exposure Function $\mathrm{expose}_c(\pi)$

relation with another pattern π if its unrolling $\pi[\mu(\alpha)\pi/\alpha]$ is. The three rules for union assert that the union of two patterns is a least upper bound of these two patterns. Binders are ignored. The rule for function patterns is the usual one: covariance on the result types and contravariance on the input types. The empty pattern is a least pattern. There is no specific interaction between the union pattern $\bot$ or a type variable α with other patterns: the rules for the unit pattern and the type variables simply ensure the reflexivity of subtyping.

We consider a naïve, syntactic, interpretation of the wildcard pattern: the pattern $_ \to _$ matches all functions as, for any function, we can find two suitable types to fill in the holes. This can be understood more formally by interpreting the wildcard pattern as a top pattern when in covariant position and as a bottom pattern when in contravariant position. This turns out to make sense from a semantic theory point of view: the wildcard pattern can be interpreted as an *interval type*, as defined by Cartwright [2]. This is also very simple to specify in our calculus: basically, this is handled by the two subtyping rules for the wildcard pattern. These rules may look somewhat surprising as the pattern seems to be considered both as a greatest pattern and as a least pattern, which would be contradictory. This makes sense here because in the typing rules we are only interested in subtyping assertions of the shape $\tau <: \pi$. In the derivation of such an assertion $\tau_0 <: \pi_0$, the rule $\pi <: _$ may occur only when the pattern $_$ is in covariant position in the pattern π_0, and the rule $_ <: \pi$ may occur only when the pattern $_$ is in contravariant position. Other design choices are possible: a separate top type could be added, or the wildcard pattern could be interpreted as a top pattern and allowed in types. But we think it is interesting to present this unusual alternative interpretation.

The four rules for variant patterns $c(\pi, \pi)$ are based on a reformulation of the XDuce subtyping rules [8] by Hosoya [Personal communication]. These rules are very precise and handle distributivity of variants over pattern union. For instance, the pattern $c(\pi_1, \pi_2 \cup \pi_3)$ can be proved equivalent to the pattern $c(\pi_1, \pi_2) \cup c(\pi_1, \pi_3)$. They are based on a simultaneously defined relation $(\pi \setminus \pi, \pi \setminus \pi) <: \mathrm{L}$ (where the symbol L denotes a list of pairs of patterns) and an exposure function $\mathrm{expose}_c(\pi)$ specified in figure 11. We adopt the following notations for lists. We write $[]$ for the empty list, $[a]$ for a list with a single element a and $\mathrm{L} + \mathrm{L}'$ for the concatenation of two lists L and L'. The exposure function $\mathrm{expose}_c(\pi)$ extracts from a pattern π the arguments of all variant patterns with constructor c. It is defined in a recursive manner. Using remark 2, one can show that it is indeed well-defined.

Intuitively, the goal of the four rules for variant patterns is, given a pattern $c(\pi_1, \pi_2)$ on the left hand side of a subtyping assertion, to derive some conditions on patterns π_1 and π_2. The easiest way to understand these rules is to interpret them in a set-theoretic way, with the following correspondence in mind:

subtyping relation $<:$	set inclusion $\subseteq$
empty pattern $\bot$	empty set $\emptyset$
symbol $\setminus$	set difference $\setminus$
union pattern $\cup$	set union $\cup$
empty list $[]$	empty set $\emptyset$
list concatenation $+$	set union $\cup$

$$\pi \approx \pi \qquad \frac{\pi_1 \approx \pi_2 \qquad \pi_2 \approx \pi_3}{\pi_1 \approx \pi_3} \qquad \frac{\pi_1 \approx \pi_1' \qquad \pi_2 \approx \pi_2'}{\pi_1 \cup \pi_2 \approx \pi_1' \cup \pi_2'}$$

$$\pi_1 \cup \pi_2 \approx \pi_2 \cup \pi_1 \qquad \pi_1 \cup (\pi_2 \cup \pi_3) \approx (\pi_1 \cup \pi_2) \cup \pi_3$$

$$\pi \cup \pi \approx \pi \qquad \pi \cup \bot \approx \pi$$

Figure 12. Equivalence between Patterns $\pi \approx \pi$

The first rule rewrites the left hand side in a trivial way so that it matches the shape required for the latter rules and extracts interesting subpatterns from the right hand side pattern. The second rule take advantage of the following set theoretic property in order to reduce the size of the left hand side list:

$$A \times B \subseteq (C \times D) \cup E$$
if and only if
$$(A \setminus C) \times B \subseteq E \quad \text{and} \quad A \times (B \setminus D) \subseteq E.$$

Finally, the last two rules rely on the fact that $A \times B \subseteq \emptyset$ if and only if any of the sets A and B is empty to eliminate the Cartesian product.

The guardedness condition is crucial for the soundness of subtyping. Indeed, we have $\alpha[\mu(\alpha)\alpha/\alpha] = \mu(\alpha)\alpha$. Hence, without the restriction, we could derive $\tau <: \mu(\alpha)\alpha$ and $\mu(\alpha)\alpha <: \tau'$ for any types τ and τ' with the two infinite derivations:

$$\frac{\frac{\vdots}{\tau <: \mu(\alpha)\alpha}}{\tau <: \mu(\alpha)\alpha} \qquad \frac{\frac{\vdots}{\mu(\alpha)\alpha <: \tau'}}{\mu(\alpha)\alpha <: \tau'}$$

Then, by applying the subtyping rule twice, if we have $\Gamma \vdash e : \tau$, we could derive $\Gamma \vdash e : \tau'$ for any type τ'.

The three following lemmas are crucial to prove different properties of the calculus.

LEMMA 5 (Reflexivity). *For all patterns π, one have $\pi <: \pi$.*

LEMMA 6 (Transitivity). *For all patterns π_1 and π_3 and for all types τ_2, if $\pi_1 <: \tau_2$ and $\tau_2 <: \pi_3$, then $\pi_1 <: \pi_3$.*

Note that this would not hold if we allowed τ_2 to be a pattern. For instance, for any patterns π_1 and π_2, we have $\pi_1 <: _$ and $_ <: \pi_2$ but fortunately not in general $\pi_1 <: \pi_2$.

LEMMA 7 (Stability by substitution). *For all patterns π_1 and π_2, for all types τ and for all type variables α, if $\pi_1 <: \pi_2$ then $\pi_1[\tau/\alpha] <: \pi_2[\tau/\alpha]$.*

It is convenient to define an equivalence relation $\pi \approx \pi$ between patterns (figure 12). With this relation, union is considered commutative, associative and idempotent, and the empty pattern $\bot$ is an identity for union. A justification to this relation if given by the following lemma.

LEMMA 8 (Subtyping and Union). *The subtyping relation is compatible with the equivalence relation: if $\tau_1 <: \tau_2$, $\tau_1 \approx \tau_1'$ and $\tau_2 \approx \tau_2'$, then $\tau_1' <: \tau_2'$.*

As a consequence of this lemma, all definitions of finite union using binary union and empty union are equivalent. We relied on this fact to show that the typing judgments are well-defined.

The subtyping rules do not provide directly an algorithm. Indeed, the set of patterns that occur in a derivation may be infinite. This is the case for instance for a derivation of the assertion $\tau_1 <: \tau_2$ where:

$$\begin{array}{rcl} \tau_1 & = & \mu(\alpha)c(\alpha, \alpha) \\ \tau_2 & = & \mu(\alpha)c(\alpha \cup \alpha, \alpha \cup \alpha) \ . \end{array}$$

By applying the rules for variants, one arrive higher up in the derivation to the assertion $\tau_1 <: (\bot \cup \tau_2) \cup \tau_2$, then to an assertion with four occurrences of τ_2, and so on... However, one can show that all patterns occurring in a derivation of an assertion $\pi_1 <: \pi_2$ are unions of subpatterns of the patterns π_1 and π_2 up to the equivalence relation $\approx$. Hence, thanks to lemma 8, the rules that may occur in a derivation of an assertions $\pi_1 <: \pi_2$ can be considered as a finite, though possibly large, set of Boolean formulas whose variables are assertions $\pi_1' <: \pi_2'$ up to the equivalence relation. The subtyping problem then becomes a Boolean constraint solving problem. This way of deciding coinductively defined relations is detailed in Frisch's PhD thesis [4].

The subtyping relation is extended to type schemes with the following additional rule:

$$\frac{\sigma <: \sigma'}{\forall\alpha.\sigma <: \forall\alpha.\sigma'}$$

2.8 Properties of the Calculus

Given an environment Γ and an expression e, we say that a type scheme σ_0 is *principal* when it is a least type scheme such that $\Gamma \vdash e : \sigma_0$. We call *principal judgment* the set of assertions $\Gamma \vdash e : \sigma_0$ where the type σ_0 is principal.

REMARK 9 (Principal Types). *If $\Gamma \vdash e : \sigma$ or $\Gamma \vdash e \mathbin{\#} \pi$, then there exists a least type scheme σ_0 such that $\Gamma \vdash e : \sigma_0$. In the second case, this type scheme is actually a type τ_0 and $\tau_0 <: \pi$.*

A principal type can be computed by applying the typing rules in a syntax-directed way, using the subtyping rule only when necessary (that is, above the typing rule for application $e_1\ e_2$).

The soundness proof rely on the following three lemmas concerning pattern matching. We say that a pattern is *ground* when it contains no free type variable. We say that a value v is *well-typed* (implicitly, in the empty environment) when there exists a type τ such that $\emptyset \vdash v : \tau$. The first lemma show that the semantics of patterns corresponds tightly to their types.

LEMMA 10 (Equivalence Matching-Typing). *For all well-typed values v and ground patterns π, $v \triangleleft \pi$ if and only if $\emptyset \vdash v \mathbin{\#} \pi$.*

LEMMA 11 (Matching and Subtyping). *For all values v, ground types τ and ground patterns π, if $v \triangleleft \tau$ and $\tau <: \pi$, then $v \triangleleft \pi$.*

LEMMA 12 (Pattern Bindings). *For all ground patterns π, expressions e, types τ, well-typed values v and substitution θ, if $\lfloor \pi \rfloor \vdash e : \tau$ and $v \triangleleft \pi \leadsto \theta$, then $\emptyset \vdash \theta(e) : \tau$*

The type soundness is proved in the usual way by showing type preservation and progress [12].

THEOREM 13 (Type Preservation). *For any expressions e and e' and any type scheme σ, if $\emptyset \vdash e : \sigma$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : \sigma$.*

THEOREM 14 (Progress). *For any expression e and any type scheme σ, if $\emptyset \vdash e : \sigma$, then either the expression e is a value or there exists and expression e' such that $e \longrightarrow e'$.*

3. Semantic Interpretation of Subtyping

The matching relation $v \triangleleft \pi \leadsto \theta$ provides a semantics for ground patterns:

$$[\![\pi]\!] = \{v \mid v \triangleleft \pi\} \ .$$

This semantics can easily be extended to all patterns using substitutions. We define a *ground substitution* s as a function from type variables α to ground types. Then, the semantics of a pattern π with respect to a ground substitution s is defined as $[\![s(\pi)]\!]$.

This induces an inclusion relation between patterns: we say that a pattern π is *semantically a subtype* of another pattern π', written $\pi \subseteq \pi'$, when for any ground substitution s we have

$$[\![s(\pi)]\!] \subseteq [\![s(\pi')]\!] \ .$$

We can compare this relation with the subtyping relation defined in section 2.7. In fact, this makes sense only when the left hand side pattern is a type. Indeed, remember that a pattern is not interpreted the same way on both sides of the subtyping relation, due to the treatment of the wildcard pattern. The semantics $[\![\pi]\!]$ corresponds to the interpretation on the right hand side. It can only be used on the left hand side when the two interpretation coincides, which is the case for types τ. The asymmetry is not an issue as patterns never occur on the left hand side of a subtyping assertion in the typing rules. It turns out that both relations coincide.

THEOREM 15. *For all types τ and all patterns π, we have $\tau <: \pi$ if and only if $\tau \subseteq \pi$.*

One direction, corresponding to the soundness of the subtyping relation $\tau <: \pi$, is a direct consequence of lemmas 7 and 11. The other direction, completeness, is more interesting.

It is clear that completeness holds for types and patterns without function patterns nor type variables: this is the completeness of the XDuce subtyping rules [8]. That it still holds with function patterns should not be that surprising, as the semantics of these patterns is defined using subtyping. In particular, due to this circularity, the completeness result does not imply that our subtyping relation is the finest subtyping relation possible. For instance, we believe that the following assertion is sound:

$$\pi_1 \to \pi_2 <: \bot \to \bot \ .$$

Indeed, the only reduction rule for which the type of a function really matters is function application. Thus, as a function of type $\bot \to \bot$ cannot be applied to any value, any function can have this type. From a theoretic point of view, it is attractive to use the finest possible subtyping relation: as the type system is monotonic with respect to the subtyping relation, more programs could be typed (though the pattern semantics would also be changed, as it depends on typing). From a practical point of view, however, it seems that this relation would not have good algorithmic properties—it seems hard to decide—and would not allow to type more interesting programs.

Completeness when including type variables is more interesting. In order to obtain a similar result, Hosoya, Frisch and Castagna [6] have to define a very specific semantics, where values are labelled with variables corresponding to the type variables of a pattern. We do not experience the same difficulties as our type system is slightly different(They have a type construction $\alpha : \tau$ with the following semantics: the result of instantiating in this type the variable α to a type τ' is the intersection of types τ and τ', which can be expressed in their type system thanks to this very construction.)

4. Untyped Semantics

4.1 Motivations

We now would like to consider a subset of the above typed calculus, with two goals in mind. First, it should be possible to implement this subset in a efficient way without too much effort, in particular by relying on previous works on optimizing regular patterns [9]. Second, we would like type variables to be opaque. It should not be possible to inspect by pattern matching (or through any other construction of the calculus) any part of a value corresponding to a type variable. Indeed, such a guarantee makes it easy to extend in a seamless way a language with additional datatypes, such as integers, arrays or file descriptors, without the need of any boxing to distinguish the values of these additional types from other values.

The main challenge for an efficient implementation is that some type operations must be performed at runtime. Indeed, type variable substitutions have an effect on the semantics of patterns. This makes it difficult to statically compile patterns containing type variables. Furthermore, a type check is required at runtime in order to match a value against a function pattern:

$$\frac{\emptyset \vdash v \ \# \ \pi_2 \to \pi_1}{v \triangleleft \pi_2 \to \pi_1 \rightsquigarrow []}$$

Thanks to the principal type property (remark 9), it is sufficient to compute the principal type τ of the value v and check whether it is a subtype of the pattern $\pi_2 \to \pi_1$. Actually, if we followed the semantics precisely, this principal type would have to be computed at runtime, as principal types are not preserved by reduction. For instance, if we start from the principal judgement:

$$x : \alpha \cup () \vdash \lambda().x : () \to (\alpha \cup ())$$

and substitute the value () for the variable x, we get the judgment:

$$\Gamma \vdash \lambda().() : () \to (\alpha \cup ())$$

instead of the principal judgement:

$$\vdash \lambda().() : () \to () \ .$$

This would be clearly unpractical, but this minor difficulty can be solved by requiring the full type of a function to be systematically given (or added during an elaboration phase after type checking). That is, the grammar of functions would become $\lambda\pi.(e : \tau)$, with the typing rule:

$$\frac{\Gamma; \lfloor\pi\rfloor \vdash e : \tau' \qquad \tau' <: \tau}{\Gamma \vdash \lambda\pi.(e : \tau) : \lceil\pi\rceil \to \tau}$$

This is the approach taken in CDuce [3]. The principal type of a function can then be directly read from its definitionand matching against a function pattern is just a matter of performing a subtyping check.

In order to demonstrate how runtime type operations can be eliminated, we propose a compilation of the typed calculus into an untyped subset, in which no type operation need to be performed. This translation is partial: some patterns are rejected. We rapidly sketch the translation here. The first optimization is to replace type variables in patterns by the wildcard pattern and to remove all type abstractions and type applications. For instance, the expression:

$$(\Lambda\alpha.\lambda(x : \alpha).x) \ [\tau]$$

would be turned into:

$$\lambda(x : _).x \ .$$

This is potentially a lot more efficient as no type variable substitution is performed anymore. Furthermore, if this function is applied to a value v during reduction, the value can be immediately accepted by the function instead of being first matched against the type τ. In this case, the rewriting is faithful: the two functions behave the same way. On the other hand, the following expression has to be rejected.

$$\mathtt{match}\ ()\ \mathtt{with}\ \alpha \to e_1 \mid () \to e_2$$

Indeed, the natural translation would be:

$$\mathtt{match}\ ()\ \mathtt{with}\ _ \to e_1 \mid () \to e_2 \ .$$

(Of course, the expressions e_1 and e_2 should also be translated. For simplicity, we assume that they coincide with their translations.) But initially the first branch is never taken when α is instantiated to $\bot$, while it may be taken after translation. The second optimization is to also replace function patterns by the wildcard pattern. For

instance, the expression:

$$\lambda(x:\tau' \to \tau).e$$

would be translated into the expression:

$$\lambda(x:_).e \ .$$

Again, the translation is not always faithful and a similar counterexample can be found. The restrictions on patterns that make these two optimizations possible can be phrased informally as follows: it should be possible to consider type variables and function patterns as assertions which are checked at compile type but have no computational effect. A more drastic restriction would be not to allow them at all but we believe they can be useful. In particular, it would be inconvenient to have to provide in the definition of a polymorphic function both the type of the parameter and a pattern to deconstruct the parameter.

We choose to also reject some patterns, not for optimization purpose, but so that the calculus satisfies some parametricity properties [10]. For instance, suppose that an implementation of complex numbers is defined by some type τ and two functions *add* and *mul* of type $\tau \to \tau \to \tau$. Then, one may expect the behavior of a function such as the one below not to depend on the particular implementation of complex numbers it is applied to.

$$\Lambda\alpha.\lambda(\mathit{add}:\alpha\to\alpha\to\alpha).\lambda(\mathit{mul}:\alpha\to\alpha\to\alpha).e$$

For instance, it should have the same behavior whether a Cartesian or a polar representation is used. But this is not the case in our calculus as patterns can inspect values of any type. For instance, the following function behaves differently whether it is passed the unit, a variant or another value.

$$\Lambda\alpha.\lambda(x:\alpha).\text{match } x \text{ with } () \to e_1 \mid c(_,_) \to e_2 \mid _ \to e_3$$

We actually seek a stronger result that parametricity. In particular, we want to reject the function below, even though its behavior does not depends on its argument (it always returns the value $()$).

$$\Lambda\alpha.\lambda(x:\alpha).\text{match } x \text{ with } c((),()) \to () \mid _ \to ()$$

Indeed, this function “breaks abstraction”, in the sense that it tests whether a value of an abstract type α has some structure $c((),())$. If this function is allowed, then such a test must be possible for all values, which put some constraints on how we may extend the calculus with additional values.

The restrictions are fairly drastic. In particular, they imply that patterns cannot be specialized by type instantiation. But the type system remains rich. There is no restriction on monomorphic code and it is still possible to encode in a straightforward fashion ML algebraic datatypes and patterns.

We will now present the translation of the calculus into the untyped subset and two analyses on patterns. The first one ensures that the calculus satisfies parametricity properties. The second one ensures that the translation is faithful. These two analyses are presented together as applying them conjointly during the translation makes some simplifications possible.

4.2 Translation

The translation of expressions is specified in figure 13 through the relation $\Gamma \vdash e : \sigma \rightsquigarrow e$. An assertion $\Gamma \vdash e : \sigma \rightsquigarrow e'$ states that the expression e has type σ in the environment Γ and translates to the expression e'. The rules are directly derived from the typing rules of figure 7 by extending the typing judgments to include translated expressions and adding some side-conditions. In most cases, the translation is achieved by recursively replacing subexpressions by their translations.

Type abstraction is restricted to values: the expression $\Lambda\alpha.e$ can only be translated when the expression e is a value. This way, type

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash c(e_1,e_2) : c(\tau_1,\tau_2) \rightsquigarrow c(e_1',e_2')} \qquad \Gamma \vdash () : () \rightsquigarrow ()$$

$$\frac{x:\sigma \in \Gamma}{\Gamma \vdash x : \sigma \rightsquigarrow x} \qquad \frac{\Gamma; \lfloor\pi\rfloor \vdash e : \tau \rightsquigarrow e' \qquad \mathrm{erase}(\lceil\pi\rceil,\pi,\pi')}{\Gamma \vdash \lambda\pi.e : \lceil\pi\rceil \to \tau \rightsquigarrow \lambda\pi'.e'}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : \tau_1 \rightsquigarrow e_1'\ e_2'}$$

$$\frac{\Gamma;\alpha \vdash v : \sigma \rightsquigarrow e}{\Gamma \vdash \Lambda\alpha.v : \forall\alpha.\sigma \rightsquigarrow e} \qquad \frac{\Gamma \vdash e : \forall\alpha.\sigma \rightsquigarrow e'}{\Gamma \vdash e\ [\tau] : \sigma[\tau/\alpha] \rightsquigarrow e'}$$

$$\frac{\Gamma \vdash e_2 : \sigma \rightsquigarrow e_2' \qquad \Gamma; x:\sigma \vdash e_1 : \tau \rightsquigarrow e_1'}{\Gamma \vdash \text{let } x = e_2 \text{ in } e_1 : \tau \rightsquigarrow \text{let } x = e_2' \text{ in } e_1'}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \rightsquigarrow e' \qquad \tau <: \bigcup_i \pi_i \\ \text{For all } i,\quad \Gamma; \lfloor\pi_i\rfloor \vdash e_i : \tau' \rightsquigarrow e_i' \quad \text{and} \quad \mathrm{erase}(\tau,\pi_i,\pi_i')\end{array}}{\Gamma \vdash \text{match } e \text{ with } \overline{\pi_i \to e_i} : \tau' \rightsquigarrow \text{match } e' \text{ with } \overline{\pi_i' \to e_i'}}$$

$$\frac{\Gamma \vdash e : \sigma' \rightsquigarrow e' \qquad \sigma' <: \sigma}{\Gamma \vdash e : \sigma \rightsquigarrow e'}$$

Figure 13. Term Translation $\Gamma \vdash e : \sigma \rightsquigarrow e$

abstractions and applications can be completely eliminated without modifying significantly the semantics: the translation of a type abstraction remains a value. Such a restriction is widely used, in particular in ML implementations, in order to ensure soundness in presence of side-effects [11]. It does not impact the expressiveness of the calculus in any significant way.

In the case of the abstraction and the pattern matching constructions, a side-condition of the form $\mathrm{erase}(\tau,\pi,\pi')$ is added. This side-condition relates the initial pattern π, the type τ of what is matched against this pattern, and a translation π' of this pattern. The assertion $\mathrm{erase}(\tau,\pi,\pi')$ holds when:

- the pattern π' is *erased*, that is, it contains no function pattern $\pi_2 \to \pi_1$ nor free type variable α;
- the pattern π' is a translation of pattern π, that is, $\pi \rightsquigarrow^{\emptyset} \pi'$ (defined just below);
- the pattern does not break abstraction, that is, the relation $\mathrm{abstract}(\tau,\pi')$ defined in section 4.3 holds;
- the translation is faithful, that is, the relation $\mathrm{faithful}(\tau,\pi,\pi')$ defined in section 4.4 holds.

One may have expected a simpler translation relation $e \rightsquigarrow e$ for expressions. The reason for a typeful translation is that the input type τ of patterns is crucial for the analyses we perform on patterns.

The pattern translation $\pi \rightsquigarrow^{V} \pi$ (where V is a set of type variables α) is specified in figure 14. This is a rather broad relation: translation is simply defined as replacing in the initial pattern some unspecified subparts containing no binder by the wildcard pattern. Indeed, in order to prove more easily the accuracy of the translation of the whole calculus, we need a translation relation which is stable by substitution. The set V keeps track of the recursion variables. It is necessary to deal with patterns such as $\mu(\alpha)(c(\alpha,()) \cup x:())$. It would be incorrect to translate this pattern into $\mu(\alpha)(_ \cup x:())$ even though the subpattern $c(\alpha,())$ does not contain explicitly any binder.

$$\frac{\text{binders}(\pi) = \emptyset \quad \forall \alpha, (\pi \prec \alpha) \Rightarrow \alpha \notin V}{\pi \overset{V}{\rightsquigarrow} _} \qquad \pi \overset{V}{\rightsquigarrow} \pi \qquad \frac{\pi_1 \overset{V}{\rightsquigarrow} \pi_1' \quad \pi_2 \overset{V}{\rightsquigarrow} \pi_2'}{c(\pi_1, \pi_2) \overset{V}{\rightsquigarrow} c(\pi_1', \pi_2')}$$

$$\frac{\pi_1 \overset{V}{\rightsquigarrow} \pi_1' \quad \pi_2 \overset{V}{\rightsquigarrow} \pi_2'}{\pi_1 \cup \pi_2 \overset{V}{\rightsquigarrow} \pi_1' \cup \pi_2'} \qquad \frac{\pi \overset{V \cup \{\alpha\}}{\rightsquigarrow} \pi'}{\mu(\alpha)\pi \overset{V}{\rightsquigarrow} \mu(\alpha)\pi'} \qquad \frac{\pi \overset{V}{\rightsquigarrow} \pi'}{x : \pi \overset{V}{\rightsquigarrow} x : \pi'}$$

Figure 14. Pattern Translation $\pi \rightsquigarrow^V \pi$

In an actual implementation, one would use a different definition of pattern translation: all type variables α, provided they are not recursion variables, and all function patterns $\pi \to \pi$ would be replaced by the wildcard pattern; other subpatterns would be preserved. This defines a deterministic subset of the relation $\pi \rightsquigarrow^V \pi$. This way, each expression has at most one translation. In order to know whether the translation e' of an expression e exists, one can rely on the principal type property: one starts from a principal derivation of $\Gamma \vdash e : \sigma$ and tries to extend it into a derivation of the assertion $\Gamma \vdash e : \sigma \rightsquigarrow e'$. Using principal types ensures that the analyses are as precise as possible and thus that the translation e' is rejected only when really necessary.

4.3 Enforcing Abstraction

We first present some examples of patterns that should be rejected. Then, we clarify our intuition of the pattern matching process. This leads us to a specification of the analysis that should be performed, from which an algorithm is derived.

It is actually hard to specify when a pattern breaks abstraction. Indeed, this depends on the pattern matching implementation. For instance, consider the following function.

$$\lambda(x : c(\alpha, c((), ()))).\text{match } x \text{ with } c((), ()) \to e_1 \mid _ \to e_2$$

A naïve pattern matching algorithm may first test whether the first component of the variant, which is polymorphic, is the unit, before testing the second component. In this case, a better strategy would be to first test the second component or, even better, to detect at compile time that the first branch is never taken and optimize it away. However, there is sometimes no alternative strategy. Indeed, consider the function:

$$\lambda(x : \tau_2).\text{match } x \text{ with } \tau_1 \to e_1 \mid _ \to e_2$$

with the two types:

$$\begin{array}{rcl} \tau_1 & = & c(c((), ()), c((), ())) \\ \tau_2 & = & c(\alpha, ()) \cup c((), \alpha) \cup \tau_1 \ . \end{array}$$

There is no good order to look inside the variant: there may be a polymorphic value on either sides. Furthermore, the first branch cannot be optimized away as it matches input values of type τ_1. For the sake of simplicity, we consider that both functions should be rejected.

The analysis is performed on the pattern π' of the relation $\text{erase}(\tau, \pi, \pi')$. Indeed, this pattern is simpler that the initial pattern π. In particular, it is erased. The input type τ is used as an indication of which values may be matched against the pattern.

As far as the analysis is concerned, we consider function types the same way as type variables. Thus, the following expression is not allowed, as a function is matched against a unit pattern.

$$\text{match } \lambda().() \text{ with } () \to e_1 \mid _ \to e_2$$

Indeed, from a practical point of view, this puts less constraints on the implementation: there is no need to be able to distinguish functions from other values at runtime. From a formalization point of view, this allow us to use functions as witnesses in the specification of the analysis: a pattern is rejected if it may try to match a function value against either the unit pattern $()$ or a variant pattern $c(\pi, \pi)$ (these are the two kind of subpatterns of an erased pattern that require inspection of the value).

As a running example for the remainder of the section, we consider the function:

$$\Lambda\alpha.\lambda(x : \tau_0).\text{match } x \text{ with } \pi_0 \to e_1 \mid _ \to e_2$$

where the input type τ_0 and the pattern π_0 are defined as follows:

$$\begin{array}{rcl} \tau_0 & = & c((), \alpha) \cup () \\ \pi_0 & = & \mu(\beta)c(\beta, ()) \cup () \ . \end{array}$$

Intuitively, we consider the matching process as starting from the assertion $v \triangleleft \pi_0 \rightsquigarrow \theta$ (where the value v is to be matched against the pattern π_0 and the substitution θ is to be determined) and applying any possible matching rules, thus producing more and more refined *partial derivations*. The process is finished when a full derivation is reached. For instance, let us assume that the type variable α has been instantiated to the type $() \to ()$ and that the value is $v = c((), \lambda().())$. Then, the only rule we can apply initially is the rule for recursive patterns. We then have the following partial derivation.

$$\frac{\begin{array}{c}\vdots \\ v \triangleleft c(\pi_0, ()) \cup () \rightsquigarrow \theta\end{array}}{v \triangleleft \pi_0 \rightsquigarrow \theta}$$

It can be refined by applying one of the two rules for union patterns. With the second rule, the partial derivation below is reached.

$$\cfrac{\cfrac{\begin{array}{c}\vdots \\ v \triangleleft () \rightsquigarrow \theta()\end{array}}{v \triangleleft c(\pi_0, ()) \cup () \rightsquigarrow \theta}}{v \triangleleft \pi_0 \rightsquigarrow \theta}$$

There is no way to complete this partial derivation. On the other hand, with the other rule, we can apply one further rule and reach the partial derivation below.

$$\cfrac{\cfrac{\cfrac{\begin{array}{c}\vdots \\ () \triangleleft \pi_0 \rightsquigarrow \theta_2\end{array} \quad \begin{array}{c}\vdots \\ \lambda().() \triangleleft () \rightsquigarrow \theta_1\end{array}}{v \triangleleft c(\pi_0, ()) \rightsquigarrow \theta_1 + \theta_2}}{v \triangleleft c(\pi_0, ()) \cup () \rightsquigarrow \theta_1 + \theta_2}}{v \triangleleft \pi_0 \rightsquigarrow \theta_1 + \theta_2}$$

Considering the pattern in the rightmost branch, one may be tempted to apply the rule for the unit pattern. One has to look at the corresponding value, which is a function, in order to know that the rule cannot be applied. Hence, the pattern should be rejected.

As noted near the end of section 2.4, there is a strong correspondence between the subtyping rules and the transition relations. Thus, a partial derivation can be abstracted away as a subgraph of the graph of these relations. Corresponding to the partial derivation above, we have the following graph. Under each pattern, we write the value it is associated to in the partial derivation.

$$\begin{array}{ccccccc} \pi_0 & \xrightarrow{\epsilon} & c(\pi_0, ()) \cup () & \xrightarrow{\epsilon} & c(\pi_0, ()) & \xrightarrow{c} & () \\ v & & v & & v & \downarrow & \lambda().() \\ & & & & & \pi_0 & \\ & & & & & () & \end{array}$$

We can see on the rightmost node that the pattern should be rejected. This information should be propagated to the root pattern π_0

$$\lambda\pi.e \lightning c(\pi_1,\pi_2) \qquad \lambda\pi.e \lightning () \qquad \frac{v \lightning \pi' \quad \pi \xrightarrow{\epsilon} \pi'}{v \lightning \pi}$$

$$\frac{v_1 \lightning \pi_1 \quad \pi \xrightarrow{c} (\pi_1,\pi_2)}{c(v_1,v_2) \lightning \pi} \qquad \frac{v_2 \lightning \pi_2 \quad \pi \xrightarrow{c} (\pi_1,\pi_2)}{c(v_1,v_2) \lightning \pi}$$

Figure 15. Disallowed Matching $v \lightning \pi$

$$\frac{\tau \xrightarrow{\epsilon} \tau'}{(\tau,\pi) \prec (\tau',\pi)} \qquad \frac{\pi \xrightarrow{\epsilon} \pi'}{(\tau,\pi) \prec (\tau,\pi')}$$

$$\frac{\tau \xrightarrow{c} (\tau',\tau'') \quad \pi \xrightarrow{c} (\pi',\pi'') \quad \triangleleft\tau''}{(\tau,\pi) \prec (\tau',\pi')}$$

$$\frac{\tau \xrightarrow{c} (\tau',\tau'') \quad \pi \xrightarrow{c} (\pi',\pi'') \quad \triangleleft\tau'}{(\tau,\pi) \prec (\tau'',\pi'')}$$

$$(\tau,\pi) \prec (\tau,\pi) \qquad \frac{(\tau,\pi) \prec (\tau',\pi') \quad (\tau',\pi') \prec (\tau'',\pi'')}{(\tau,\pi) \prec (\tau'',\pi'')}$$

Figure 16. Type Propagation $(\tau,\pi) \prec (\tau,\pi)$

through the derivations. This leads us to the definition of the disallowed matching relation $v \lightning \pi$ (figure 15). The first two rules are for immediately disallowed matching. As we said earlier, functions are used as witnesses. Hence, a pattern is disallowed if a function is compared with either a variant pattern or the unit pattern. The three other rules propagate the information down the pattern. We thus have the following derivation showing that the matching is disallowed (side conditions are put aside for clarity).

$$\dfrac{\dfrac{\dfrac{\lambda().() \lightning ()}{v \lightning c(\pi_0,())}\; c(\pi_0,()) \xrightarrow{c} (\pi_0,())}{v \lightning c(\pi_0,())\cup()}\; c(\pi_0,())\cup() \xrightarrow{\epsilon} c(\pi_0,())}{v \lightning \pi_0}\; \pi_0 \xrightarrow{\epsilon} c(\pi_0,())\cup()$$

The analysis can then be specified as follows: an erased pattern π associated to an input type τ should be rejected if we have both $\emptyset \vdash v : s(\tau)$ and $v \lightning \pi$ for some ground substitution s and some value v.

In order to derive an algorithm, we reason in a symbolic way and represent a set of values by a type. The decomposition of a value corresponds to transitions from this type. For instance, we have the following transitions for the input type τ_0 of our example. We write below each type which part of the value $v = c((), \lambda().())$ it corresponds to.

$$\begin{array}{ccccc} \tau_0 & \xrightarrow{\epsilon} & c((),\alpha) & \xrightarrow{c} & \alpha \\ v & & v & & \lambda().() \\ & & \downarrow & & \\ & & () & & \\ & & () & & \end{array}$$

The transition graphs corresponding to the input type τ_0 and the pattern π_0 are traversed in parallel (for the disallowed matching relation a value and a pattern were traversed in parallel). The type propagation relation $(\tau,\pi) \prec (\tau,\pi)$, defined in figure 16, states that there is a path from the first pair to the second pair. The first rules define paths of length one. Epsilon transitions can happen

$$\frac{\triangleleft\tau_1 \quad \triangleleft\tau_2 \quad \tau \xrightarrow{c} (\tau_1,\tau_2)}{\triangleleft\tau} \qquad \frac{\triangleleft\tau' \quad \tau \xrightarrow{\epsilon} \tau'}{\triangleleft\tau}$$

$$\triangleleft() \qquad \triangleleft\tau_2 \to \tau_1 \qquad \triangleleft\alpha$$

Figure 17. Type Non-Emptiness $\triangleleft\tau$

independently on the type or the pattern, as the corresponding part of a value remain unchanged. On the other hand, other transitions must be performed simultaneously on the type and the pattern, as they correspond to focusing on a subpart of a value. The side conditions $\triangleleft\tau'$ and $\triangleleft\tau''$ are non-emptiness tests. They deal with cases such as a type $\tau_1 = c(\alpha,\bot)$ together with a pattern $\pi_1 = c((),())$. There is no value of type τ_1, so the pattern π_1 should be allowed. Hence, it would be incorrect to have a path from the pair (τ_1,π_1) to the pair $(\alpha,())$. And indeed, there is no such path as the assertion $\triangleleft\bot$ does not hold. Finally, the last two rules ensure that the relation is closed under reflexivity (zero-length path) and transitivity (path concatenation).

The definition of the non-emptiness relation $\triangleleft\tau$ in figure 17 can be understood as follows. For the first rule, if there is a value v_1 of type τ_1 and a value v_2 of type τ_2, then the value $c(v_1,v_2)$ has type τ. For the second rule, if we have a value of type τ', this value also has type τ. The unit value $()$ has type $()$ and we can always find a value of type $\tau_2 \to \tau_1$ (for instance, a function which accepts a value of type τ_2 then loops). Finally, as long as a type with a non-empty semantics is substituted to the type variable α (in our running example, we took the type $() \to ()$), there is a value corresponding to type α.

The specification of the analysis and the type propagation relation are related as follows.

THEOREM 16. *Given a type τ and an erased pattern π, the two following propositions are equivalent:*

- *$\emptyset \vdash v : s(\tau)$ and $v \lightning \pi$ for some ground substitution s and some value v;*
- *$(\tau,\pi) \prec (\tau',\pi')$ for some type τ' which is either a type variable α or a function type $\tau_1' \to \tau_1$, and some pattern π' which is either the unit pattern $()$ or a variant pattern $c(\pi_1,\pi_1')$.*

Given a type τ and an erased pattern π, we define $\mathrm{abstract}(\tau,\pi)$ as the negation of either of these propositions.

We can check that our example is indeed rejected. We have the four following derivations. The first one corresponds to the epsilon transition from the first node to the second node on the transition subgraph of the initial type τ_0.

$$\frac{\tau_0 \xrightarrow{\epsilon} c((),\alpha)}{(\tau_0,\pi_0) \prec (c((),\alpha),\pi_0)}$$

Then, two epsilon transitions are performed on the subgraph corresponding to the pattern π_0.

$$\frac{\pi_0 \xrightarrow{\epsilon} c(\pi_0,())\cup()}{(c((),\alpha),\pi_0) \prec (c((),\alpha), c(\pi_0,())\cup())}$$

$$\frac{c(\pi_0,())\cup() \xrightarrow{\epsilon} c(\pi_0,())}{(c((),\alpha), c(\pi_0,())\cup()) \prec (c((),\alpha), c(\pi_0,()))}$$

Finally, one step is performed simultaneously on both subgraphs.

$$\frac{c((),\alpha) \xrightarrow{c} ((),\alpha) \quad c(\pi_0,()) \xrightarrow{c} (\pi_0,()) \quad \triangleleft()}{(c((),\alpha), c(\pi_0,())) \prec (\alpha,())}$$

Thus, by transitivity, $(\tau_0,\pi_0) \prec (\alpha,())$, which is prohibited.

The two following remarks show that there is indeed an algorithm for deciding the relation. This is a consequence of remark 3.

REMARK 17. *Any occurrence of an assertion $\lhd\tau'$ in a derivation of $\lhd\tau$ is such that $\tau \prec \tau'$. As a corollary, the type non-emptiness relation is decidable.*

REMARK 18. *If $(\tau,\pi) \prec (\tau',\pi')$, then $\tau \prec \tau'$ and $\pi \prec \pi'$. As a corollary, the type propagation relation is decidable.*

An important point in language design is to be able to give precise and understandable error messages. Here, we can locate precisely the error as the type propagation relation defines a common path in the input type and the pattern to the error. Thus, for our example, we can report that the pattern is not allowed as, when matching polymorphic values of shape $c(\alpha,\ldots)$, a subvalue of type α may be matched against the subpattern $()$.

4.4 Checking for Faithfulness

Given an input type τ, an initial pattern π and a candidate translation π' of this pattern, we want to state a condition ensuring that the pattern π' behaves the same way as the initial pattern, and to provide an algorithm for checking this condition. In order to deal with the reduction rule concerning type application, the condition should be stable by type variable substitution. The semantics of function application and pattern matching should also be preserved. A possible condition is that, for any ground substitution s, if $\emptyset \vdash v : s(\tau)$, then $v \lhd s(\pi) \leadsto \theta$ if and only if $v \lhd \pi' \leadsto \theta$. But this seems hard to check. For instance, consider the following function.

$$\Lambda\alpha.\Lambda\beta.\lambda(x:\alpha).\text{match } x \text{ with } (x:\alpha)\cup(x:\beta) \to e_1 \mid _ \to e_2$$

It translates into the function:

$$\lambda(x:_).\text{match } x \text{ with } (x:_)\cup(x:_) \to e_1 \mid _ \to e_2 \ .$$

When the type variable α is bound to the type $()$ and the type variable β is bound to the type $\bot$, the subpattern $x : \beta$ never matches anything while in the translated function the second part of the union may match the input value. The substitution produced remains unchanged, but this seems somewhat accidental. Thus, we demand a stronger condition. Intuitively, we would like values to be matched the same way by both patterns.

As a running example for the remainder of the section, we consider the function:

$$\Lambda\alpha.\lambda(x:\tau_0).\text{match } x \text{ with } \pi_0 \to e$$

where the input type τ_0 and the pattern π_0 are defined as follows:

$$\begin{array}{rcl}\tau_0 & = & c((),\alpha)\\ \pi_0 & = & \mu(\beta)c(\beta,\alpha)\cup()\ .\end{array}$$

We consider the following candidate translation for this function:

$$\lambda(x:c((),_)).\text{match } x \text{ with } \pi'_0 \to e'$$

where the pattern π'_0 is the following:

$$\pi'_0 = \mu(\beta)c(\beta,_)\cup()\ .$$

The initial function can be applied to the type $()$ and the value $v = c((),())$. Then, the pattern π_0 is instantiated to the pattern:

$$\pi_1 = \mu(\beta)c(\beta,())\cup()\ .$$

We then have the following matching derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{() \lhd () \leadsto []}{() \lhd c(\pi_1,())\cup() \leadsto []}}{() \lhd \pi_1 \leadsto []} \qquad () \lhd () \leadsto []}{v \lhd c(\pi_1,()) \leadsto []}}{v \lhd c(\pi_1,())\cup() \leadsto []}}{v \lhd \pi_1 \leadsto []}$$

$$(c(\pi_1,\pi_2),c(\pi'_1,\pi'_2)) \xrightarrow{c} ((\pi_1,\pi'_1),(\pi_2,\pi'_2))$$

$$(\pi_1\cup\pi_2,\pi'_1\cup\pi'_2) \xrightarrow{\epsilon} (\pi_1,\pi'_1)$$

$$(\pi_1\cup\pi_2,\pi'_1\cup\pi'_2) \xrightarrow{\epsilon} (\pi_2,\pi'_2)$$

$$(\mu(\alpha)\pi,\mu(\alpha)\pi') \xrightarrow{\epsilon} (\pi[\mu(\alpha)\pi/\alpha],\pi'[\mu(\alpha)\pi'/\alpha])$$

$$(x:\pi,x:\pi') \xrightarrow{\epsilon} (\pi,\pi')$$

Figure 18. Lockstep Transitions $(\pi,\pi) \longrightarrow^{c} ((\pi,\pi),(\pi,\pi))$ and $(\pi,\pi) \longrightarrow^{\epsilon} (\pi,\pi)$

This is the corresponding derivation for the translated pattern: the same rules are applied, except in the case of the wildcard pattern corresponding to subpattern $()$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{() \lhd () \leadsto []}{() \lhd c(\pi'_0,_)\cup() \leadsto []}}{() \lhd \pi'_0 \leadsto []} \qquad \dfrac{\emptyset \vdash () \,\#\, _}{() \lhd _ \leadsto []}}{v \lhd c(\pi'_0,_) \leadsto []}}{v \lhd c(\pi'_0,_)\cup() \leadsto []}}{v \lhd \pi'_0 \leadsto []}$$

Clearly, given a derivation with the initial pattern, one can always find a corresponding derivation with the translated pattern. On the other hand, the converse does not necessary hold in general. Hence, we adopt the following specification of the analysis: given an input type τ, a input pattern π and a candidate translation π', we consider that the translation is faithful when for all ground substitutions s and all values v, if $\emptyset \vdash v : s(\tau)$, then any derivation of the assertion $v \lhd \pi' \leadsto \theta$ can be extended in a derivation of $v \lhd s(\pi) \leadsto \theta$ (for the same substitution θ).

A way to check this is to find all assertions $v' \lhd _ \leadsto []$ in a derivation of $v \lhd \pi' \leadsto \theta$ and verify that $v' \lhd \pi'' \leadsto []$ for the pattern π'' corresponding to the wildcard pattern $_$ in the pattern $s(\pi)$. Indeed, the subderivation of $v' \lhd _ \leadsto []$ can then be replaced by a derivation of $v' \lhd \pi'' \leadsto []$.

As in the section 4.3, we derive an algorithm by reasoning in a symbolic way and representing a set of possible values by a type. Intuitively, for each occurrence of the wildcard pattern in the translated pattern π', we need to find the pattern π'' it corresponds to in the initial pattern π and the type τ' of values it may be matched against. Then, for all values v' such that $v' \lhd \tau'$, we should check that $v' \lhd \pi''$. In other words, according to lemma 11, we should check that $\tau' <: \pi''$.

In order to find out which subpattern of the initial pattern π corresponds to each occurrence of a wildcard pattern $_$ in the translated pattern π', we define two lockstep transition relations $(\pi,\pi) \longrightarrow^{c} ((\pi,\pi),(\pi,\pi))$ and $(\pi,\pi) \longrightarrow^{\epsilon} (\pi,\pi)$ (figure 18). They are similar to the transition relations $\pi \longrightarrow^{c} (\pi,\pi)$ and $\pi \longrightarrow^{\epsilon} \pi$ but perform the same operation on two patterns simultaneously. The lockstep transition relations preserve the translation relation:

REMARK 19. *If $\pi \leadsto^{V} \pi'$ and $(\pi,\pi') \longrightarrow^{c} ((\pi_1,\pi'_1),(\pi_2,\pi'_2))$, then we have both $\pi_1 \leadsto^{V} \pi'_1$ and $\pi_2 \leadsto^{V} \pi'_2$; likewise, if $\pi_1 \leadsto^{V} \pi'_1$ and $(\pi_1,\pi'_1) \longrightarrow^{\epsilon} (\pi_2,\pi'_2)$, then we have $\pi_2 \leadsto^{V} \pi'_2$.*

The transition subgraph corresponding to our example patterns π_0 and π'_0 is given in figure 19. In this graph, we can read that the type variable α corresponds to the wildcard pattern $_$.

A type propagation relation $(\tau,\pi,\pi) \prec (\tau,\pi,\pi)$ is defined in figure 20. It is very similar to the type propagation relation of the

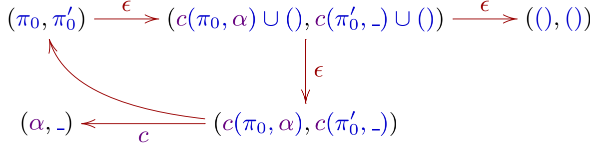

Figure 19. Example of Lockstep Transition Graph

$$\frac{\tau \xrightarrow{\epsilon} \tau'}{(\tau,\pi_1,\pi_2) \prec (\tau',\pi_1,\pi_2)} \qquad \frac{(\pi_1,\pi_2) \xrightarrow{\epsilon} (\pi_1',\pi_2')}{(\tau,\pi_1,\pi_2) \prec (\tau,\pi_1',\pi_2')}$$

$$\frac{\tau \xrightarrow{c} (\tau',\tau'') \quad (\pi_1,\pi_2) \xrightarrow{c} ((\pi_1',\pi_2'),(\pi_1'',\pi_2'')) \quad \tau'' \bowtie \pi_2''}{(\tau,\pi_1,\pi_2) \prec (\tau',\pi_1',\pi_2')}$$

$$\frac{\tau \xrightarrow{c} (\tau',\tau'') \quad (\pi_1,\pi_2) \xrightarrow{c} ((\pi_1',\pi_2'),(\pi_1'',\pi_2'')) \quad \tau' \bowtie \pi_2'}{(\tau,\pi_1,\pi_2) \prec (\tau'',\pi_1'',\pi_2'')}$$

$$(\tau,\pi_1,\pi_2) \prec (\tau,\pi_1,\pi_2) \qquad \frac{(\tau,\pi_1,\pi_2) \prec (\tau',\pi_1',\pi_2') \quad (\tau',\pi_1',\pi_2') \prec (\tau'',\pi_1'',\pi_2'')}{(\tau,\pi_1,\pi_2) \prec (\tau'',\pi_1'',\pi_2'')}$$

Figure 20. Type Propagation $(\tau,\pi,\pi) \prec (\tau,\pi,\pi)$

$$\frac{\tau' \bowtie \pi \quad \tau \xrightarrow{\epsilon} \tau'}{\tau \bowtie \pi} \qquad \frac{\tau \bowtie \pi' \quad \pi \xrightarrow{\epsilon} \pi'}{\tau \bowtie \pi}$$

$$\frac{\tau' \bowtie \pi' \quad \tau'' \bowtie \pi'' \quad \tau \xrightarrow{c} (\tau',\tau'') \quad \pi \xrightarrow{c} (\pi',\pi'')}{\tau \bowtie \pi} \qquad () \bowtie () \qquad \frac{\lhd \tau}{\tau \bowtie _}$$

Figure 21. Non-Disjointness $\tau \bowtie \pi$

previous section (figure 16). Beside the use of lockstep transitions, the only notable change is the replacement of the side-conditions of the form $\lhd\tau$ by side-conditions of the form $\tau \bowtie \pi$, which states that the type τ and the pattern π have at least one common value. The reason for this difference is that we are now interested in full derivations of the matching relation $v \lhd \pi \rightsquigarrow \theta$ rather than simply partial derivations. Hence, the values considered must match both the type and the pattern.

The relation $\tau \bowtie \pi$ is defined in figure 21. It only makes sense when the pattern π is erased and $\mathrm{abstract}(\tau,\pi)$ holds. For the first rule, if there is a value common to type τ' and pattern π, then it is included in type τ and, therefore, it is also common to type τ and pattern π. The second rule is similar. For the third rule, if there is a value v' common to type τ' and pattern π' and a value v'' common to type τ'' and pattern π'', then the value $c(v',v'')$ is common to type τ and pattern π. The last two rules are straightforward.

The specification of the analysis and the type propagation relation are related as follows.

THEOREM 20. *Given an input type τ, a pattern π and a candidate translation π' of the pattern, assuming that the pattern π' is erased, that the relation $\pi \rightsquigarrow^{\emptyset} \pi'$ holds and that $\mathrm{abstract}(\tau,\pi')$, the two following propositions are equivalent:*

- *for any ground substitution s and any value v, if $\emptyset \vdash v : s(\tau)$, then any derivation of the assertion $v \lhd \pi' \rightsquigarrow \theta$ can be extended in a derivation of $v \lhd s(\pi) \rightsquigarrow \theta$ (for the same substitution θ);*
- *for any type τ_1 and pattern π_1 such that $(\tau,\pi,\pi') \prec (\tau_1,\pi_1,_)$, we have $\tau_1 <: \pi_1$.*

We define the relation $\mathrm{faithful}(\tau,\pi,\pi')$ as either of these two propositions.

In our example, the assertion $(\tau_0,\pi_0,\pi_0') \prec (\tau_2,\pi_2,_)$ holds if and only if $\tau_2 = \pi_2 = \alpha$. As $\alpha <: \alpha$, the translation is indeed faithful.

The two following remarks show that there is indeed an algorithm for deciding the relation. This is a consequence of remark 3.

REMARK 21. *Any occurrence of an assertion $\tau' \bowtie \pi'$ in a derivation of $\tau \bowtie \pi$ is such that $\tau \prec \tau'$ and $\pi \prec \pi'$. As a corollary, the non-disjointness relation is decidable.*

REMARK 22. *If $(\tau,\pi,\pi') \prec (\tau_1,\pi_1,\pi_1')$, then $\tau \prec \tau_1$, $\pi \prec \pi_1$ and $\pi' \prec \pi_1'$. As a corollary, the type propagation relation is decidable.*

As with the previous analysis, one can provide an understandable error message using the type propagation relation to locate precisely the error.

4.5 Faithfulness of the Translation

The faithfulness of the translation is expressed by the two following simulation theorems. The reduction steps corresponding to type application are eliminated by reduction. This explains the asymmetry between the two theorems.

THEOREM 23. *If $e_1 \longrightarrow e_2$ and $\emptyset \vdash e_1 : \sigma \rightsquigarrow e_1'$, then either $\emptyset \vdash e_2 : \sigma \rightsquigarrow e_1'$ or there exists a term e_2' such that $e_1' \longrightarrow e_2'$ and $\emptyset \vdash e_2 : \sigma \rightsquigarrow e_2'$.*

THEOREM 24. *If $e_1' \longrightarrow e_2'$ and $\emptyset \vdash e_1 : \sigma \rightsquigarrow e_1'$, then there exists a term e_2 such that $e_1 \longrightarrow^{+} e_2$ and $\emptyset \vdash e_2 : \sigma \rightsquigarrow e_2'$.*

5. Type Inference in Patterns

Suppose we want to extract the two components of values of type $\tau_0 = c(\tau_1,\tau_2)$. According to the typing rules, one has to write a pattern $c(x:\tau_1, y:\tau_2)$, repeating the types τ_1 and τ_2. One should be able to write instead a pattern $\pi_0 = c(x:_, y:_)$ and rely on a type inference algorithm to compute the types of bound variables. Due to lack of space, we only sketch the algorithm.

The two occurrences of the wildcard pattern need to be distinguished. Hence, we annotate them: $\pi_0 = c(x:_{(1)}, y:_{(2)})$. The type propagation relation of figure 20 is then used. We have $(\tau_0,\pi_0,\pi_0) \prec (\tau_1,_{(1)},_{(1)})$ and $(\tau_0,\pi_0,\pi_0) \prec (\tau_2,_{(2)},_{(2)})$. (The translation of pattern π_0 is itself.) The types corresponding to each wildcard patterns are collected. In this case, there is a single type for each pattern. Then, each wildcard is replaced with the union of its associated types. This yields a pattern π_1 containing no wildcard subpatterns. Here, we get $\pi_1 = c(x:\tau_1, y:\tau_2)$. Finally, we take $\lfloor \pi_1 \rfloor$ as the bindings associated to the pattern π_0.

6. Related Works

The type system we present here is very close to the XDuce type system [7, 8] but also features polymorphism and function types. This is not immediately apparent as XDuce uses syntactic sugar for values and types so that its values look like fragments of XML documents.

CDuce [3] has a very powerful type system, which is basically an extension of XDuce type system with function types and intersection types. Pattern matching on functions involves subtype checks at runtime. We wanted to keep our type system as simple as possible, so we did not considered intersection types.

Hosoya, Frisch and Castagna propose an extension of XDuce type system with polymorphism [6]. Though there is no explicit type intersection construction, their types are closed under intersection. This is an elegant feature and it makes it possible to encode bounded quantification. But it interacts badly with polymorphism. In particular, the straightforward semantic interpretation of subtyping used in XDuce and CDuce yields a subtyping relation which is difficult to check. Hence, they propose a different interpretation based on marking values with type variables. They do not propose anything similar to our untyped semantics: their pattern matching construction is allowed to break abstraction and type variables are not allowed in patterns.

7. Future Work

It would be interesting to extend our calculus with bounded quantification. An easy way is to provide a bound when abstracting over a type variable and check the bound at type application. The subtyping relation can be extended so that a type variable is a subtype of its bound. But this does not yield a complete subtyping relation: we encounter the same difficulties as Hosoya, Frisch and Castagna [6]. Our analyses should probably also be refined to take into account the bounds. We have not yet studied how they should be modified.

Acknowledgments

I thank Giuseppe Castagna, Alain Frisch and Haruo Hosoya for useful discussions on the subject. I thank the anonymous referees for their detailed and insightful feedbacks on the paper.

References

[1] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In R. Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2-4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63–81. Springer-Verlag, April 1997.

[2] R. Cartwright. Types as intervals. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 22–36. ACM Press, 1985.

[3] CDuce Development Team. *CDuce Programming Language User's Manual*. Available from `http://www.cduce.org/documentation.html`.

[4] A. Frisch. *Théorie, conception et réalisation d'un langage adapté à XML.* Thèse de doctorat, University of Paris 7, 2004.

[5] V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2003. Preliminary version in ICFP 2000.

[6] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 50–62. ACM Press, 2005.

[7] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[8] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005. Short version in ICFP 2000.

[9] M. Y. Levin. Compiling regular patterns. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 65–77, New York, NY, USA, 2003. ACM Press.

[10] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[11] A. K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, 1995.

[12] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.