Subtyping Union Types

May 5, 2004

Jérôme Vouillon

Abstract

Subtyping rules can be fairly complex for union types, due to interactions with other types, such as function types. Furthermore, these interactions turn out to depend on the calculus considered: for instance, a call-by-value calculus and a call-by-name calculus will have different possible subtyping rules. In order to abstract ourselves away from this dependence, we consider a fairly large class of calculi. We define types in a semantic fashion, as sets of terms. Then, a type can be a subtype of another type if its denotation is included in the denotation of the other type. We first consider a simple type system with union, function, pair and constant types. Using inference rules, we specify a subtyping relation which is both sound and complete with respect to the class of calculi. We then extend this result to a richer type system with ML-style polymorphism and type constructors. We expect this framework to allow the study of subtyping relations that only hold for some calculi by restricting the class considered, and to allow the study of subtyping relations for richer type systems by enriching the class.

1 Introduction

The design of a subtyping relation for a language with a rich type system is hard. The subtyping relation should satisfy conflicting requirements. On the one hand, one would like the relation to have strong theoretical foundations, rather than being defined in an ad hoc, purely algorithmic, fashion. It is therefore tempting to base it on the semantics of the language. But, on the other hand, one should be careful not to tie it too tightly to a particular language. Especially, one should avoid accidental special cases which happen to hold only in the language considered. Indeed, the relation should be robust in order to accommodate future language extensions. It should also be simple enough so that the users can understand it, and should possess good algorithmic properties: checking whether two types are in a subtyping relation should be reasonably simple and efficient.

We should emphasize the fact that the possible subtyping relations depend on the language considered by providing some examples. Let us first provide some rough intuition about types. We take the view that well-typed terms may diverge but will evaluate without error. A term of type $\bot$ is a term that always diverges. A term of type $\top$ evaluates without error. A term of type $\tau' \to \tau$ behaves like a term of type τ once applied to a term of type τ'. A term of type $\tau \cup \tau'$ behaves as a term of type either τ or τ'. We write $\tau <: \tau'$ to mean that τ is a subtype of τ' and $\tau = \tau'$ to mean that τ and τ' are equivalent, that is, subtype of one another. We can now present some typing relations that only hold under some conditions on the calculus.

- In a call-by-value language, we can have $\top <: \bot \to \bot$. Indeed, suppose we take a term e of type $\top$. When we apply it to a term e' of type $\bot$ (that is a term whose evaluation does not terminate), we get a term $e\,e'$ whose evaluation does not terminate. So, the term e has type $\bot \to \bot$.
- In a call-by-name language, we can have $\top <: \top \to \top$ (as in Pierce's thesis [14, p. 20]). Indeed, as argued by Dami [2], in a call-by-name language, it makes sense to consider $\top$ as the set of all terms. Then, types need to be interpreted in a slightly unusual way. A well-typed terms does not necessarily evaluate without error. Rather, only terms whose type τ is not equivalent to $\top$ have these properties. Then, if we apply a term of type $\top$ to another term of type $\top$, we get a term of type $\top$, which corresponds to the subtyping assertion $\top <: \top \to \top$.
- In a call-by-value language, we can have the distributivity law $(\tau_1 \cup \tau_2) \times \tau = (\tau_1 \times \tau) \cup (\tau_2 \times \tau)$. This law does not hold in a call-by-name language with non-determinism. Indeed, a term of type $(\tau_1 \cup \tau_2) \times \tau$ may well be a pair whose first component evaluates sometimes to a value of type τ_1 and sometimes to a value of type τ_2. Still, it holds in a call-by-need language with non-determinism, as an expression is then evaluated at most once.
- In a deterministic calculus, union of function types $\tau \to \tau'$ obey very special subtyping rules when τ is finite (as observed by Damm [3]). The reason is that these types are isomorphic to tuple types.

On the other hand, some rules seem very robust:

- The arrow is covariant on the left and contravariant on the right: if $\tau_1 <: \tau'_1$ and $\tau'_2 <: \tau_2$, then $\tau_2 \to \tau_1 <: \tau'_2 \to \tau'_1$;
- Union types are least upper bounds: if $\tau <: \tau_1$ or $\tau <: \tau_2$, then $\tau <: \tau_1 \cup \tau_2$; if $\tau_1 <: \tau$ and $\tau_2 <: \tau$, then $\tau_1 \cup \tau_2 <: \tau$.

The aim of this paper is to develop a framework in which we can substantiate the above claims, and thus understand which subtyping assertions $\tau <: \tau'$ hold "by accident" (depending on some specific properties of a language), and which are more universal (valid for a large class of calculi).

Rather than choosing a particular calculus, we specify a broad class of calculi in a fairly abstract way. For each calculus, we interpret a type τ as a set of terms $[\![\tau]\!]$. Given a subtyping relation $<:$, defined for instance by inference rules, we can state that a subtyping assertion $\tau <: \tau'$ is *valid* when $[\![\tau]\!] \subseteq [\![\tau']\!]$. Then, a subtyping relation is *sound* when any derivable subtyping assertion is valid in all calculi. It is *complete* when every universally valid assertion can be derived. We present a relation which is both sound and complete for the class of calculi considered. Though this is not addressed in this paper, it would then be possible to study relations which are only sound under some assumptions by restricting the class of calculi.

The paper is organized as follows. The class of calculi is defined in Sect. 3. A particular instance is given in Sect. 4. We present a simple type system, with union types, constant types, pair types, and function types, define a subtyping relation and prove the soundness and completeness of the relation (Sect. 5). We refine the type system with ML-style parametric polymorphism and type constructors (Sect. 6). We conclude by presenting related work (Sect. 7) and directions for future work (Sect. 8).

2 Order Theory

Preorder A *preorder* $\preceq$ is a binary relation over a set X which is reflexive and transitive. A *preordered set* is a pair $(X,\preceq)$ of a set X and a preorder $\preceq$ over the set X.

Closure A *closure operator* over a preordered set $(X,\preceq)$ is a function which associates to each element x of X an element $\overline{x}$ of X such that:

- $x \preceq \overline{x}$ (extensive);
- $\overline{\overline{x}} = \overline{x}$ (idempotent);
- if $x \preceq x'$, then $\overline{x} \preceq \overline{x'}$ (monotone).

An element x is *closed* when $\overline{x} = x$. The greatest lower bound of a family of closed elements is closed.

Galois Connection Let $(X,\preceq)$ and $(Y,\sqsubseteq)$ be two preordered sets. A *Galois connection* is a pair of two functions $f : X \to Y$ and $g : Y \to X$ such that for all $x \in X$ and $y \in Y$, $x \preceq g(y)$ iff $f(x) \sqsubseteq y$. A Galois connection (f,g) induces a closure operator $g \circ f$ over the preordered set $(X,\preceq)$.

Downward Closed Set A subset A of a preordered set $(X,\preceq)$ is *downward closed* when, for every x in A and y in X, if $y \preceq x$, then y is in A.

Directed Set A subset A of a preordered set $(X,\preceq)$ is *directed* when it is non-empty and when each pair of elements of this subset has an upper bound in this subset.

Prime Element An element x of a preordered set $(X,\preceq)$ is *prime* when $x \preceq y \vee z$ implies $x \preceq y$ or $x \preceq z$ whenever the least upper bound $y \vee z$ of y and z exists.

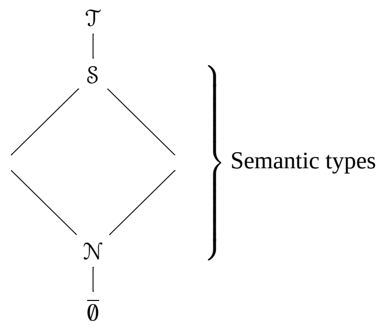
3 A Class of Abstract Calculi

3.1 Informal Presentation and Definitions

We would like to study subtyping for a class of calculi with functions, pairs and constants. The first step is to associate to each type τ its semantics $[\![\tau]\!]$, that is, the set of terms of type τ. We type terms rather than values, because the notion of terms is more fundamental: the notion of value depends on the language considered. Besides, it is not always possible to reduce the behavior of a term to the behavior of a set of values, especially in call-by-name calculi. This is actually possible in the calculus of Sect. 4, but only because we made some specific choices about types.

As it turns out, it is convenient to only consider sets of terms that satisfies a given *closure* property: the closure $\overline{\mathcal{E}}$ of a set of terms $\mathcal{E}$ is the set of terms that cannot be distinguished from the terms in $\mathcal{E}$. Notice that the choice of a closure operator is not neutral. It dictates what can be observed about terms.

Types categorize terms according to their behavior. We should be able to use them to avoid some unsafe behavior, typically runtime errors. So, we distinguish a set $\mathcal{S}$ of *safe terms*. Dually, we define a set $\mathcal{N}$ of *neutral terms* (typically, terms that loop) as the intersection of all non-empty closed terms. We call *semantic type* a closed set of terms included in $\mathcal{S}$ and including $\mathcal{N}$. The set of semantic types is a complete lattice. We therefore have the diagram below (where the set $\mathcal{T}$ is the set of all terms):



We require the semantics $[\![\tau]\!]$ of a syntactic type τ to be a semantic type. For the sake of flexibility, we don't assume $\mathcal{S} \neq \mathcal{T}$ nor $\mathcal{N} \neq \overline{\emptyset}$. In particular, we can take $\mathcal{S} = \mathcal{T}$ if we want to interpret the type $\top$ as the set of all terms. The disadvantage is then that, in order to get a soundness result, one must characterize the types that only contain safe terms.

It seems really important in practice to distinguish a set of safe terms $\mathcal{S}$ from the set of all terms $\mathcal{T}$, and a set of neutral terms $\mathcal{N}$ from the least closed set $\overline{\emptyset}$. Indeed, in Sect. 4, we will have $\mathcal{S} \neq \mathcal{T}$ and $\mathcal{N} = \overline{\emptyset}$, but in [17], we had $\mathcal{S} \neq \mathcal{T}$ and $\mathcal{N} \neq \overline{\emptyset}$, and in [13], we had $\mathcal{S} = \mathcal{T}$ and $\mathcal{N} \neq \overline{\emptyset}$. Finally, in the case of *reducibility candidates* [8], one has both $\mathcal{S} \neq \mathcal{T}$ and $\mathcal{N} \neq \overline{\emptyset}$.

Note that if we take $\mathcal{N} = \overline{\emptyset}$, then all continuous functions are strict. Indeed, if f is continuous, then $f^{-1}(\overline{\emptyset})$ is closed and therefore contains $\overline{\emptyset}$. Thus, if we want constant functions to be continuous, which seems reasonable, we need to have $\mathcal{N} \neq \overline{\emptyset}$.

Let us now sketch how we define the semantics of types. The idea is that we want to be able to build more complex typed terms by assembling smaller typed terms according to simple (typing) rules. For instance:

$$\textsc{App}\ \frac{e:\tau'\to\tau \qquad e':\tau'}{e\,e':\tau} \qquad \textsc{Fst}\ \frac{e:\tau\times\tau'}{\texttt{fst}\,e:\tau} \qquad \textsc{Snd}\ \frac{e:\tau\times\tau'}{\texttt{snd}\,e:\tau'}$$

The rules above suggest the following inclusions.

$$\begin{array}{lcl} [\![\tau'\to\tau]\!] & \subseteq & \{e \in \mathcal{S} \mid \forall e' \in [\![\tau']\!].\, e\,e' \in [\![\tau]\!]\} \\ [\![\tau\times\tau']\!] & \subseteq & \{e \in \mathcal{S} \mid \texttt{fst}\,e \in [\![\tau]\!] \wedge \texttt{snd}\,e \in [\![\tau']\!]\} \end{array}$$

These inclusions ensure the *soundness* of the typing rules. In order to reason about types, it is important to have a more precise characterization of their semantics. It seems therefore natural to replace these inclusions by an equality.

$$\begin{array}{lcl} [\![\tau'\to\tau]\!] & = & \{e \in \mathcal{S} \mid \forall e' \in [\![\tau']\!].\, e\,e' \in [\![\tau]\!]\} \\ [\![\tau\times\tau']\!] & = & \{e \in \mathcal{S} \mid \texttt{fst}\,e \in [\![\tau]\!] \wedge \texttt{snd}\,e \in [\![\tau']\!]\} \end{array}$$

This is not an unimportant choice. In particular, we should not expect an equality if we want a parametric property to hold: non-parametric functions should be rejected. (Actually, parametricity fit in our framework by interpreting $\mathcal{T}$ as a set of pairs of terms rather than a set of terms.) Also, this assumes that the functions have an extensional behavior. Therefore, we do not handle calculi with side effects.

Now the sets $[\![\tau' \to \tau]\!]$ and $[\![\tau \times \tau']\!]$ must be semantic types. The definitions above clearly ensure that these sets are included in $\mathcal{S}$. They must also be closed and must contain $\mathcal{N}$. We cannot force this by making the sets larger, as this would violate the soundness conditions. Instead, we make more assumptions on the calculi. We say that a function is *continuous* when the inverse image of a closed set is closed, that a function is *strict* when the inverse image of $\mathcal{N}$ is included in $\mathcal{N}$. We can prove inductively that the sets $[\![\tau' \to \tau]\!]$ and $[\![\tau \times \tau']\!]$ are closed if $\mathcal{S}$ is closed and the functions `fst`, `snd`, and $e \mapsto e\,e'$ (for all terms e' in $\mathcal{S}$) are continuous. Similarly, we can prove that these sets contain $\mathcal{N}$ if $\mathcal{N} \subseteq \mathcal{S}$ and the same functions are strict. This appears more clearly if the equations above are rewritten in a more algebraic form.

$$\begin{array}{rcl} [\![\tau' \to \tau]\!] & = & \mathcal{S} \cap \bigcap_{e' \in [\![\tau']\!]} \{e \mid e\,e' \in [\![\tau]\!]\} \\ [\![\tau \times \tau']\!] & = & \mathcal{S} \cap \mathtt{fst}^{-1}([\![\tau]\!]) \cap \mathtt{snd}^{-1}([\![\tau']\!]) \end{array}$$

It is really natural for all these functions to be strict. The continuity properties may seem harder to achieve. We will see in Sect. 4.2, that it is actually straightforward to define a closure operator ensuring these properties.

The calculi also have constants, denoted κ. These constants are assumed to be safe. We define a singleton type κ for each constant κ. Its semantics is the least closed set of term containing the constant κ:

$$[\![\kappa]\!] = \overline{\{\kappa\}}\ .$$

3.2 Formal Specification

The class of calculi we consider are the calculi to which we can associate:

- a set of terms $\mathcal{T}$;
- a closure operator $\mathcal{E} \mapsto \overline{\mathcal{E}}$ on terms;
- a closed subset $\mathcal{S} \subseteq \mathcal{T}$ of safe terms;
- three operators:

$$\begin{array}{lccccc} \mathtt{app}: & \mathcal{T} & \to & \mathcal{T} & \to & \mathcal{T} \\ & e & \mapsto & e' & \mapsto & e\,e' \\ \mathtt{fst}: & \mathcal{T} & \to & \mathcal{T} & & \\ & e & \mapsto & \mathtt{fst}\,e & & \\ \mathtt{snd}: & \mathcal{T} & \to & \mathcal{T} & & \\ & e & \mapsto & \mathtt{snd}\,e & & \end{array}$$

  such that $e \mapsto e\,e'$ (where $e' \in \mathcal{S}$), `fst` and `snd` are continuous and strict;
- a set of constants $\kappa \in \mathcal{S}$.

3.3 Semantic Operations

We define one operation on set of terms for each type construction we have in mind: bottom type, union of two types, function types, pair types and constant types. Note that the semantic union $\boxed{\cup}$ of two sets of terms is not simply their union. Indeed, there may be some terms that are in neither of the sets but cannot be distinguished from the terms in the union of both sets. These operations are used to define the semantics of types in a straightforward fashion in Sect. 5.1.

$$\begin{array}{lcl} \boxed{\bot} & = & \mathcal{N} \\ \mathcal{E}\,\boxed{\cup}\,\mathcal{E}' & = & \overline{\mathcal{E} \cup \mathcal{E}'} \\ \mathcal{E}'\,\boxed{\to}\,\mathcal{E} & = & \{e \in \mathcal{S} \mid \forall e' \in \mathcal{E}'.e\,e' \in \mathcal{E}\} \\ \mathcal{E}\,\boxed{\times}\,\mathcal{E}' & = & \{e \in \mathcal{S} \mid \mathtt{fst}\,e \in \mathcal{E} \wedge \mathtt{snd}\,e \in \mathcal{E}'\} \\ \boxed{\kappa} & = & \overline{\{\kappa\}} \end{array}$$

It is clear that all these operations map semantic types to semantic types.

4 A Concrete Calculus

We present a particular instance of the class of calculi considered. This calculus is used in Sect. 5 to prove the completeness of a subtyping relation. It actually turns out to be *universal*, in the sense that a subtyping relation is complete if and only if it is complete for this particular calculus.

4.1 The Calculus

The calculus we consider is a call-by-name calculus with pairs and constants. Its main remarkable characteristics are a notion of errors, a strict `let` binder and two non-deterministic choice operators. The syntax of the calculus is given by the following grammar:

e	::=	x	variable
		$\lambda x.e$	abstraction
		$e\,e$	application
		(e,e)	pair
		$\mathtt{fst}\,e$	first projection
		$\mathtt{snd}\,e$	second projection
		κ	constant
		$\mathtt{if}\ e = \kappa\ \mathtt{then}\ e\ \mathtt{else}\ e$	conditional
		$e \sqcup e$	erratic choice
		$e \vee e$	error-avoiding choice
		$\mathtt{let}\ x = e\ \mathtt{in}\ e$	strict let
		`error`	error

The set of constants κ is supposed to be infinite. A bigstep semantics is given in Fig. 1. The values are a subgrammar of terms:

v	::=	$\lambda x.e$	abstraction
		(e,e)	pair
		κ	constant
		`error`	error

In the reduction rules, we write $v \neq v'$ where v' describes a specific shape of values (for instance, v' is (e_1,e_2)) to mean that v is not of the same shape as v'.

The semantics is rather standard and unsurprising. We simply say a few words about the two non-deterministic choice operators. The first one $e \sqcup e'$ is the standard erratic operator: $e \sqcup e' \Downarrow v$ if and only if either $e \Downarrow v$ or $e \Downarrow v$. The second one $e \vee e'$ is a bit like an angelic choice operator, but instead of attempting to avoid non-termination, it attempts to avoid errors. Another way of understanding this operator is to consider it as a symmetric variant of a `catch` operator: it evaluates one of the terms e or e' and, if this fails, falls back to evaluating the other term. The unusual notations emphasize the fact

$$\textsc{Var-Error}\quad x \Downarrow \texttt{error} \qquad \textsc{Abs}\quad \lambda x.e \Downarrow \lambda x.e \qquad \textsc{App}\quad \frac{e \Downarrow \lambda x.e_1 \qquad e_1[e'/x] \Downarrow v}{e\,e' \Downarrow v} \qquad \textsc{App-Error}\quad \frac{e \Downarrow v \qquad v \neq \lambda x.e_1}{e\,e' \Downarrow \texttt{error}} \qquad \textsc{Pair}\quad (e_1,e_2) \Downarrow (e_1,e_2) \qquad \textsc{Fst}\quad \frac{e \Downarrow (e_1,e_2) \qquad e_1 \Downarrow v}{\texttt{fst}\,e \Downarrow v}$$

$$\textsc{Fst-Error}\quad \frac{e \Downarrow v \qquad v \neq (e_1,e_2)}{\texttt{fst}\,e \Downarrow \texttt{error}} \qquad \textsc{Snd}\quad \frac{e \Downarrow (e_1,e_2) \qquad e_2 \Downarrow v}{\texttt{snd}\,e \Downarrow v} \qquad \textsc{Snd-Error}\quad \frac{e \Downarrow v \qquad v \neq (e_1,e_2)}{\texttt{snd}\,e \Downarrow \texttt{error}} \qquad \textsc{Constant}\quad \kappa \Downarrow \kappa \qquad \textsc{If-Equal}\quad \frac{e \Downarrow \kappa \qquad e' \Downarrow v}{\texttt{if}\ e = \kappa\ \texttt{then}\ e'\ \texttt{else}\ e'' \Downarrow v}$$

$$\textsc{If-Not-Equal}\quad \frac{e \Downarrow \kappa' \qquad \kappa \neq \kappa' \qquad e'' \Downarrow v}{\texttt{if}\ e = \kappa\ \texttt{then}\ e'\ \texttt{else}\ e'' \Downarrow v} \qquad \textsc{If-Error}\quad \frac{e \Downarrow v \qquad v \neq \kappa'}{\texttt{if}\ e = \kappa\ \texttt{then}\ e'\ \texttt{else}\ e'' \Downarrow \texttt{error}} \qquad \textsc{Para-Left}\quad \frac{e \Downarrow v}{e \sqcup e' \Downarrow v} \qquad \textsc{Para-Right}\quad \frac{e' \Downarrow v}{e \sqcup e' \Downarrow v} \qquad \textsc{Catch-Left}\quad \frac{e \Downarrow v \qquad v \neq \texttt{error}}{e \vee e' \Downarrow v}$$

$$\textsc{Catch-Right}\quad \frac{e' \Downarrow v \qquad v \neq \texttt{error}}{e \vee e' \Downarrow v} \qquad \textsc{Catch-Error}\quad \frac{e \Downarrow \texttt{error} \qquad e' \Downarrow \texttt{error}}{e \vee e' \Downarrow \texttt{error}} \qquad \textsc{Let}\quad \frac{e \Downarrow v \qquad v \neq \texttt{error} \qquad e'[v/x] \Downarrow v'}{\texttt{let}\ x = e\ \texttt{in}\ e' \Downarrow v'} \qquad \textsc{Let-Error}\quad \frac{e \Downarrow \texttt{error}}{\texttt{let}\ x = e\ \texttt{in}\ e' \Downarrow \texttt{error}}$$

$$\textsc{Error}\quad \texttt{error} \Downarrow \texttt{error}$$

Figure 1. Semantics

that both operations correspond to a least upper bound, as we will see in Sect. 4.4.3.

We define the following diverging term (there is no value v such that $\texttt{diverge} \Downarrow v$):

$$\texttt{diverge} = (\lambda x.xx)\,(\lambda x.xx) \ .$$

4.2 Orthogonality

Remember that we need to specify not only a calculus but also a closure operator on sets of terms. We first present a generic way of building a closure operator. The choice of a particular closure operator is made in the next section 4.3.

A convenient way to define a closure operator on sets of terms is by *orthogonality* between terms and contexts. At this point, it does not matter what the set of contexts is. We just assume given an orthogonality relation $e \perp c$ between contexts c and terms e. Its intended meaning is that the term e behaves properly in the context c. We define the *orthogonal* of a set of terms $\mathcal{E}$ as the set of contexts in which all terms in $\mathcal{E}$ behave properly:

$$\mathcal{E}^{\perp} = \{c \mid \forall e \in \mathcal{E}. e \perp c\} \ .$$

Conversely, we define the *orthogonal* of a set of contexts $\mathcal{C}$ as the set of terms that behave properly in all the contexts in $\mathcal{C}$:

$$\mathcal{C}^{\perp} = \{e \mid \forall c \in \mathcal{C}. e \perp c\} \ .$$

These two functions defines a Galois connection between sets of terms and sets of contexts. The important point here is that the composition of these two functions, which associates to a set of terms $\mathcal{E}$ its *biorthogonal* $\overline{\mathcal{E}} = \mathcal{E}^{\perp\perp}$, is a closure operator. (Dually, we can define a closure operator which associates to a set of contexts its biorthogonal $\overline{\mathcal{C}} = \mathcal{C}^{\perp\perp}$.)

Furthermore, we can rely on the following lemma to guide us in the choice of a set of contexts. Let f be a function from terms to terms, and g be a function from contexts to contexts. We say that g is an *adjoint* of f iff

$$f(e) \perp c \Leftrightarrow e \perp g(c) \ .$$

LEMMA 1. *If a function f has an adjoint g, then it is continuous.*

PROOF. Let f be a function with an adjoint g. We first prove that the following diagram commutes ($\mathcal{T}$ is the set of all terms, Σ is the set of all contexts).

$$\begin{tikzcd}
\mathcal{P}(\Sigma) \arrow[r, "\perp"] \arrow[d, "g"] & \mathcal{P}(\mathcal{T}) \arrow[d, "f^{-1}"] \\
\mathcal{P}(\Sigma) \arrow[r, "\perp"] & \mathcal{P}(\mathcal{T})
\end{tikzcd}$$

(In other words, $f^{-1}(\mathcal{C}^{\perp}) = (g(\mathcal{C}))^{\perp}$.)

Indeed, the following propositions are equivalent:

$$\begin{array}{ll}
e \in f^{-1}(\mathcal{C}^{\perp}) & f(e) \in \mathcal{C}^{\perp} \\
\forall c \in \mathcal{C}.\ f(e) \perp c & \forall c \in \mathcal{C}.\ e \perp g(c) \\
\forall c \in g(\mathcal{C}).\ e \perp c & e \in (g(\mathcal{C}))^{\perp}
\end{array}$$

We finish the proof by taking $\mathcal{C} = \mathcal{E}^{\perp}$:

$$f^{-1}(\overline{\mathcal{E}}) = f^{-1}(\mathcal{E}^{\perp\perp}) = (g(\mathcal{E}^{\perp}))^{\perp}$$

Therefore, the inverse image of a closed set is closed. □

4.3 The Closure Operator

Using the tools just developed, we can now specify the closure operator. Contexts are given by the following grammar:

$$\begin{array}{lcll}
c & ::= & Id & \text{identity} \\
 & & c \circ F & \text{frame concatenation} \\
 & & c \vee c & \text{join} \\
F & ::= & \_\,e & \\
 & & \texttt{fst}\,\_ & \\
 & & \texttt{snd}\,\_ & \\
 & & \texttt{if}\ \_ = \kappa\ \texttt{then}\ e\ \texttt{else}\ e &
\end{array}$$

A context c can be viewed as a stack, with a weird "stack join" operation, and F can be viewed as a stack frame. Every context c and term e may be combined to generate a term denoted $c\,e$ and defined as follows:

$$\begin{array}{lcl} Id\,e & = & e \\ (c \circ F)\,e & = & c\,(F[e]) \\ (c \vee c')\,e & = & \texttt{let}\, x = e \,\texttt{in}\, ((c\,x) \vee (c'\,x)) \\ & & \qquad\qquad \text{where } x \text{ is fresh} \ . \end{array}$$

A term e is safe when it does not reduce to the error:

$$\mathcal{S} = \{e \,|\, \neg(e \Downarrow \texttt{error})\} \ .$$

The orthogonality relation is defined by:

$$e \perp c \text{ iff } c\,e \in \mathcal{S} \ .$$

As indicated in the previous section 4.2, this induces a closure operator on sets of terms.

The choice of this operator is crucial: it controls what can be observed by typed terms. We should therefore explain how the contexts are chosen. The identity context Id ensures that $\mathcal{S}$ is closed. The frame concatenation operation $c \circ F$ ensures that each frame is continuous (by Lemma 1). The join operation $c \vee c'$ allows for disjunctive tests. For instance, the context $(Id \circ \texttt{fst}\,\_) \vee (Id \circ \_\texttt{diverge})$ will behave properly against terms which reduce to either a pair or a function, but will fail with other terms. This ensures that the closed union $\overline{\mathcal{E}} \boxed{\sqcup} \overline{\mathcal{E}'}$ of two semantic types $\overline{\mathcal{E}}$ and $\overline{\mathcal{E}'}$ is not "too large" (see Sect. 4.4.2 for a more precise characterization of this property).

An important consequence of this definition of closed sets is that it gives a syntactic proof technique to show an inclusion between two closed sets of terms. Indeed, the following assertions are equivalent:

- $\overline{\mathcal{E}} \subseteq \overline{\mathcal{E}'}$;
- for all contexts c, if $e' \perp c$ for all terms $e' \in \mathcal{E}'$, then $e \perp c$ for all terms $e \in \mathcal{E}$;
- for all contexts c, if $c\,e \Downarrow \texttt{error}$ for some term $e \in \mathcal{E}$, then there exists a term $e' \in \mathcal{E}'$ such that $c\,e' \Downarrow \texttt{error}$.

4.4 Properties of the Calculus

We study some notable properties of the calculus. The completeness proof will make use of most of these properties.

4.4.1 Contextual Preorder.

A preorder between terms and a preorder between contexts can be derived from the closure relation. Note that some information is lost in doing so: the closure operator cannot be recovered from the preorder. These preorders are useful to state some of the properties of the calculus.

The *contextual preorder* on terms is defined by: $e \leq e'$ if and only if one of the three equivalent propositions holds:

- $\overline{\{e\}} \subseteq \overline{\{e'\}}$;
- $\{e'\}^{\perp} \subseteq \{e\}^{\perp}$;
- $e \in \overline{\mathcal{E}}$ whenever $e' \in \overline{\mathcal{E}}$.

Likewise, a preorder on contexts is defined by: $c \leq c'$ if and only if one of the three equivalent propositions holds:

- $\{c\}^{\perp} \subseteq \{c'\}^{\perp}$;
- $\overline{\{c'\}} \subseteq \overline{\{c\}}$;
- $c' \in \overline{\mathcal{C}}$ whenever $c \in \overline{\mathcal{C}}$.

Note that we chose to define both preorders so that the ordering between two elements (either two terms or two contexts) derives from the inclusion ordering between the two naturally associated sets of terms. As a consequence, the definition of the ordering between contexts looks the opposite of the definition of the ordering between terms.

4.4.2 Terms and Values

An important property of the calculus is that the behavior of a term (as specified by the closure operator) is characterized by the behavior of the values it reduces to.

LEMMA 2 (TERMS AND VALUES). *A term e is included in a closed set of terms $\overline{\mathcal{E}}$ if and only if any value v it reduces to is included in $\overline{\mathcal{E}}$.*

We will often use the following immediate corollary.

COROLLARY 3 (TRIVIAL APPROXIMATION). *Let e and e' be two terms such that $e \Downarrow v$ whenever $e' \Downarrow v$. Then, $e' \leq e$.*

The contexts has been carefully chosen for the lemma 2 to hold. For instance, it does not hold if the syntax of frames is extended with a family of frames $e\,\_$. Indeed, consider the term:

$$\begin{array}{lcl} f & = & \lambda x.\texttt{if}\ \kappa = x\ \texttt{then} \\ & & \qquad \texttt{if}\ \kappa = x\ \texttt{then diverge else error} \\ & & \quad \texttt{else} \\ & & \qquad \texttt{diverge} \ . \end{array}$$

We have $f\,\kappa' \in \mathcal{S}$ for all constant κ', but $f\,(\kappa \sqcup \kappa') \notin \mathcal{S}$ if the constants κ and κ' are distinct. So, if $Id \circ (f\,\_)$ is a context, then we have $\kappa' \in \{Id \circ (f\,\_)\}^{\perp}$ for all constant κ', but not $\kappa \sqcup \kappa' \in \{Id \circ (f\,\_)\}^{\perp}$ (when the constants κ and κ' are distinct).

Intuitively, the result holds if the evaluation of a term $c\,e$ first involves the evaluation of the term e. We formalize this property by introducing a notion of linearity: we say that a function f from terms to terms is *linear* when for any term e and value v, $f\,e \Downarrow v$ if and only if there exists a value v' such that $e \Downarrow v'$ and $f\,v' \Downarrow v$. Note that linearity implies strictness. We then have the expected result.

LEMMA 4 (CONTEXT LINEARITY). *Contexts are linear.*

A key ingredient to prove this result is a term replacement lemma.

LEMMA 5 (TERM REPLACEMENT). *If $e \Downarrow v$ whenever $e' \Downarrow v$, then for all contexts c, $c\,e \Downarrow v$ whenever $c\,e' \Downarrow v$.*

We can now perform the different proofs.

PROOF SKETCH OF LEMMA 5 (TERM REPLACEMENT). The proof is by induction on contexts. $\square$

PROOF SKETCH OF LEMMA 4 (CONTEXT LINEARITY). By Lemma 5 (Term Replacement), if $e \Downarrow v'$ and $c\,v' \Downarrow v$, then $c\,e \Downarrow v$.

The converse implication is proved by induction on contexts. We consider each possible context:

- Context Id. Clear
- Context $c \circ F$. We assume $(c \circ F)\,e \Downarrow v$, that is, $c\,(F[e]) \Downarrow v$. By

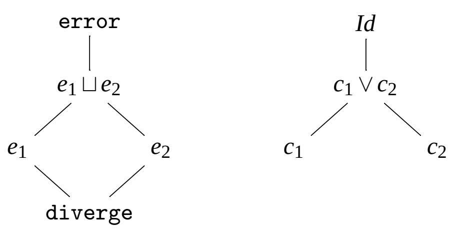


Figure 2. Ordering of Terms and Contexts

induction hypothesis, there exists a value v'' such that $F[e] \Downarrow v''$ and $c\,v'' \Downarrow v$. By case on a derivation of $F[e] \Downarrow v''$, we prove that there exists v' such that $e \Downarrow v'$ and $F[e'] \Downarrow v''$. Then, by Lemma 5 (Term Replacement), $c\,(F[v']) \Downarrow v$, that is, $(c \circ F)\,v' \Downarrow v$ as wanted.

- Context $c \vee c'$. We assume $(c \vee c')\,e \Downarrow v$. By case on a derivation of this relation, we can see that there exists v' such that $e \Downarrow v'$ and $(c \vee c')\,v' \Downarrow v$, as wanted. □

PROOF OF LEMMA 2 (TERMS AND VALUES). It is sufficient to prove that for any context $c \in \mathcal{E}^{\perp}$, $c\,e \in \mathcal{S}$ if and only if any value v it reduces to satisfies $c\,v \in \mathcal{S}$. This can be rephrased into: $c\,e \Downarrow$ `error` if and only if there exists a value v such that $e \Downarrow v$ and $c\,v \Downarrow$ `error`, which is a direct consequence of Lemma 4 (Context Linearity). □

4.4.3 Ordering of Terms and Contexts

We present the relative ordering of some interesting terms and contexts. This ordering is illustrated by Fig. 2.

LEMMA 6 (LEAST UPPER BOUNDS). *For all terms e, e' and for all contexts c, c', we have:*

- $\overline{\{e \sqcup e'\}} = \overline{\{e\}} \boxed{\cup} \overline{\{e'\}}$;
- $\{c \vee c'\}^{\perp} = \{c\}^{\perp} \boxed{\cup} \{c'\}^{\perp}$

As a consequence, the term $e \sqcup e'$ is a least upper bound of the two terms e and e', and the context $c \vee c'$ is a least upper bound of the two contexts c and c'.

PROOF. By Lemma 3 (Trivial Approximation), we have $e \leq e \sqcup e'$ and $e' \leq e \sqcup e'$, that is, $e \sqcup e'$ is an upper bound of e and e'. This statement is equivalent to the inclusion $\overline{\{e \sqcup e'\}} \supseteq \overline{\{e\}} \boxed{\cup} \overline{\{e'\}}$.

Let us now prove that $\overline{\{e \sqcup e'\}} \supseteq \overline{\{e\}} \boxed{\cup} \overline{\{e'\}}$. Notice that $\overline{\{e\}} \boxed{\cup} \overline{\{e'\}} = (\{e\}^{\perp} \cap \{e'\}^{\perp})^{\perp}$. Therefore, it is sufficient to prove that $\{e\}^{\perp} \cap \{e'\}^{\perp} \subseteq \{e \sqcup e'\}^{\perp}$, that is, for all contexts c, if $c\,e \in \mathcal{S}$ and $c\,e' \in \mathcal{S}$, then $c\,(e \sqcup e') \in \mathcal{S}$. The proof is by contradiction. Suppose $c\,(e \sqcup e') \notin \mathcal{S}$, that is, $c\,(e \sqcup e') \Downarrow$ `error`. By Lemma 4 (Context Linearity), there exists a value v such that $e \sqcup e' \Downarrow v$ and $c\,v \Downarrow$ `error`. By inspection of the possible derivations of $e \sqcup e' \Downarrow v$, we see that either $e \Downarrow v$ or $e' \Downarrow v$. Then, by Lemma 4 again, either $c\,e \Downarrow$ `error` or $c\,e' \Downarrow$ `error`, that is, either $c\,e \notin \mathcal{S}$ or $c\,e' \notin \mathcal{S}$, as wanted.

Finally, let us prove that $e \sqcup e'$ is a least upper bound. Let e'' be an upper bound of e and e'. Clearly, $\overline{\{e''\}} \supseteq \overline{\{e\}} \boxed{\cup} \overline{\{e'\}}$. Therefore, $\overline{\{e''\}} \supseteq \overline{\{e \sqcup e'\}}$, that is, $e'' \geq e \sqcup e'$.

We now prove that $c \leq c \vee c'$, that is, $\{c\}^{\perp} \subseteq \{c \vee c'\}^{\perp}$. Let e be a term in $\{c\}^{\perp}$. We have $c\,e \in \mathcal{S}$. Hence, by Lemma 2 (Terms and Values), for all value v such that $e \Downarrow v$, $c\,v \in \mathcal{S}$. Thus, for all value v such that $e \Downarrow v$, $(c \vee c')\,v \in \mathcal{S}$. Indeed, if $(c \vee c')\,v \Downarrow$ `error`, then both $c\,v \Downarrow$ `error` and $c'\,v \Downarrow$ `error`. Finally, by Lemma 2 again, $(c \vee c')\,e \in \mathcal{S}$, that is, $e \in \{c \vee c'\}^{\perp}$ as wanted. The proof that $c' \leq c \vee c'$ is similar. Thus, $c \vee c'$ is an upper bound of c and c'.

The two assertions $c \leq c \vee c'$ and $c' \leq c \vee c'$ are equivalent to the inclusion $\{c\}^{\perp} \boxed{\cup} \{c'\}^{\perp} \subseteq \{c \vee c'\}^{\perp}$.

We now prove that $\{c \vee c'\}^{\perp} \subseteq \{c\}^{\perp} \boxed{\cup} \{c'\}^{\perp}$. By Lemma 2 (Terms and Values), is is sufficient to prove that for all values v, if $v \in \{c \vee c'\}^{\perp}$ then $v \in \{c\}^{\perp} \boxed{\cup} \{c'\}^{\perp}$. We actually prove the contraposition. Suppose $v \notin \{c\}^{\perp} \cup \{c'\}^{\perp} \subseteq \{c\}^{\perp} \boxed{\cup} \{c'\}^{\perp}$. Then, $c\,v \Downarrow$ `error` and $c'\,v \Downarrow$ `error`. Hence, $(c \vee c')\,v \Downarrow$ `error`, that is, $v \notin \{c \vee c'\}^{\perp}$.

Finally, let us prove that $c \vee c'$ is a least upper bound. Let c'' be an upper bound of c and c'. Clearly, $\{c''\}^{\perp} \supseteq \{c\}^{\perp} \boxed{\cup} \{c'\}^{\perp}$. Therefore, $\{c''\}^{\perp} \supseteq \{c \vee c'\}^{\perp}$, that is, $c'' \geq c \vee c'$. □

LEMMA 7 (DIVERGENCE). *The term* `diverge` *is a least term. In particular,* `diverge` $\in \boxed{\perp}$.

PROOF. We first prove that `diverge` $\in \boxed{\perp}$. There is no value v such that `diverge` $\Downarrow v$. Therefore, by Lemma 2 (Terms and Values), `diverge` $\in \overline{\emptyset} \subseteq \boxed{\perp}$.

We then prove that `diverge` is a least term. For all term e, we have $\overline{\{\texttt{diverge}\}} = \overline{\{\emptyset\}} \subseteq \overline{\{e\}}$, so `diverge` $\leq e$. □

The next two lemmas are not used in the completeness proof. We find them interesting nonetheless. They both rely on the following remark.

REMARK 8 (ERROR PROPAGATION). *If $e \Downarrow$* `error`*, then $c\,e \Downarrow$* `error`.

PROOF SKETCH. By induction on contexts. □

LEMMA 9 (ERROR). *The term* `error` *is a largest term.*

PROOF. Let e be a term. By the remark above, for all contexts c, c `error` $\Downarrow$ `error`. Hence, $\{\texttt{error}\}^{\perp} = \emptyset \subseteq \{e\}^{\perp}$. That is, $e \leq$ `error`. □

LEMMA 10 (IDENTITY CONTEXT). *The context Id is a largest context.*

PROOF. Let c be a context. We prove that $\{c\}^{\perp} \subseteq \{Id\}^{\perp}$. Suppose that $e \notin \{Id\}^{\perp}$. We have $Id\,e \Downarrow$ `error`, that is, $e \Downarrow$ `error`. By remark 8, $c\,e \Downarrow$ `error`, that is, $e \notin \{c\}^{\perp}$. □

4.4.4 Sets of Values

We write $\mathcal{V}(\mathcal{E})$ for the set of values contained in a set of terms $\mathcal{E}$:

$$\mathcal{V}(\mathcal{E}) = \{v | v \in \mathcal{E}\} \enspace .$$

A direct consequence of Lemma 2 (Terms and Values) is that a closed set of terms is characterized by its values:

$$\overline{\mathcal{E}} = \overline{\mathcal{V}(\overline{\mathcal{E}})} \enspace .$$

It seems therefore natural to study some of the properties of the sets of values $\mathcal{V}(\overline{\mathcal{E}})$.

LEMMA 11 (LEAST SEMANTIC TYPE). *The least semantic type $\boxed{\bot} = \mathcal{N}$ does not contain any value. As a consequence, it is the least closed set of terms: $\boxed{\bot} = \overline{\emptyset}$.*

PROOF. By definition, $\boxed{\bot} \subseteq \overline{\{\texttt{diverge}\}}$. Besides, the context $Id \circ (\texttt{if}\ \_ = \kappa\ \texttt{then error else error})$ is included in $\{\texttt{diverge}\}^{\perp}$, but is not orthogonal to any value. Therefore, $\boxed{\bot}$ does not contain any value.

Then, $\boxed{\bot} = \overline{\mathcal{V}(\boxed{\bot})} = \overline{\emptyset}$. □

LEMMA 12 (UNION AND VALUES). *The values of the closed union of two closed sets is the union of the values of each closed sets: $\mathcal{V}(\overline{\mathcal{E}} \boxed{\cup} \overline{\mathcal{E}'}) = \mathcal{V}(\overline{\mathcal{E}}) \cup \mathcal{V}(\overline{\mathcal{E}'})$*

PROOF. Clearly, $\mathcal{V}(\overline{\mathcal{E}} \boxed{\cup} \overline{\mathcal{E}'}) \supseteq \mathcal{V}(\overline{\mathcal{E}} \cup \overline{\mathcal{E}'}) = \mathcal{V}(\overline{\mathcal{E}}) \cup \mathcal{V}(\overline{\mathcal{E}'})$. We show the converse inclusion. Let $v \in \mathcal{V}(\overline{\mathcal{E}} \boxed{\cup} \overline{\mathcal{E}'})$. Suppose $v \notin \mathcal{V}(\overline{\mathcal{E}}) \cup \mathcal{V}(\overline{\mathcal{E}'})$. Then, there exists two contexts $c \in \overline{\mathcal{E}}^{\perp}$ and $c' \in \overline{\mathcal{E}'}^{\perp}$ such that $c\,v \Downarrow \texttt{error}$ and $c'\,v \Downarrow \texttt{error}$. Therefore, $(c \vee c')\,v \Downarrow \texttt{error}$. But, by Lemma 6 (Least Upper Bounds), $\overline{\mathcal{E}} \boxed{\cup} \overline{\mathcal{E}'} \subseteq \{c\}^{\perp} \boxed{\cup} \{c'\}^{\perp} = \{c \vee c'\}^{\perp}$, that is, $c \vee c' \in (\overline{\mathcal{E}} \boxed{\cup} \overline{\mathcal{E}'})^{\perp}$. This contradicts the assumption $v \in \mathcal{V}(\overline{\mathcal{E}} \boxed{\cup} \overline{\mathcal{E}'})$. □

LEMMA 13 (PRIME WHEN DIRECTED). *If the set of values $\mathcal{V}(\overline{\mathcal{E}})$ is directed then the set of terms $\overline{\mathcal{E}}$ is* prime, *that is, if $\overline{\mathcal{E}} \subseteq \overline{\mathcal{E}_1} \boxed{\cup} \overline{\mathcal{E}_2}$, then either $\overline{\mathcal{E}} \subseteq \overline{\mathcal{E}_1}$ or $\overline{\mathcal{E}} \subseteq \overline{\mathcal{E}_2}$.*

This lemma is a direct consequence of the next lemma.

LEMMA 14. *Let A be a directed set. Let B and C be two downward closed sets. If $A \subseteq B \cup C$, then either $A \subseteq B$ or $A \subseteq C$.*

PROOF. The proof is by contraposition. We assume $A \not\subseteq B$ and $A \not\subseteq C$. Then, there exists two elements y and z in A such that $y \notin B$ and $z \notin C$. As A is directed, there exists $x \in A$ such that $y \preceq x$ and $z \preceq x$. As both B and C are downward closed, $x \notin B$ and $x \notin C$ (otherwise, either $y \in B$ or $z \in C$). Therefore, $A \not\subseteq B \cup C$ as wanted. □

4.4.5 *Instance of the Class of Calculi*

We have the expected result:

LEMMA 15. *The calculus is in instance of the class of calculi specified in Sect. 3.2.*

PROOF. The functions $e \mapsto e\,e'$ (where $e' \in \mathcal{S}$), `fst` and `snd` are continuous by definition of the closure operator by orthogonality. They are also strict. Indeed, by Lemma 11 (Least Semantic Type), $\boxed{\bot} = \overline{\emptyset}$, and therefore all continuous functions are strict. Finally, we clearly have $\kappa \in \mathcal{S}$ for all constants κ. □

4.4.6 *Orthogonality Functions-Arguments*

Just like we defined an orthogonality relation between terms and contexts in Sect. 4.2, we can define a family of orthogonality relations between functions and arguments.

In the remainder of this section, we assume given a semantic type $\mathcal{E}_0$. We define an orthogonality relation between the elements of $\mathcal{T}$ (all terms), considered as function arguments, and the elements of $\mathcal{S}$ (safe terms), considered as functions: an argument $e' \in \mathcal{T}$ is orthogonal to a function $e \in \mathcal{S}$ when $e\,e' \in \mathcal{E}_0$. From this relation, we define the orthogonal of a set $\mathcal{E}$ of arguments by

$$\mathcal{E}^{\text{fun}} = \{e \in \mathcal{S} \,|\, \forall e' \in \mathcal{E}. e\,e' \in \mathcal{E}_0\} = \mathcal{E} \boxed{\rightarrow} \mathcal{E}_0$$

and the orthogonal of a set $\mathcal{E} \subseteq \mathcal{S}$ of functions by

$$\mathcal{E}^{\text{arg}} = \{e' \,|\, \forall e \in \mathcal{E}. e\,e' \in \mathcal{E}_0\} \ .$$

We have the following diagram.

$$\begin{tikzcd}\mathcal{P}(\mathcal{T}) \arrow[r, bend left, "\text{fun}"] & \mathcal{P}(\mathcal{S}) \arrow[l, bend left, "\text{arg}"]\end{tikzcd}$$

The function $\mathcal{E} \mapsto \mathcal{E}^{\text{fun}\,\text{arg}}$ is a closure on set of arguments.

LEMMA 16 (FUNCTION ORTHOGONALITY). *The closure induced by function orthogonality is strictly finer than the closure induced by context orthogonality: for all set of terms $\mathcal{E}$, we have:*

$$\mathcal{E}^{\text{fun}\,\text{arg}} \subseteq \overline{\mathcal{E}} \ ,$$

but we can find a set of terms $\mathcal{E}$ such that:

$$\overline{\mathcal{E}} \not\subseteq \mathcal{E}^{\text{fun}\,\text{arg}} \ .$$

As a consequence,

$$\overline{\mathcal{E}}^{\text{fun}\,\text{arg}} = \overline{\mathcal{E}^{\text{fun}\,\text{arg}}} = \overline{\mathcal{E}}$$

and

$$\begin{cases} \overline{\mathcal{E}}^{\text{fun}} & = & \overline{\mathcal{E}} \boxed{\rightarrow} \mathcal{E}_0 \\ \overline{\mathcal{E}} & = & (\overline{\mathcal{E}} \boxed{\rightarrow} \mathcal{E}_0)^{\text{arg}} \end{cases} .$$

The key idea to prove the first inclusion is to show that for each context c there is a function $\langle c \rangle$ that behaves "similarly". This function is defined as follows.

$$\langle c \rangle = \lambda x.\texttt{let}\ y = c\,x\ \texttt{in diverge}$$

It satisfies the following property.

LEMMA 17 (CONTEXT AS FUNCTION). *For any set of terms $\mathcal{E}$ and any context c, we have $c \in \mathcal{E}^{\perp}$ if and only if $\langle c \rangle \in \mathcal{E} \boxed{\rightarrow} \mathcal{E}_0$.*

PROOF. Suppose $c \notin \mathcal{E}^{\perp}$. There exists $e \in \mathcal{E}$ such that $c\,e \Downarrow \texttt{error}$. It is easy to check that, then, $\langle c \rangle\, e \Downarrow \texttt{error}$. Therefore, $\langle c \rangle \notin \mathcal{E} \boxed{\rightarrow} \mathcal{E}_0$.

Conversely, suppose that $c \in \mathcal{E}^{\perp}$. Let $e \in \mathcal{E}$. We have $\neg(c\,e \Downarrow \texttt{error})$. Then, $\langle c \rangle\, e$ reduces to the same values as `diverge`. Therefore, by Lemmas 2 (Terms and Values), $\langle c \rangle\, e \in \overline{\emptyset} \subseteq \mathcal{E}_0$. Hence, $\langle c \rangle \in \mathcal{E} \boxed{\rightarrow} \mathcal{E}_0$. □

PROOF OF LEMMA 16 (FUNCTION ORTHOGONALITY).
We prove that $\mathcal{E}^{\text{fun}\,\text{arg}} \subseteq \overline{\mathcal{E}}$. It is sufficient to prove that, for any term $e \in \mathcal{E}^{\text{fun}\,\text{arg}}$ and for any context $c \in \mathcal{E}^{\perp}$, we have $e \perp c$. So, let e and c be such a term and a context. By lemma 17 (Context as Function), $\langle c \rangle \in \mathcal{E} \boxed{\rightarrow} \mathcal{E}_0 = \mathcal{E}^{\text{fun}}$. Therefore, the function $\langle c \rangle$ is orthogonal to the argument $e \in \mathcal{E}^{\text{fun}\,\text{arg}}$. In other words, $\langle c \rangle \in \{e\}^{\text{fun}} = \{e\} \boxed{\rightarrow} \mathcal{E}_0$. By lemma 17 again, $c \in \{e\}^{\perp}$, that is, $e \perp c$ as wanted.

We prove that there exists a set of terms $\mathcal{E}$ such that $\overline{\mathcal{E}} \not\subseteq \mathcal{E}^{\text{fun}\,\text{arg}}$. We take $\mathcal{E} = \{\kappa, \kappa'\}$ where κ and κ' are two distinct constants. By

Lemma 2 (Terms and Values), $\kappa \sqcup \kappa' \in \overline{\mathcal{E}}$. Now, consider the following term:

```
f   =   λx.if κ = x then
                if κ = x then diverge else error
            else
                diverge.
```

By Lemma 2 (Terms and Values), we have $f\,\kappa'' \in \mathcal{E}_0$ for all constant κ'' (the application diverges). Hence, $f \in \mathcal{E}^{\text{fun}}$. But $f\,(\kappa \sqcup \kappa') \Downarrow$ `error`. Therefore, $\kappa \sqcup \kappa' \notin \mathcal{E}^{\text{fun}\,\text{arg}}$.

We prove $\overline{\mathcal{E}}^{\text{fun}\,\text{arg}} = \overline{\mathcal{E}^{\text{fun}\,\text{arg}}} = \overline{\mathcal{E}}$. We have $\overline{\mathcal{E}} \subseteq \overline{\mathcal{E}^{\text{fun}\,\text{arg}}}$ and $\overline{\mathcal{E}} \subseteq \overline{\mathcal{E}}^{\text{fun}\,\text{arg}}$ by extensivity and monotony of the closure operators. The converse inclusions can be derived from the inclusion $\mathcal{E}^{\text{fun}\,\text{arg}} \subseteq \overline{\mathcal{E}}$ by idempotence of the closure operators.

Finally, we have $\overline{\mathcal{E}}^{\text{fun}} = \overline{\mathcal{E}} \boxed{\rightarrow} \mathcal{E}_0$ by definition. Hence, $\overline{\mathcal{E}} = \overline{\mathcal{E}}^{\text{fun}\,\text{arg}} = (\overline{\mathcal{E}} \boxed{\rightarrow} \mathcal{E}_0)^{\text{arg}}$. □

5 A Simple Type System

We present a simple type system and prove its soundness and completeness. These properties have been mechanically checked using the Coq proof assistant [1].

5.1 Types

The syntax of types is given by the following grammar.

τ	$::=$	χ	constructed type
		$\bot$	bottom type
		$\tau \cup \tau$	union type
χ	$::=$	$\tau \rightarrow \tau$	function type
		$\tau \times \tau$	pair type
		κ	constant type

The semantics $[\![\tau]\!]$ of a type τ is defined inductively on the syntax of types in a straightforward manner:

$$\begin{array}{lcl}
[\![\tau \rightarrow \tau']\!] & = & [\![\tau]\!] \boxed{\rightarrow} [\![\tau']\!] \\
[\![\tau \times \tau']\!] & = & [\![\tau]\!] \boxed{\times} [\![\tau']\!] \\
[\![\kappa]\!] & = & \boxed{\kappa} \\
[\![\bot]\!] & = & \boxed{\bot} \\
[\![\tau \cup \tau']\!] & = & [\![\tau]\!] \boxed{\cup} [\![\tau']\!]
\end{array}$$

Clearly, the semantics $[\![\tau]\!]$ of a *syntactic type* τ is a semantic type.

5.2 Subtyping Relation

The subtyping relation <: is defined inductively. The subtyping rules are given in Fig. 3 . Note that the rules are almost syntax-directed: the conclusion of the rules are syntactically disjoint, except in the case of the rules UNION-RIGHT-1 and UNION-RIGHT-2.

5.3 Soundness of the Subtyping Relation

The soundness of the subtyping relation is straightforward.

THEOREM 18 (SOUNDNESS). *If $\tau <: \tau'$, then $[\![\tau]\!] \subseteq [\![\tau']\!]$.*

PROOF. By induction on a derivation of $\tau <: \tau'$.

$$\textsc{Function}\ \frac{\tau_1 <: \tau_1' \qquad \tau_2' <: \tau_2}{\tau_2 \rightarrow \tau_1 <: \tau_2' \rightarrow \tau_1'} \qquad \textsc{Pair}\ \frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'} \qquad \textsc{Constant}\ \kappa <: \kappa$$

$$\textsc{Bottom}\ \bot <: \tau \qquad \textsc{Union-Left}\ \frac{\tau <: \tau'' \qquad \tau' <: \tau''}{\tau \cup \tau' <: \tau''} \qquad \textsc{Union-Right-1}\ \frac{\chi <: \tau}{\chi <: \tau \cup \tau'}$$

$$\textsc{Union-Right-2}\ \frac{\chi <: \tau'}{\chi <: \tau \cup \tau'}$$

Figure 3. Subtyping Rules (Simple Types)

- Rule FUNCTION: by covariance and contravariance of the operation $\boxed{\rightarrow}$.
- Rule PAIR: by covariance of the operation $\boxed{\times}$.
- Rule CONSTANT: immediate.
- Rule BOTTOM: the semantic type $\boxed{\bot}$ is the least semantic type.
- Rule UNION-LEFT: by induction hypothesis,
$$[\![\tau]\!] \cup [\![\tau']\!] \subseteq [\![\tau'']\!] \ ;$$
hence, as $[\![\tau'']\!]$ is closed,
$$[\![\tau]\!] \boxed{\cup} [\![\tau']\!] = \overline{[\![\tau]\!] \cup [\![\tau']\!]} \subseteq [\![\tau'']\!] \ .$$
- Rule UNION-RIGHT-1: $[\![\tau]\!] \subseteq [\![\tau]\!] \boxed{\cup} [\![\tau']\!]$.
- Rule UNION-RIGHT-2: $[\![\tau']\!] \subseteq [\![\tau]\!] \boxed{\cup} [\![\tau']\!]$. □

5.4 Properties of Constructed Types

Before proving the completeness of the subtyping relation <:, we first state some interesting properties of the semantics of constructed types.

LEMMA 19 (HOMOGENEITY). *The set of values $\mathcal{V}([\![\chi]\!])$ of a constructed type χ is homogeneous: $\mathcal{V}([\![\tau' \rightarrow \tau]\!])$ only contain functions, $\mathcal{V}([\![\tau \times \tau']\!])$ only contain pairs, $\mathcal{V}([\![\kappa]\!])$ only contain the constant κ.*

PROOF. We consider a value v in $[\![\chi]\!]$ and prove that it has the expected shape. We consider each kind of constructed type in turn.

- Case $\chi = \tau' \rightarrow \tau$. We have $v\,$`diverge` $\in [\![\tau]\!] \subseteq \mathcal{S}$. Therefore, the relation $v\,$`diverge` $\Downarrow$ `error` does not hold. By rule APP-ERROR, we can see that this implies that v is a functional value.
- Case $\chi = \tau \times \tau'$. We have `fst`$\,v \in [\![\tau]\!] \subseteq \mathcal{S}$. Therefore, the relation `fst`$\,v \Downarrow$ `error` does not hold. By rule FST-ERROR, we can see that this implies that v is a pair.
- Case $\chi = \kappa$. We have $v \in [\![\kappa]\!] = \overline{\{\kappa\}}$. Consider the context $c = Id \circ ($`if` $\_ = \kappa$ `then diverge else error`$)$. Clearly, $c\,\kappa \in \mathcal{S}$. Hence, $c \in \{\kappa\}^{\perp}$, so $c\,v \in \mathcal{S}$. By rules IF-NOT-EQUAL and IF-ERROR, we can see that this implies that $v = \kappa$. □

LEMMA 20 (DIRECTED SET). *The set of values $\mathcal{V}([\![\chi]\!])$ of a constructed type χ is directed.*

PROOF. We first prove that $\mathcal{V}([\![\chi]\!])$ is not empty by case on χ.

- Case $\chi = \tau' \to \tau$. The value $\lambda x.\mathtt{diverge}$ is safe and for all terms e, $(\lambda x.\mathtt{diverge})\,e$ diverges. Hence,
$$\lambda x.\mathtt{diverge} \in \mathcal{V}([\![\tau' \to \tau]\!]) \ .$$

- Case $\chi = \tau \times \tau'$. The value $(\mathtt{diverge}, \mathtt{diverge})$ is safe and both the terms
$$\mathtt{fst}\,(\mathtt{diverge}, \mathtt{diverge})$$
and
$$\mathtt{snd}\,(\mathtt{diverge}, \mathtt{diverge})$$
diverge. Therefore, the value $(\mathtt{diverge}, \mathtt{diverge})$ is included in $\mathcal{V}([\![\tau \times \tau']\!])$.

- Case $\chi = \kappa$. By definition, $[\![\kappa]\!] = \boxed{\kappa} = \overline{\{\kappa\}}$. Hence, $\kappa \in \mathcal{V}([\![\kappa]\!])$.

We then prove that if two values v_1 and v_2 are included in $\mathcal{V}([\![\chi]\!])$, then they have an upper bound in this set.

- Case $\chi = \tau' \to \tau$. By Lemma 19 (Homogeneity), the values are of the shape $\lambda x.e_1$ and $\lambda x.e_2$. Let us show that $\lambda x.(e_1 \sqcup e_2)$ is an upper bound of these values in $\mathcal{V}([\![\tau' \to \tau]\!])$.

  We first show that this term is included in the set. First, it is safe. Then, let e be a term in $[\![\tau']\!]$. It is clear that, if $(\lambda x.(e_1 \sqcup e_2))\,e \Downarrow v$, then either $(\lambda x.e_1)\,e \Downarrow v$ or $(\lambda x.e_2)\,e \Downarrow v$. Therefore, by Lemma 2 (Terms and Values), as $(\lambda x.e_1)\,e$ and $(\lambda x.e_2)\,e$ are in $[\![\tau]\!]$, $(\lambda x.(e_1 \sqcup e_2))\,e$ is also in $[\![\tau]\!]$, as wanted.

  We now show that the term is an upper bound. We prove that $\lambda x.e_1 \leq \lambda x.(e_1 \sqcup e_2)$. The proof of the other relation $\lambda x.e_2 \leq \lambda x.(e_1 \sqcup e_2)$ is similar. It is sufficient to prove that for all contexts c, if $c\,(\lambda x.e_1) \Downarrow \mathtt{error}$, then $c\,(\lambda x.(e_1 \sqcup e_2)) \Downarrow \mathtt{error}$. The proof is by induction on the context c.

  – Case $c = Id$. This is clear as neither term reduces to $\mathtt{error}$.

  – Case $c = c' \circ \_e$. We have $c'\,((\lambda x.e_1)\,e) \Downarrow \mathtt{error}$. Hence, by Lemma 5 (Term Replacement), we have $c'\,((\lambda x.(e_1 \sqcup e_2))\,e) \Downarrow \mathtt{error}$ as wanted.

  – Case $c = c' \circ F$ where F is not of the form $\_e$. Both terms reduce to $\mathtt{error}$.

  – Case $c = c_1 \vee c_2$. We remark that for any value v, $(c_1 \vee c_2)\,v \Downarrow \mathtt{error}$ if and only if either $c_1\,v \Downarrow \mathtt{error}$ or $c_2\,v \Downarrow \mathtt{error}$. Therefore, the assumption $c\,(\lambda x.e_1) \Downarrow \mathtt{error}$ implies that either $c_1\,(\lambda x.e_1) \Downarrow \mathtt{error}$ or $c_2\,(\lambda x.e_1) \Downarrow \mathtt{error}$. In both cases, we can conclude by induction hypothesis and another use of the remark.

- Case $\chi = \tau \times \tau'$.

  By Lemma 19 (Homogeneity), the values are of the shape (e_1, e'_1) and (e_2, e'_2). Let us show that $((e_1 \sqcup e_2), (e'_1 \sqcup e'_2))$ is an upper bound of these values in $\mathcal{V}([\![\tau \times \tau']\!])$.

  We first show that this term is included in the set. First, it is safe. Then, it is clear that, if $\mathtt{fst}\,((e_1 \sqcup e_2), (e'_1 \sqcup e'_2)) \Downarrow v$, then either $\mathtt{fst}\,(e_1, e'_1) \Downarrow v$ or $\mathtt{fst}\,(e_2, e'_2) \Downarrow v$. Therefore, by Lemma 2 (Terms and Values), as the terms $\mathtt{fst}\,(e_1, e'_1)$ and $\mathtt{fst}\,(e_2, e'_2)$ are in $[\![\tau]\!]$, the term $\mathtt{fst}\,((e_1 \sqcup e_2), (e'_1 \sqcup e'_2))$ is also in $[\![\tau]\!]$, as wanted. Similarly, the term $\mathtt{snd}\,((e_1 \sqcup e_2), (e'_1 \sqcup e'_2))$ is in $[\![\tau']\!]$.

  We now show that the term is an upper bound. We prove that $(e_1, e'_1) \leq ((e_1 \sqcup e_2), (e'_1 \sqcup e'_2))$. The proof of the other relation $(e_2, e'_2) \leq ((e_1 \sqcup e_2), (e'_1 \sqcup e'_2))$ is similar. It is sufficient to prove that for all contexts c, if $c\,(e_1, e'_1) \Downarrow \mathtt{error}$, then $c\,((e_1 \sqcup e_2), (e'_1 \sqcup e'_2)) \Downarrow \mathtt{error}$. The proof is by induction on the context c.

  – Case $c = Id$. This is clear as neither term reduces to $\mathtt{error}$.

  – Case $c = c' \circ \mathtt{fst}\,\_$. We have $c'\,(\mathtt{fst}\,(e_1, e'_1)) \Downarrow \mathtt{error}$. Hence, by Lemma 5 (Term Replacement), we have
  $$c'\,(\mathtt{fst}\,((e_1 \sqcup e_2), (e'_1 \sqcup e'_2))) \Downarrow \mathtt{error}$$
  as wanted.

  – Case $c = c' \circ \mathtt{snd}\,\_$. This case is similar to the previous one.

  – Case $c = c' \circ F$ where F is not of one of the forms above. Both terms reduce to $\mathtt{error}$.

  – Case $c = c_1 \vee c_2$. We remark that for any value v, $(c_1 \vee c_2)\,v \Downarrow \mathtt{error}$ if and only if either $c_1\,v \Downarrow \mathtt{error}$ or $c_2\,v \Downarrow \mathtt{error}$. Therefore, the assumption $c\,(e_1, e'_1) \Downarrow \mathtt{error}$ implies that either $c_1\,(e_1, e'_1) \Downarrow \mathtt{error}$ or $c_2\,(e_1, e'_1) \Downarrow \mathtt{error}$. In both cases, we can conclude by induction hypothesis and another use of the remark.

- Case $\chi = \kappa$. The set $\mathcal{V}([\![\kappa]\!])$ is a singleton $\{\kappa\}$. Any pair of values κ and κ from this set has an obvious upper bound κ. □

These two lemmas are illustrated by Fig. 4, respectively for function types, pair types and constant types. In this figure, values are underlined. The value just above `diverge` is included in all constructed types of the corresponding kind. Given two values, one of their upper bound is given.

5.5 Completeness of the Subtyping Relation

We now have all the elements to prove the completeness of the subtyping relation.

THEOREM 21 (COMPLETENESS). *If $[\![\tau]\!] \subseteq [\![\tau']\!]$ for all calculi, then $\tau <: \tau'$.*

At several times in the proof of completeness, we need to prove an inclusion $[\![\tau_1]\!] \subseteq [\![\tau'_1]\!]$ assuming that an inclusion between the semantics of two types built from τ_1 and τ'_1 (for instance, $[\![\tau_1 \times \tau_2]\!] \subseteq [\![\tau'_1 \times \tau'_2]\!]$) holds. The proof is similar in each case. Let us call *typed transformation* a pair of a function F from types to types and a function f from terms to terms such that, for all type τ and all term e, $e \in [\![\tau]\!]$ if and only if $f(e) \in [\![F(\tau)]\!]$. Then, it is easy to see that, if (F, f) is a typed transformation and $[\![F(\tau)]\!] \subseteq [\![F(\tau')]\!]$, then

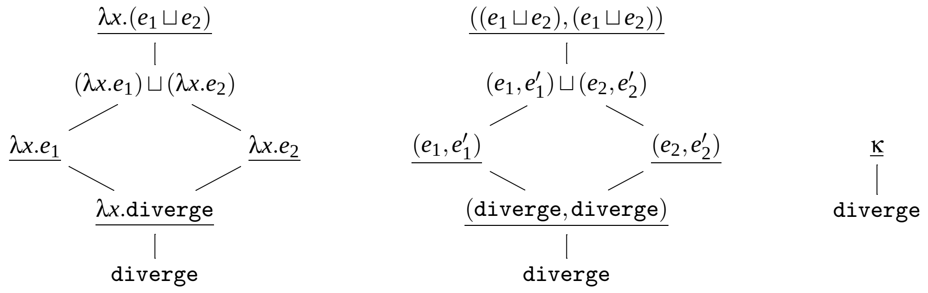

$\lambda x.(e_1 \sqcup e_2)$
$(\lambda x.e_1) \sqcup (\lambda x.e_2)$
$\lambda x.e_1$
$\lambda x.e_2$
$\lambda x.$diverge
diverge
$((e_1 \sqcup e_2),(e_1 \sqcup e_2))$
$(e_1,e_1') \sqcup (e_2,e_2')$
(e_1,e_1')
(e_2,e_2')
(diverge,diverge)
diverge
κ
diverge

Figure 4. Ordering of Terms in a Constructed Type

$\llbracket\tau\rrbracket \subseteq \llbracket\tau'\rrbracket$, as illustrated by the diagram below:

$$\begin{array}{ccccccc} \llbracket\tau\rrbracket & \xmapsto{F} & \llbracket F(\tau)\rrbracket & \subseteq & \llbracket F(\tau')\rrbracket & \xleftarrow{F} & \llbracket\tau'\rrbracket \\ \rotatebox{90}{$\in$} & \Longleftrightarrow & \rotatebox{90}{$\in$} & \Longrightarrow & \rotatebox{90}{$\in$} & \Longleftrightarrow & \rotatebox{90}{$\in$} \\ e & & f(e) & & f(e) & & e \end{array}$$

We thus define three families of typed transformations.

LEMMA 22 (TYPED TRANSFORMATIONS). *The following families of pairs of functions are typed transformations (for the calculus of Sect. 4).*

$$\begin{cases} F_1(\tau') : \tau \mapsto \tau \times \tau' \\ f_1 : e \mapsto (e, \texttt{diverge}) \end{cases}$$

$$\begin{cases} F_2(\tau') : \tau \mapsto \tau' \times \tau \\ f_2 : e \mapsto (\texttt{diverge}, e) \end{cases}$$

$$\begin{cases} F_3(\tau') : \tau \mapsto \tau' \to \tau \\ f_3 : e \mapsto \lambda x.e \end{cases}$$

PROOF. We only prove the result for the first and the last families. The case of the second family is similar.

- We first notice that, by Lemma 3 (Trivial Approximation), $e \in \overline{\mathcal{E}}$ if and only if $\texttt{fst}\,(e,e') \in \overline{\mathcal{E}}$. Likewise, $e' \in \overline{\mathcal{E}}$ if and only if $\texttt{snd}\,(e,e') \in \overline{\mathcal{E}}$.

  Then, let τ be a type and e be a term in $\llbracket\tau\rrbracket$. By Lemma 7 (Divergence), we have $\texttt{diverge} \in \llbracket\tau'\rrbracket$ for all types τ'. Besides, it is clear that $(e,\texttt{diverge})$ is safe. Hence, $f_1(e) = (e,\texttt{diverge}) \in \llbracket\tau \times \tau'\rrbracket = \llbracket F_1(\tau')(\tau)\rrbracket$.

  Conversely, if $f_1(e) \in \llbracket F_1(\tau')(\tau)\rrbracket$, then $(e,\texttt{diverge}) \in \llbracket\tau \times \tau'\rrbracket$, and therefore $e \in \llbracket\tau\rrbracket$.

- Let τ and τ' be two types and e be a term. By Lemma 3 (Trivial Approximation), for all terms e', $e \in \llbracket\tau\rrbracket$ if and only if $(\lambda x.e)\,e' \in \llbracket\tau\rrbracket$. Besides, the term $\lambda x.e$ is safe.

  Thus, if $e \in \llbracket\tau\rrbracket$, then for all $e' \in \llbracket\tau'\rrbracket$, $(\lambda x.e)\,e' \in \llbracket\tau\rrbracket$, and therefore $\lambda x.e \in \llbracket\tau' \to \tau\rrbracket = \llbracket F(\tau')(\tau)\rrbracket$.

  Conversely, if $\lambda x.e \in \llbracket\tau' \to \tau\rrbracket = \llbracket F(\tau')(\tau)\rrbracket$, then we have $(\lambda x.e)\,\texttt{diverge} \in \llbracket\tau\rrbracket$, and therefore $e \in \llbracket\tau\rrbracket$. □

PROOF OF THEOREM 21 (COMPLETENESS). We interpret the semantics of types in the calculus defined in Sect. 4.

In order to handle the contravariance of the function type, we simultaneously prove by induction on τ and τ' that if $\llbracket\tau\rrbracket \subseteq \llbracket\tau'\rrbracket$ then $\tau <: \tau'$, and if $\llbracket\tau'\rrbracket \subseteq \llbracket\tau\rrbracket$ then $\tau' <: \tau$.

For each pair of type τ and τ', we prove that if $\llbracket\tau\rrbracket \subseteq \llbracket\tau'\rrbracket$, then there exists a subtyping rule whose conclusion is $\tau <: \tau'$ and whose premises are a consequence of the induction hypothesis.

- Case $\llbracket\bot\rrbracket \subseteq \llbracket\tau\rrbracket$. By rule BOTTOM, we have $\bot <: \tau$.

- Case $\llbracket\tau \cup \tau'\rrbracket \subseteq \llbracket\tau''\rrbracket$. This implies $\llbracket\tau\rrbracket \subseteq \llbracket\tau''\rrbracket$ and $\llbracket\tau'\rrbracket \subseteq \llbracket\tau''\rrbracket$. Hence, by induction hypothesis, $\tau <: \tau''$ and $\tau' <: \tau''$. Finally, by rule UNION-LEFT, $\tau \cup \tau' <: \tau''$.

- Case $\llbracket\chi\rrbracket \subseteq \llbracket\bot\rrbracket$. By lemma 11 (Least Semantic Type), the set $\llbracket\bot\rrbracket$ does not contain any value. By Lemma 20 (Directed Set), $\llbracket\chi\rrbracket$ contains at least one value(the set $\mathcal{V}(\llbracket\chi\rrbracket)$ is directed). Thus, this case is not possible.

- Case $\llbracket\chi\rrbracket \subseteq \llbracket\tau \cup \tau'\rrbracket$. This is a direct corollary of Lemmas 20 (Directed Set) and 13 (Prime when Directed).

- Case $\llbracket\chi\rrbracket \subseteq \llbracket\chi'\rrbracket$ where χ and χ' are distinct constructed types. By Lemmas 20 (Directed Set) and 19 (Homogeneity), constructed types all contain at least a value(the set $\mathcal{V}(\llbracket\chi\rrbracket)$ is directed), and their values are homogeneous. Hence, $\llbracket\chi\rrbracket$ contains a value which is not in $\llbracket\chi'\rrbracket$. This case is not possible.

- Case $\llbracket\tau_2 \to \tau_1\rrbracket \subseteq \llbracket\tau_4 \to \tau_3\rrbracket$. We prove that $\llbracket\tau_1\rrbracket \subseteq \llbracket\tau_3\rrbracket$ and $\llbracket\tau_4\rrbracket \subseteq \llbracket\tau_2\rrbracket$. This allow us to conclude by induction hypothesis and rule FUNCTION.

  The inclusion $\llbracket\tau_1\rrbracket \subseteq \llbracket\tau_3\rrbracket$ is a direct consequence of Lemma 22 (Typed Transformations).

  Let us prove that $\llbracket\tau_4\rrbracket \subseteq \llbracket\tau_2\rrbracket$. It is sufficient to show that $\llbracket\tau_2\rrbracket^\perp \subseteq \llbracket\tau_4\rrbracket^\perp$. Let c in $\llbracket\tau_2\rrbracket^\perp$. By Lemma 17 (Context as Function), $\langle c\rangle \in \llbracket\tau_2\rrbracket \boxed{\Rightarrow} \llbracket\tau_1\rrbracket = \llbracket\tau_2 \to \tau_1\rrbracket \subseteq \llbracket\tau_4 \to \tau_3\rrbracket = \llbracket\tau_4\rrbracket \boxed{\Rightarrow} \llbracket\tau_3\rrbracket$. Hence, by this lemma again, $c \in \llbracket\tau_4\rrbracket^\perp$as wanted.

- Case $\llbracket\tau_1 \times \tau_2\rrbracket \subseteq \llbracket\tau_3 \times \tau_4\rrbracket$. By Lemma 22 (Typed Transformations), $\llbracket\tau_1\rrbracket \subseteq \llbracket\tau_3\rrbracket$ and $\llbracket\tau_2\rrbracket \subseteq \llbracket\tau_4\rrbracket$. We conclude by induction hypothesisand rule PAIR. □

The proof of the completeness theorem actually leaded us to use an orthogonality relation to define types. Indeed, for completeness to hold, we must have that, if $\tau_1 \to \tau <: \tau_2 \to \tau$, then $\tau_2 <: \tau_1$. This means that, if a term e has type τ_2 but not type τ_1, then there must exist a function e' of type $\tau_1 \to \tau$ but not $\tau_2 \to \tau$. Given that the term

e has type τ_2, a natural way to prove that the function e' does not have type $\tau_2 \to \tau$ is to show that the term $e'\,e$ does not have type τ. So, now, for any term e of type τ_2 but not τ_1, we must be able to find a function of type $\tau_1 \to \tau$ such that the term $e'\,e$ does not have type τ. This must hold for any type τ_2, so the assumption that the term e has type τ_2 does not really put any constraint on the term e and it is natural to drop it. So, finally, we would like that if a term e does not have type τ_1, then there is a function e' of type $\tau_1 \to \tau$ such that $e'\,e$ does not have type τ. In other words, if $e \notin [\![\tau_1]\!]$, then there exists a function $e' \in [\![\tau_1]\!]^{\text{fun}}$ such that e and e' are not orthogonal. That is, if a term is orthogonal to all functions in $[\![\tau_1]\!]^{\text{fun}}$, then it should have type $[\![\tau_1]\!]$: the set $[\![\tau_1]\!]$ must be closed.

A noteworthy point in this discussion is that if τ is not a subtype of τ', then it is unsafe to apply a function accepting terms of type τ' to a term of type τ.

LEMMA 23. *For the calculus of Sect. 4, if τ is not a subtype of τ', then there exists a term e in $[\![\tau]\!]$ and a function e' in $[\![\tau' \to \bot]\!]$ such that $e'\,e \Downarrow$ `error`.*

PROOF. By completeness, $[\![\tau]\!] \not\subseteq [\![\tau']\!]$. Therefore, there exists a term $e \in [\![\tau]\!]$ such that $e \notin [\![\tau']\!]$. By Lemma 16 (Function Orthogonality), $e \notin [\![\tau' \to \bot]\!]^{\text{fun}}$. Therefore, there exists $e' \in [\![\tau' \to \bot]\!]$ such that $e'\,e \Downarrow$ `error`. $\square$

5.6 Typing Rules

We sketch how typing rules can be defined. This cannot be done in a generic way, as the handling of the environment depends on some additional assumptions on the calculus.

Some rules are a direct consequence of the definition of types.

$$\frac{e:\tau' \to \tau \qquad e':\tau'}{e\,e':\tau} \qquad \frac{e:\tau\times\tau'}{\texttt{fst}\,e:\tau} \qquad \frac{e:\tau\times\tau'}{\texttt{snd}\,e:\tau'}$$

$$\kappa:\kappa \qquad \frac{e:\tau}{e:\tau\cup\tau'}$$

The soundness of the subtyping relation yield the following typing rule.

$$\frac{e:\tau' \qquad \tau' <: \tau}{e:\tau}$$

Given a notion of substitution on terms, environments can be interpreted as follows. A judgment $x:\tau' \vdash e:\tau$ holds if and only if for all terms e' in $[\![\tau']\!]$ we have $e[e'/x] \in [\![\tau]\!]$. This immediately generalizes to environments with more than one binding. Then, if, for instance,

$$(e\,e')[e''/x] = (e[e''/x])\,(e'[e''/x]) \ ,$$

we have:

$$\frac{x:\tau'' \vdash e:\tau' \to \tau \qquad x:\tau'' \vdash e':\tau'}{x:\tau'' \vdash e\,e':\tau}$$

Similar properties of the substitution are needed to generalize the other rules with environments. For the abstraction, we need that $\lambda x.e \in [\![\tau' \to \tau]\!]$ if, for all terms e' in $[\![\tau']\!]$, the term $e[e'/x]$ is included in $[\![\tau]\!]$. Then,

$$\frac{x:\tau' \vdash e:\tau}{\lambda x.e:\tau' \to \tau}$$

Some rules do not always hold. For instance, the two following rules hold in the calculus of Sect. 4 due to the lemma 2 (Terms and Values).

$$\frac{e:\tau \qquad e':\tau}{e \sqcup e':\tau} \qquad \frac{e:\tau \qquad e':\tau}{e \vee e':\tau}$$

They would not hold if we had made a different choice of closure operator (see the remark about the choice of the contexts in Sect. 4.4.2).

Another problematic rule is the elimination rule for union. For the calculus of Sect. 4, the following rule holds:

$$\frac{e:\tau\cup\tau' \qquad x:\tau \vdash c\,x:\tau'' \qquad x:\tau' \vdash c\,x:\tau''}{c\,e:\tau''}$$

It can be generalized slightly by replacing the context c by any continuous function from term to term. But the rule below does not hold for arbitrary terms e'.

$$\frac{e:\tau\cup\tau' \qquad x:\tau \vdash e'\,x:\tau'' \qquad x:\tau' \vdash e'\,x:\tau''}{e'\,e:\tau''}$$

Indeed, if we take the function f of Sect. 4.4.2, we have $x:\kappa \vdash f\,x:\bot$ and $x:\kappa' \vdash f\,x:\bot$, but not $f\,(\kappa \sqcup \kappa'):\bot$. We need to make a choice between this last rule and the rules for the choice operators: for this rule to hold, the term $\kappa \sqcup \kappa'$ should not have type $\kappa \cup \kappa'$. The interpretation of terms was chosen so that the second set of typing rules hold, but the other choice arises naturally when considering a larger set of contexts.

6 A Richer Type System

We extend the previous type system with ML-style parametric type constructors and head polymorphism. The reason for restricting ourselves to head polymorphism is that, then, we don't need to consider the subtyping problem between polymorphic types, but only between types with free type variables. Additionally, type constructors and type variables can be handled in a uniform way. Formally, we distinguish types τ and type schemes σ. Types do not contain any quantification. Types schemes are given by the grammar below.

σ	$::=$	τ	type
		$\forall\alpha.\sigma$	polymorphic quantification

Polymorphism is handled using two typing rules involving type schemes, one for instantiation and one for generalization. Subtyping only involves types.

$$\textsc{Gen}\ \frac{\Gamma;\alpha \vdash e:\sigma}{\Gamma \vdash e:\forall\alpha.\sigma} \qquad \textsc{Inst}\ \frac{\Gamma \vdash e:\forall\alpha.\sigma}{\Gamma \vdash e:\sigma[\tau/\alpha]} \qquad \textsc{Sub}\ \frac{\Gamma \vdash e:\tau' \qquad \tau' <: \tau}{\Gamma \vdash e:\tau}$$

Then we consider that a subtyping assertion $\tau <: \tau'$ is valid if and only if it is valid for all possible interpretations of the type variables and the constructors occurring in τ and τ'.

6.1 Types

The syntax of types is given by the following grammar.

τ	$::=$	$\bot$	bottom type
		$\tau\cup\tau$	union type
		χ	constructed type
χ	$::=$	c	type constructor
		$\chi\,\tau$	type application

We use kinds to ensure that type constructors are applied to the right number of arguments. Kinds also indicate the variance of the parameters of the type constructors. To each type constructor c we associate a kind $K(c)$ called the *signature* of c. Kinds are defined by the following grammar.

$$\begin{array}{lcll} k & ::= & * & \\ & & \varepsilon \to k & \\ \varepsilon & ::= & \oplus & \text{covariant argument} \\ & & \ominus & \text{contravariant argument} \end{array}$$

In the following, we only consider *well-kinded* types, that is types τ which satisfy the assertion $\tau : *$ defined inductively below.

$$\textsc{Kind-Constr}\quad c : K(c) \qquad\qquad \textsc{Kind-App}\quad \frac{\chi : \varepsilon \to k \qquad \tau : *}{\chi\tau : k}$$

$$\textsc{Kind-Bottom}\quad \bot : * \qquad\qquad \textsc{Kind-Union}\quad \frac{\tau : * \qquad \tau' : *}{\tau \cup \tau' : *}$$

We extend the signature function K to (well-kinded) constructed types χ by $K(\chi) = k$ when $\chi : k$.

Note that there is not specific construction for function and pair types. Instead, they can be encoded by distinguishing two type constructors $\_ \to \_$ and $\_ \times \_$ of signature respectively $\ominus \to \oplus \to *$ and $\oplus \to \oplus \to *$. Likewise a constant type κ can be encoded as a constructor κ of signature $*$.

6.2 Semantics of Types

The semantics $[\![\tau]\!]_\rho$ of a type τ is parameterized over an environment ρ which provides the semantics of the type constructors occuring in τ. The interpretation $\rho(c)$ of a type constructor c depends on its kind $K(c)$: if $K(c) = *$, then $\rho(c)$ should be a set of terms; if $K(c) = \varepsilon \to k$, then $\rho(c)$ should be a function. We therefore define the semantics $[\![k]\!]$ of kinds and will demand that $\rho(c) \in [\![K(c)]\!]$. The semantics of a kind is an ordered set defined inductively as follows:

$$\begin{array}{lcl} [\![*]\!] & = & (\mathcal{P}(\mathcal{T}), \subseteq) \\ [\![\oplus \to k]\!] & = & (\mathcal{P}(\mathcal{T}), \subseteq) \to_{\mathrm{M}} [\![k]\!] \\ [\![\ominus \to k]\!] & = & (\mathcal{P}(\mathcal{T}), \supseteq) \to_{\mathrm{M}} [\![k]\!] \end{array}$$

where $A \to_{\mathrm{M}} B$ is the set of monotone functions from A to B ordered canonically by:

$$f \preceq g \text{ iff } \forall x \in A. f(x) \preceq g(x) \ .$$

An *environment* ρ is a function from type constructors to ordered sets that satisfies the two following properties:

- $\rho(c) \in [\![K(c)]\!]$;
- $\rho(c)$ maps semantic types to semantic types.

The semantics of types is defined inductively on the syntax of types.

$$\begin{array}{lcl} [\![\bot]\!]_\rho & = & \boxed{\bot} \\ [\![\tau \cup \tau']\!]_\rho & = & [\![\tau]\!]_\rho \boxed{\cup} [\![\tau']\!]_\rho \\ [\![c]\!]_\rho & = & \rho(c) \\ [\![\chi\tau]\!]_\rho & = & [\![\chi]\!]_\rho([\![\tau]\!]_\rho) \end{array}$$

Clearly, the semantics $[\![\tau]\!]_\rho$ of a type τ is well-defined and is a semantic type.

$$\textsc{Bottom}\quad \bot <: \tau \qquad \textsc{Union-Left}\quad \frac{\tau <: \tau'' \qquad \tau' <: \tau''}{\tau \cup \tau' <: \tau''} \qquad \textsc{Union-Right-1}\quad \frac{\chi <: \tau}{\chi <: \tau \cup \tau'}$$

$$\textsc{Union-Right-2}\quad \frac{\chi <: \tau'}{\chi <: \tau \cup \tau'} \qquad \textsc{Constructor}\quad c <: c$$

$$\textsc{Covariance}\quad \frac{\chi : \oplus \to k \quad \chi' : \oplus \to k \quad \chi <: \chi' \quad \tau <: \tau'}{\chi\tau <: \chi'\tau'} \qquad \textsc{Contravariance}\quad \frac{\chi : \ominus \to k \quad \chi' : \ominus \to k \quad \chi <: \chi' \quad \tau' <: \tau}{\chi\tau <: \chi'\tau'}$$

Figure 5. Subtyping Rules (Rich Types)

6.3 Subtyping Relation

The subtyping relation <: is defined using the inference rules of Fig. 5.

6.4 Soundness of the Subtyping Relation

THEOREM 24 (SOUNDNESS). *If* $\tau <: \tau'$ *then, for any environment* ρ, *the inclusion* $[\![\tau]\!]_\rho \subseteq [\![\tau']\!]_\rho$ *holds.*

PROOF. By induction on a derivation of $\tau <: \tau'$. We simultaneously prove that if $\chi <: \chi'$, then $[\![\chi]\!]_\rho, [\![\chi']\!]_\rho \in [\![K(\chi)]\!]$ and $[\![\chi]\!]_\rho \preceq [\![\chi']\!]_\rho$.

- Rule BOTTOM: the semantic type $\boxed{\bot}$ is the least semantic type.
- Rule UNION-LEFT: by induction hypothesis,
$$[\![\tau]\!]_\rho \cup [\![\tau']\!]_\rho \subseteq [\![\tau'']\!]_\rho \ ;$$
hence, as $[\![\tau'']\!]_\rho$ is closed,
$$[\![\tau]\!]_\rho \boxed{\cup} [\![\tau']\!]_\rho = \overline{[\![\tau]\!]_\rho \cup [\![\tau']\!]_\rho} \subseteq [\![\tau'']\!]_\rho \ .$$
- Rule UNION-RIGHT-1: $[\![\tau]\!]_\rho \subseteq [\![\tau]\!]_\rho \boxed{\cup} [\![\tau']\!]_\rho$.
- Rule UNION-RIGHT-2: $[\![\tau']\!]_\rho \subseteq [\![\tau]\!]_\rho \boxed{\cup} [\![\tau']\!]_\rho$.
- Rule CONSTRUCTOR: immediate.
- Rule COVARIANCE: By induction hypothesis, we have $[\![\chi]\!]_\rho, [\![\chi']\!]_\rho \in [\![K(\chi)]\!] = [\![\oplus \to k]\!]$, $[\![\chi]\!]_\rho \preceq [\![\chi']\!]_\rho$, and $[\![\tau]\!]_\rho \subseteq [\![\tau']\!]_\rho$. Therefore, $[\![\chi\tau]\!]_\rho, [\![\chi'\tau']\!]_\rho \in [\![k]\!] = [\![K(\chi\tau)]\!]$, and $[\![\chi\tau]\!]_\rho \preceq [\![\chi'\tau']\!]_\rho$ as wanted.
- Rule CONTRAVARIANCE: By induction hypothesis, we have $[\![\chi]\!]_\rho, [\![\chi']\!]_\rho \in [\![K(\chi)]\!] = [\![\ominus \to k]\!]$, $[\![\chi]\!]_\rho \preceq [\![\chi']\!]_\rho$, and $[\![\tau']\!]_\rho \subseteq [\![\tau]\!]_\rho$. Therefore, $[\![\chi\tau]\!]_\rho, [\![\chi'\tau']\!]_\rho \in [\![k]\!] = [\![K(\chi\tau)]\!]$, and $[\![\chi\tau]\!]_\rho \preceq [\![\chi'\tau']\!]_\rho$ as wanted. □

6.5 Completeness of the Subtyping Relation

The converse of the soundess theorem would be that, if for all calculi and for all environments ρ we have $[\![\tau]\!]_\rho \subseteq [\![\tau']\!]_\rho$, then $\tau <: \tau'$. We actually prove a stronger result. We distinguish some type constructors for function, pair and constant type constructions, and we restrict ourselves to environments ρ that map these constructors to the expected semantics:

$$\begin{array}{lcl} \rho(\_ \to \_)(\mathcal{E})(\mathcal{E}') & = & \mathcal{E}\boxed{\rightarrow}\mathcal{E}' \\ \rho(\_ \times \_)(\mathcal{E})(\mathcal{E}') & = & \mathcal{E}\boxtimes\mathcal{E}' \\ \rho(\kappa) & = & \boxed{\kappa} \end{array}$$

This way, we can consider the refined type system as an extension of the simple type system of the previous section. We prove the completeness of the subtyping relation for this extension.

The completeness of the subtyping relation is proved by translating types into the simpler type system of Sect. 5. Ideally, we would like to define a translation $\langle \_ \rangle$ with two properties:

- it should be *faithful*, that is, there exists an environment ρ_0 such that for all type τ we have $[\![\tau]\!]_{\rho_0} = [\![\langle\tau\rangle]\!]$;
- it should lift one subtyping relation into the other: if $\langle\tau\rangle <: \langle\tau'\rangle$, then $\tau <: \tau'$.

Indeed, if we can find such a translation, it is clear that the subtyping relation is complete: if $[\![\tau]\!]_{\rho_0} \subseteq [\![\tau']\!]_{\rho_0}$, then $[\![\langle\tau\rangle]\!] \subseteq [\![\langle\tau'\rangle]\!]$ by faithfulness, $\langle\tau\rangle <: \langle\tau'\rangle$ by completeness of the simple type system, and finally $\tau <: \tau'$ by lifting. Actually, we need to define two faithful translations that satisfy a natural generalization of the second properties.

The two translations $\langle\tau\rangle_1$ and $\langle\tau\rangle_1$ are defined inductively below, together with two translations of constructed types $(\chi)_1$ and $\langle\chi\rangle_2$. The constructors for function, pair and constant types are translated into the corresponding constructions in the simple type system. For the remaining constructors, we assume an injective mapping ϕ which associates to each constructor c a constant κ. The definition of the translations differ for these constructors, which allows to distinguish pair types and function types from other type constructors.

$$\begin{array}{lcll} \langle\bot\rangle_i & = & \bot & \\ \langle\tau\cup\tau'\rangle_i & = & \langle\tau\rangle_i\cup\langle\tau'\rangle_i & \\ \langle\tau\to\tau'\rangle_i & = & \langle\tau\rangle_i\to\langle\tau'\rangle_i & \\ \langle\tau\times\tau'\rangle_i & = & \langle\tau\rangle_i\times\langle\tau'\rangle_i & \\ \langle\kappa\rangle_i & = & \kappa & \\ \langle\chi\rangle_1 & = & \bot\times(\chi)_1 & \\ \langle\chi\rangle_2 & = & \bot\to(\chi)_2 & \\ (c)_i & = & \phi(c) & \\ (\chi\tau)_i & = & \langle\tau\rangle_i\times(\chi)_i & \text{when } \chi:\oplus\to k \\ (\chi\tau)_i & = & \langle\tau\rangle_i\to(\chi)_i & \text{when } \chi:\ominus\to k \end{array}$$

The two corresponding environment ρ_1 and ρ_2 are defined by:

$$\begin{array}{lcll} \rho_i(\_ \to \_)(\mathcal{E})(\mathcal{E}') & = & \mathcal{E}\boxed{\rightarrow}\mathcal{E}' & \\ \rho_i(\_ \times \_)(\mathcal{E})(\mathcal{E}') & = & \mathcal{E}\boxtimes\mathcal{E}' & \\ \rho_i(\kappa) & = & \boxed{\kappa} & \\ \rho_i(c) & = & w^i_{K(c)}(\boxed{\kappa}) & \text{where } \kappa=\phi(c) \end{array}$$

where the family of wrapper functions w^i_k is defined by:

$$\begin{array}{lcl} w^1_*(\mathcal{E}) & = & \boxed{\bot}\boxtimes\mathcal{E} \\ w^2_*(\mathcal{E}) & = & \boxed{\bot}\boxed{\rightarrow}\mathcal{E} \\ w^i_{\oplus\to k}(\mathcal{E})(\mathcal{E}') & = & w^i_k(\mathcal{E}'\boxtimes\mathcal{E}) \\ w^i_{\ominus\to k}(\mathcal{E})(\mathcal{E}') & = & w^i_k(\mathcal{E}'\boxed{\rightarrow}\mathcal{E}) \end{array}$$

LEMMA 25. *The environments ρ_1 and ρ_2 are well-formed and provide the right semantics for function, pair and constant constructions.*

PROOF. We only prove that $\rho_i(c) \in [\![K(c)]\!]$. The remaining conditions are clearly satisfied. We simultaneously show by induction on k that, for all set of terms $\mathcal{E}$, we have $w^i_k(\mathcal{E}) \in [\![k]\!]$ and that, if $\mathcal{E} \subseteq \mathcal{E}'$, then $w^i_k(\mathcal{E}) \preceq w^i_k(\mathcal{E})$.

- Case $k = *$. First, $w^1_*(\mathcal{E}) = (\boxed{\bot}\boxtimes\mathcal{E}) \in \mathcal{P}(\mathcal{T}) = [\![*]\!]$ and $w^2_*(\mathcal{E}) = (\boxed{\bot}\boxed{\rightarrow}\mathcal{E}) \in \mathcal{P}(\mathcal{T}) = [\![*]\!]$. Second, $w^1_*(\mathcal{E}) = (\boxed{\bot}\boxtimes\mathcal{E}) \subseteq (\boxed{\bot}\boxtimes\mathcal{E}') = w^1_*(\mathcal{E}')$ and $w^2_*(\mathcal{E}) = (\boxed{\bot}\boxed{\rightarrow}\mathcal{E}) \subseteq (\boxed{\bot}\boxed{\rightarrow}\mathcal{E}') = w^1_*(\mathcal{E}')$.

- Case $k = \oplus \to k'$. First, we have $w^i_{\oplus\to k'}(\mathcal{E})(\mathcal{E}') = w^i_{k'}(\mathcal{E}'\boxtimes\mathcal{E}) \in [\![k']\!]$ by induction hypothesis. Hence, $w^i_{\oplus\to k'}(\mathcal{E}) \in \mathcal{P}(\mathcal{T}) \to [\![k']\!]$. Besides, if $\mathcal{E}' \subseteq \mathcal{E}''$, then we have $w^i_{\oplus\to k'}(\mathcal{E})(\mathcal{E}') = w^i_{k'}(\mathcal{E}'\boxtimes\mathcal{E}) \preceq w^i_{k'}(\mathcal{E}''\boxtimes\mathcal{E}) = w^i_{\oplus\to k'}(\mathcal{E})(\mathcal{E}'')$ by induction hypothesis. Therefore, $w^i_{\oplus\to k'}(\mathcal{E})$ is monotone. So, $w^i_{\oplus\to k'}(\mathcal{E}) \in [\![k]\!]$.

  Second, by induction hypothesis, if $\mathcal{E} \subseteq \mathcal{E}'$, then we have $w^i_{\oplus\to k'}(\mathcal{E})(\mathcal{E}'') = w^i_{k'}(\mathcal{E}''\boxtimes\mathcal{E}) \preceq w^i_{k'}(\mathcal{E}''\boxtimes\mathcal{E}') = w^i_{\oplus\to k'}(\mathcal{E}')(\mathcal{E}'')$. Hence, $w^i_{\oplus\to k'}(\mathcal{E}) \preceq w^i_{\oplus\to k'}(\mathcal{E}')$.

- Case $k = \ominus \to k'$. First, we have $w^i_{\ominus\to k'}(\mathcal{E})(\mathcal{E}') = w^i_{k'}(\mathcal{E}'\boxed{\rightarrow}\mathcal{E}) \in [\![k']\!]$ by induction hypothesis. Hence, $w^i_{\ominus\to k'}(\mathcal{E}) \in \mathcal{P}(\mathcal{T}) \to [\![k']\!]$. Besides, if $\mathcal{E}' \supseteq \mathcal{E}''$, then we have $w^i_{\ominus\to k'}(\mathcal{E})(\mathcal{E}') = w^i_{k'}(\mathcal{E}'\boxed{\rightarrow}\mathcal{E}) \preceq w^i_{k'}(\mathcal{E}''\boxed{\rightarrow}\mathcal{E}) = w^i_{\ominus\to k'}(\mathcal{E})(\mathcal{E}'')$ by induction hypothesis. Therefore, $w^i_{\ominus\to k'}(\mathcal{E})$ is monotone. So, $w^i_{\ominus\to k'}(\mathcal{E}) \in [\![k]\!]$.

  Second, by induction hypothesis, if $\mathcal{E} \subseteq \mathcal{E}'$, then we have $w^i_{\ominus\to k'}(\mathcal{E})(\mathcal{E}'') = w^i_{k'}(\mathcal{E}''\boxed{\rightarrow}\mathcal{E}) \preceq w^i_{k'}(\mathcal{E}''\boxed{\rightarrow}\mathcal{E}') = w^i_{\ominus\to k'}(\mathcal{E}')(\mathcal{E}'')$. Hence, $w^i_{\ominus\to k'}(\mathcal{E}) \preceq w^i_{\ominus\to k'}(\mathcal{E}')$. □

LEMMA 26 (FAITHFUL TRANSLATION).

$$[\![\tau]\!]_{\rho_i} = [\![\langle\tau\rangle_i]\!]$$

PROOF SKETCH. The proof is by induction on τ. We simultaneously prove $[\![\chi]\!]_{\rho_i} = w^i_{K(\chi)}([\![(\chi)_i]\!])$. We only consider contructed types. Other cases are immediate.

- $[\![c]\!]_{\rho_i} = \rho_i(c) = w^i_{K(c)}(\boxed{\kappa}) = w^i_{K(c)}([\![(c)_i]\!])$ where $\kappa = \phi(c)$.
- $[\![\chi\tau]\!]_{\rho_i} = [\![\chi]\!]_{\rho_i}([\![\tau]\!]_{\rho_i}) = w^i_{K(\chi)}([\![(\chi)_i]\!])([\![\langle\tau\rangle_i]\!]) = w^i_{K(\chi\tau)}([\![(\chi\tau)_i]\!])$.
- $[\![\chi]\!]_{\rho_i} = w^i_*([\![(\chi)_i]\!]) = [\![\langle\chi\rangle_i]\!]$ when $K(\chi) = *$. □

THEOREM 27 (COMPLETENESS). *If, for any calculus and for any environment ρ such that $\rho(\_ \to \_) = \boxed{\rightarrow}$, $\rho(\_ \times \_) = \boxtimes$, and $\rho(\kappa) = \boxed{\kappa}$ for all constants κ, the inclusion $[\![\tau]\!]_\rho \subseteq [\![\tau']\!]_\rho$ holds, then $\tau <: \tau'$.*

PROOF. We interpret the semantics of types in the calculus defined in Sect. 4.

By lemma 26 (Faithful Translation), if $[\![\tau]\!]_{\rho_i} \subseteq [\![\tau']\!]_{\rho_i}$, then $[\![\langle\tau\rangle_i]\!] \subseteq [\![\langle\tau'\rangle_i]\!]$, so, by Theorem 21 (Completeness), $\langle\tau\rangle_i <: \langle\tau'\rangle_i$. Therefore, it is sufficient to prove that, if $\langle\tau\rangle_i <: \langle\tau'\rangle_i$, then $\tau <: \tau'$. The proof is by induction on τ and τ'. We simultaneously prove that if $(\chi)_i <: (\chi')_i$, than $K(\chi) = K(\chi')$ and $\chi <: \chi'$.

- Case $\langle\bot\rangle_i <: \langle\tau\rangle_i$. By rule BOTTOM, $\bot <: \tau$.
- Case $\langle\tau\cup\tau'\rangle_i <: \langle\tau''\rangle_i$. We have $\langle\tau\rangle_i\cup\langle\tau'\rangle_i <: \langle\tau''\rangle_i$. This must be derived from the rule UNION-LEFT of the first subtyp-

ing relation. Hence, we have $\langle\tau\rangle_i <: \langle\tau''\rangle_i$ and $\langle\tau'\rangle_i <: \langle\tau''\rangle_i$. By induction hypothesis, $\tau <: \tau''$ and $\tau' <: \tau''$. Finally, by rule UNION-LEFT, $\tau \cup \tau' <: \tau''$.

- Case $\langle\chi\rangle_i <: \langle\bot\rangle_i$. This case is not possible: this cannot be derived from any rule.

- Case $\langle\chi\rangle_i <: \langle\tau \cup \tau'\rangle_i$. We have $\langle\chi\rangle_i <: \langle\tau\rangle_i \cup \langle\tau'\rangle_i$ where $\langle\chi\rangle_i$ is a constructed type. Hence, this must be derived either from the rule UNION-RIGHT-1 or from the rule UNION-RIGHT-2 of the first subtyping relation. Hence, we have either $\langle\chi\rangle_i <: \langle\tau\rangle_i$ or $\langle\chi\rangle_i <: \langle\tau'\rangle_i$. By induction hypothesis, we have either $\chi <: \tau$ or $\chi <: \tau'$. Finally, by rule UNION-RIGHT-1 or rule UNION-RIGHT-2, $\chi <: \tau \cup \tau'$.

- Case $\langle\chi\rangle_i <: \langle\chi'\rangle_i$ where either one of χ and χ' is a distinguished constructed type (function, pair or constant type) but not the other, or are disctinct distinguished constructed type. There exist i such that $\langle\chi\rangle_i <: \langle\chi'\rangle_i$ cannot be derived. So, this case is not possible.

- Case $\langle\tau_2 \to \tau_1\rangle_i <: \langle\tau_4 \to \tau_3\rangle_i$. We have $\langle\tau_2\rangle_i \to \langle\tau_1\rangle_i <: \langle\tau_4\rangle_i \to \langle\tau_3\rangle_i$. This must be derived from the rule FUNCTION. Hence, we have $\langle\tau_1\rangle_i <: \langle\tau_3\rangle_i$ and $\langle\tau_4\rangle_i <: \langle\tau_2\rangle_i$. By induction hypothesis, $\tau_1 <: \tau_3$ and $\tau_4 <: \tau_2$. Finally, by rules CONSTRUCTOR, CONTRAVARIANCE and COVARIANCE, $\tau_2 \to \tau_1 <: \tau_4 \to \tau_3$.

- Case $\langle\tau_1 \times \tau_2\rangle_i <: \langle\tau_3 \times \tau_4\rangle_i$. We have $\langle\tau_1\rangle_i \times \langle\tau_2\rangle_i <: \langle\tau_3\rangle_i \times \langle\tau_4\rangle_i$. This must be derived from the rule PAIR. Hence, we have $\langle\tau_1\rangle_i <: \langle\tau_3\rangle_i$ and $\langle\tau_2\rangle_i <: \langle\tau_4\rangle_i$. By induction hypothesis, $\tau_1 <: \tau_3$ and $\tau_2 <: \tau_4$. Finally, by rules CONSTRUCTOR and COVARIANCE, $\tau_1 \times \tau_2 <: \tau_3 \times \tau_4$.

- Case $\langle\kappa\rangle_i <: \langle\kappa\rangle_i$. By rule CONSTRUCTOR, $\kappa <: \kappa$.

- Case $\langle\chi\rangle_i <: \langle\chi'\rangle_i$ where neither χ nor χ' is a distinguished constructed type. We have $\bot \times (\chi)_1 <: \bot \times (\chi')_1$ and $\bot \to (\chi)_1 <: \bot \to (\chi')_1$. Hence, by rule PAIR and FUNCTION, $(\chi)_i <: (tabs')_i$. Therefore, by induction hypothesis, $\chi <: \chi'$.

- Case $(\chi)_i <: (\chi')_i$ where one of χ or χ' is a type constructor and the other a type application. One of $(\chi)_i$ and $(\chi')_i$ is a constant type while the other is a pair type or a function type. This cannot be derived from any rule. So, this case is not possible.

- Case $(c)_i <: (c')_i$. We have $\phi(c) <: \phi(c')$. Only rule CONSTANT may apply. Then, $\phi(c) = \phi(c')$. By injectivity of ϕ, $c = c'$. Finally, $K(c) = K(c')$ and, by rule CONSTRUCTOR, $c <: c'$.

- Case $(\chi\tau)_i <: (\chi'\tau')_i$.

  If the first parameters of χ and χ' have different variance, then the translation of one of these constructed types is a function type while the translation of the other is a pair. This cannot be derived from any rule. So, the variance must be the same.

  Then, either rule PAIR or rule FUNCTION apply depending on the variance. So, $(\chi)_i <: (\chi')_i$ and either $\langle\tau\rangle_i <: \langle\tau'\rangle_i$ or $\langle\tau'\rangle_i <: \langle\tau\rangle_i$ depending on the variance. By induction hypothesis, $K(\chi) = K(\chi')$, $\chi <: \chi'$, and either $\tau <: \tau'$ or $\tau' <: \tau$. Finally, $K(\chi\tau) = K(\chi'\tau')$ and, by one of the rule COVARIANCE or CONTRAVARIANCE, $\chi\tau <: \chi'\tau'$. □

7 Related Work

This work is a continuation of our work with Melliès on semantic types [13, 17]. These two papers focus on defining types, especially recursive types, as set of terms, while we study here the subtyping relation induced by these definitions.

Defining the semantics of types as closed sets of terms is very natural. For instance, in domain theory, types can be interpreted as *ideals* [12], that is, sets that are downward closed and closed under directed limits. *Reducibility candidates* [8] are also closed sets of terms. Girard [9] reformulates the candidates as sets of terms closed by biorthogonality in his proof of cut elimination for linear logic. Meanwhile, Krivine [4, 11] has developed a comprehensive framework based on orthogonality, in order to analyze types as *specification* of terms. In semantics, Pitts [15] uses relations closed by biorthogonality to study parametric polymorphism in an operational setting.

Damm [3] studies subtyping for a deterministic calculus with recursive types with union and intersection. He takes a domain theoretic approach based on the ideal model [12]. A subtyping algorithm is specified by encoding types into tree automata and defining the subtyping relation as the inclusion of the recognized languages. The soundness and completeness of this algorithm with respect to the semantics of types is proven.

Frisch, Castagna and Benzaken [7] use an approach similar to ours to design a subtyping relation for a typed calculus with union and intersection types. They want to define the subtyping relation of this calculus in a semantic way, as the inclusion of the denotation of types. But their calculus is typed, so its semantics depends on the subtyping relation. In order to get rid of this circularity, they consider a class of calculi (called *models*). While we try to describe as large a class as possible, the authors design a class such that the subtyping relation has good properties (for instance, distributivity of union and intersection).

8 Future Work

8.1 Strict Pairs and Recursive Types

The type system presented here is not as rich as the type systems of XDuce [10] and CDuce [7] for two reasons. First, for the sake of simplicity, we have not considered recursive types. Second, we deal with a very large class of calculi, in which some subtyping assertions such as $(\tau_1 \cup \tau_2) \times \tau <: (\tau_1 \times \tau) \cup (\tau_2 \times \tau)$ do not hold (as hinted in the introduction). We would need to reduce the class of calculi to get a coarser subtyping relation.

In previous work [13, 17], we have developped some tools to deal with recursive types. Severe restrictions must be made on the class of calculi considered in order to be able to define the semantics of types. Still, we believe the class is still large enough to get interesting results.

Intuitively, the subtyping assertion above hold when pairs are strict (that is, when their two components are evaluated once when the pair is build). But this requirement is hard to state in an abstract way. We believe a natural restriction would be instead to require product and closure to commute, that is, $\overline{\mathcal{E} \boxtimes \mathcal{E}'} = \overline{\mathcal{E}} \boxtimes \overline{\mathcal{E}'}$. It is easy to check that the subtyping assertion above holds under this assumption.

8.2 Intersection Types

Intersection types are harder to handle than union types. The natural semantics for intersection types is set intersection:

$$\mathcal{E} \boxed{\cap} \mathcal{E}' = \mathcal{E} \cap \mathcal{E}' \ .$$

Then, it is clear that the dual of the subtyping rules for union types are sound. But there are other sound subtyping rules. For instance, we have $(\tau_1 \times \tau_3) \cap (\tau_2 \times \tau_4) <: (\tau_1 \cap \tau_2) \times (\tau_3 \cap \tau_4)$. Furthermore, in order to check a subtyping assertion involving intersection types, it seems one need to introduce union types: to check $(\tau_1 \to \tau_2) \cap (\tau_3 \to \tau_4) <: \tau_5 \to \tau_6$, one may have to check whether $\tau_5 <: \tau_1 \cup \tau_3$. But union does not interact well with intersection. In particular, the distributivity law $(\tau_1 \cup \tau_2) \cap \tau = (\tau_1 \cap \tau) \cup (\tau_2 \cap \tau_2)$ does not hold in general.

8.3 Side Effects

Our framework does not handle calculi with side effects. Referring to Pitts and Stark [16], it seems we need to introduce a notion of *value*, where a value is an expression with an extensional behavior. Then, for instance, a value has type $\tau' \to \tau$ if whenever it is applies to an expression of type τ' the resulting expression has type τ. This definition is extended to expression by closure: an expression has type $\tau' \to \tau$ if it is in the closure of the set of values of type $\tau' \to \tau$.

It is interesting to notice that in presence of side effects, the typing rule for union elimination need to restricted in a similar way to ours, as shown by Dunfield and Pfenning [6]. Intuitively, in both cases, this restriction comes from the fact that the result of several evaluations of a same expression may differ, either due to non-determinism or side effects.

We conjecture that our subtyping rules remain sound in presence of side effects. On the other hand, some subtyping rules, such as $(\tau \to \tau_1) \cap (\tau \to \tau_2) <: \tau \to (\tau_1 \cap \tau_2)$, become unsound, as shown by Davies and Pfenning [5].

9 References

[1] Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.4*, Feb. 2003. Available from `http://coq.inria.fr/doc/main.html`.

[2] L. Dami. Labelled reductions, runtime errors and operational subsumption. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 782–793. Springer, 1997.

[3] F. Damm. Subtyping with union types, intersection types and recursive types II. Research Report 2259, INRIA Rennes, May 1994.

[4] V. Danos and J.-L. Krivine. Disjunctive tautologies and synchronisation schemes. In *Computer Science Logic'00*, volume 1862 of *Lecture Notes in Computer Science*, pages 292–301. Springer, 2000.

[5] R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 198–208, Montréal, Canada, Sept. 2000. ACM Press.

[6] J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Proc. 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'03)*, Lecture Notes in Computer Science. Springer–Verlag, 2003.

[7] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *17th IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[8] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de doctorat d'État, University of Paris VII, 1972.

[9] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[10] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. To appear; short version in ICFP 2000.

[11] J.-L. Krivine. Typed lambda-calculus in classical Zermelo-Fraenkel set theory. *Archive of Mathematical Logic*, 40(3):189–205, 2001.

[12] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1-2):95–130, 1986.

[13] P.-A. Melliès and J. Vouillon. Recursive polymorphic types and parametricity in an operational framework, 2004. Available from `http://www.pps.jussieu.fr/~vouillon/publi/#semtypes2`.

[14] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[15] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in computer Science*, 10:321–359, 2000.

[16] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

[17] J. Vouillon and P.-A. Melliès. Semantic types: A fresh look at the ideal model for types. In *Proceedings of the 31th ACM Conference on Principles of Programming Languages*, pages 52–63, Venezia, Italia, Jan. 2004. ACM Press.