Subtyping Union Types

Jérôme Vouillon

CNRS and Université Paris 7
Case 7014, 2 Place Jussieu, 75251 Paris Cedex 05, France
`Jerome.Vouillon@pps.jussieu.fr`

Abstract. Subtyping 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. This allows us to find a subtyping relation which is both robust (it is sound for all calculi) and precise (it is complete with respect to the class of calculi).

Keywords: union types, subtyping, semantics, lambda-calculus.

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 give some rough intuition about types. For these examples, 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 $\top$ is a term that 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, subtypes of one another. We can now present some typing relations that only hold under some conditions on the language.

– In some call-by-value languages, we can have $\top <: \bot \to \bot$. Indeed, this assertion holds when the application is strict on its right argument (for any

first argument which evaluates without error), that is, when we can apply any term e which evaluates without error to a term e' that diverges and get a term $e\,e'$ which diverges.
- In some call-by-value languages, 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 can hold in a call-by-need language with non-determinism, as an expression is then evaluated at most once.
- In a deterministic language, union of function types $\tau \rightarrow \tau'$ obey very special subtyping rules when τ is finite (as observed by Damm [1]). 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 $\tau_2 \rightarrow \tau_1 <: \tau_2' \rightarrow \tau_1'$;
- 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 calculus), 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 (Sect. 2) and a particular instance is given (Sect. 3). We present a simple type system, define a subtyping relation and prove the soundness and completeness of the relation (Sect. 4). We conclude by presenting related work (Sect. 5) and directions for future work (Sect. 6). Most proofs are omitted for lack of space. They are available online in an extended version of the paper [2].

2 A Class of Abstract Calculi

2.1 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 a call-by-name calculus. This is actually possible in the calculus of Sect. 3, but only because we made some specific choices about types.

As it turns out, it is convenient to only consider sets of terms that satisfy a given closure property: we assume given a *closure operator* on sets of terms, that is, a function $\mathcal{E} \mapsto \overline{\mathcal{E}}$ which is extensive ($\mathcal{E} \subseteq \overline{\mathcal{E}}$), idempotent ($\overline{\overline{\mathcal{E}}} = \overline{\mathcal{E}}$) and monotone. A set of terms $\mathcal{E}$ is said to be *closed* if $\mathcal{E} = \overline{\mathcal{E}}$. The idea is that the closure $\overline{\mathcal{E}}$ of a set of terms $\mathcal{E}$ is the set of terms that cannot be distinguished (as far as types are concerned) from the terms in $\mathcal{E}$. Thus, different choices of a closure operator yields different interpretation of types.

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 sets of terms. We call *semantic type* a closed set of terms included in $\mathcal{S}$ and including $\mathcal{N}$. We require the semantics $[\![\tau]\!]$ of a syntactic type τ to be a semantic type.

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. 3, we will have $\mathcal{S} \neq \mathcal{T}$ and $\mathcal{N} = \overline{\emptyset}$, but in [3], we have $\mathcal{S} \neq \mathcal{T}$ and $\mathcal{N} \neq \overline{\emptyset}$, and in [4], we have $\mathcal{S} = \mathcal{T}$ and $\mathcal{N} \neq \overline{\emptyset}$. Finally, in the case of *reducibility candidates* [5], one has $\mathcal{S} \neq \mathcal{T}$ and $\mathcal{N} \neq \overline{\emptyset}$ (safe terms are strongly normalizing terms, and some terms such as a variable x can be given any type).

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.

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

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.

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

But 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 set $\mathcal{N}$ is included in the inverse image of $\mathcal{N}$. 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.

$$[\![\tau' \to \tau]\!] = \mathcal{S} \cap \bigcap_{e' \in [\![\tau']\!]} \{e \,|\, e\,e' \in [\![\tau]\!]\}$$
$$[\![\tau \times \tau']\!] = \mathcal{S} \cap \mathtt{fst}^{-1}([\![\tau]\!]) \cap \mathtt{snd}^{-1}([\![\tau']\!])$$

It is really natural for all these functions to be strict, as they are destructors. The continuity properties may seem harder to achieve. We will see in Sect. 3.2, that it is actually straightforward to define a closure operator ensuring these properties.

Note that if $\mathcal{N} = \overline{\emptyset}$, 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 $\mathcal{N} \neq \overline{\emptyset}$.

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:

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

2.2 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:

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

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

Note that we consider the closure operator as part of the calculus. Thus, two different calculi can be identical except for their closure operators. They can be understood as two (semantically) typed variants of a same untyped calculus.

2.3 Semantic Operations

We define one operation on sets of terms for each type construction we have in mind: bottom type, union of two types, function types, pair types and constant types. These operations are used to define the semantics of types in a straightforward fashion in Sect. 4.1. Note that the semantic union $\boxed{\cup}$ of two sets of terms is not simply their union. Indeed, the union of two closed sets is usually not a closed set. In other words, 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. Our solution is to take the least closed set containing the union. This is not just a technical point, but is actually crucial for typing a calculus with non-determinism, for which we could expect, for instance, a term to be in $\mathcal{E} \boxed{\cup} \mathcal{E}'$ if it behaves erratically either as a term in $\mathcal{E}$ or as a term in $\mathcal{E}'$.

$$\begin{array}{ll}
\boxed{\perp} & = \mathcal{N} \\
\mathcal{E} \boxed{\cup} \mathcal{E}' & = \overline{\mathcal{E} \cup \mathcal{E}'} \\
\mathcal{E}' \boxed{\rightarrow} \mathcal{E} & = \{e \in \mathcal{S} \,|\, \forall e' \in \mathcal{E}'. e\, e' \in \mathcal{E}\} \\
\mathcal{E} \boxed{\times} \mathcal{E}' & = \{e \in \mathcal{S} \,|\, \texttt{fst}\, e \in \mathcal{E} \wedge \texttt{snd}\, e \in \mathcal{E}'\} \\
\boxed{\kappa} & = \overline{\{\kappa\}}
\end{array}$$

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

3 A Concrete Calculus

We present a particular instance of the class of calculi considered. This calculus is used in Sect. 4 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.

3.1 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
$\texttt{fst}\, e$	first projection
$\texttt{snd}\, e$	second projection
κ	constant
$\texttt{if}\ e = \kappa\ \texttt{then}\ e\ \texttt{else}\ e$	conditional
$e \sqcup e$	erratic choice
$e \vee e$	error-avoiding choice
$\texttt{let}\ x = e\ \texttt{in}\ e$	strict let
$\texttt{error}$	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:

$$v ::= \lambda x.e \mid (e, e) \mid \kappa \mid \texttt{error}$$

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'.

$$\textsc{Var-Error}\ \ x \Downarrow \texttt{error} \qquad \textsc{Abs}\ \ \lambda x.e \Downarrow \lambda x.e \qquad \textsc{App}\ \ \frac{e \Downarrow \lambda x.e_1 \qquad e_1[e'/x] \Downarrow v}{e\,e' \Downarrow v} \qquad \textsc{App-Error}\ \ \frac{e \Downarrow v \qquad v \neq \lambda x.e_1}{e\,e' \Downarrow \texttt{error}}$$

$$\textsc{Pair}\ \ (e_1, e_2) \Downarrow (e_1, e_2) \qquad \textsc{Fst}\ \ \frac{e \Downarrow (e_1, e_2) \qquad e_1 \Downarrow v}{\texttt{fst}\, e \Downarrow v} \qquad \textsc{Fst-Error}\ \ \frac{e \Downarrow v \qquad v \neq (e_1, e_2)}{\texttt{fst}\, e \Downarrow \texttt{error}}$$

$$\textsc{Snd}\ \ \frac{e \Downarrow (e_1, e_2) \qquad e_2 \Downarrow v}{\texttt{snd}\, e \Downarrow v} \qquad \textsc{Snd-Error}\ \ \frac{e \Downarrow v \qquad v \neq (e_1, e_2)}{\texttt{snd}\, e \Downarrow \texttt{error}} \qquad \textsc{Constant}\ \ \kappa \Downarrow \kappa$$

$$\textsc{If-Equal}\ \ \frac{e \Downarrow \kappa \qquad e' \Downarrow v}{\texttt{if}\ e = \kappa\ \texttt{then}\ e'\ \texttt{else}\ e'' \Downarrow v} \qquad \textsc{If-Not-Equal}\ \ \frac{e \Downarrow \kappa' \qquad \kappa \neq \kappa' \qquad e'' \Downarrow v}{\texttt{if}\ e = \kappa\ \texttt{then}\ e'\ \texttt{else}\ e'' \Downarrow v}$$

$$\textsc{If-Error}\ \ \frac{e \Downarrow v \qquad v \neq \kappa'}{\texttt{if}\ e = \kappa\ \texttt{then}\ e'\ \texttt{else}\ e'' \Downarrow \texttt{error}} \qquad \textsc{Para-Left}\ \ \frac{e \Downarrow v}{e \sqcup e' \Downarrow v} \qquad \textsc{Para-Right}\ \ \frac{e' \Downarrow v}{e \sqcup e' \Downarrow v}$$

$$\textsc{Catch-Left}\ \ \frac{e \Downarrow v \qquad v \neq \texttt{error}}{e \vee e' \Downarrow v} \qquad \textsc{Catch-Right}\ \ \frac{e' \Downarrow v \qquad v \neq \texttt{error}}{e \vee e' \Downarrow v} \qquad \textsc{Catch-Error}\ \ \frac{e \Downarrow \texttt{error} \qquad e' \Downarrow \texttt{error}}{e \vee e' \Downarrow \texttt{error}}$$

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

Fig. 1. Semantics

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 that both operations correspond to a least upper bound, as we will see in Sect. 3.4.

We define the following diverging term:

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

3.2 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 3.3.

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:

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

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 $\mathcal{C}$:

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

These two functions define 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.*

3.3 The Closure Operator

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

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

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 (the term $F[e]$ is the term which results from replacing _ by e in the frame F):

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

A term e is safe when it does not reduce to the error. Thus, we define the set $\mathcal{S}$ by:

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

The orthogonality relation is defined by:

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

As indicated in the previous section 3.2, this induces a closure operator on sets of terms. This is the closure operator that we choose to associate to our calculus.

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. 3.4 for a more precise characterization of this property).

3.4 Properties of the Calculus

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

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 $\overline{\mathcal{E}}$.*

The contexts have 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:

$$f = \lambda x.\texttt{if}\ x = \kappa\ \texttt{then}\ (\texttt{if}\ x = \kappa\ \texttt{then diverge else error})\ \texttt{else diverge}\ .$$

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 $f\,v' \Downarrow v$. We then have the expected result.

Lemma 3 (Context Linearity). *Contexts are linear.*

Ordering of Terms and Contexts. We define the *contextual preorder* on terms by $e \leq e'$ if and only if $\overline{\{e\}} \subseteq \overline{\{e'\}}$. Likewise, we define a preorder on contexts by $c \leq c'$ if and only if $\{c\}^\perp \subseteq \{c'\}^\perp$. Note that we choose 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. We present the relative ordering of some interesting terms and contexts. This ordering is illustrated below.

$$\begin{tikzcd}
 & \texttt{error} \arrow[d, dash] & & & & Id \arrow[d, dash] & \\
 & e_1 \sqcup e_2 \arrow[dl, dash] \arrow[dr, dash] & & & & c_1 \vee c_2 \arrow[dl, dash] \arrow[dr, dash] & \\
e_1 \arrow[dr, dash] & & e_2 \arrow[dl, dash] & & c_1 & & c_2 \\
 & \texttt{diverge} & & & & &
\end{tikzcd}$$

Lemma 4 (Least Upper Bounds). *For all terms e, e' and for all contexts c, c', we have: $\overline{\{e \sqcup e'\}} = \overline{\{e\}}\boxed{\sqcup}\overline{\{e'\}}$ and $\{c \vee c'\}^\perp = \{c\}^\perp\boxed{\sqcup}\{c'\}^\perp$. 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'.*

Lemma 5 (Divergence). *The term* `diverge` *is a least term. In particular,* `diverge` $\in \boxed{\bot}$.

Sets of Values. We write $\mathcal{V}(\mathcal{E})$ for the set of values contained in a set of terms $\mathcal{E}$: $\mathcal{V}(\mathcal{E}) = \{v \mid v \in \mathcal{E}\}$. A direct consequence of Lemma 2 (Terms and Values) is that a closed set of terms is characterized by its values: $\overline{\mathcal{E}} = \overline{\mathcal{V}(\overline{\mathcal{E}})}$. It seems therefore natural to study some of the properties of the sets of values $\mathcal{V}(\overline{\mathcal{E}})$.

Lemma 6 (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: $\boxed{\bot} = \overline{\emptyset}$.*

Lemma 7 (Union and Values). *The values of the closed union of two closed sets is the union of the values of each closed sets: $\mathcal{V}(\overline{\mathcal{E}}\boxed{\sqcup}\overline{\mathcal{E}'}) = \mathcal{V}(\overline{\mathcal{E}}) \cup \mathcal{V}(\overline{\mathcal{E}'})$*

We say that a set of terms $\mathcal{E}$ is *directed* when it is non-empty and when each pair of terms of this subset has an upper bound in this subset.

Lemma 8 (Prime when Directed). *If the set $\mathcal{V}(\overline{\mathcal{E}})$ is directed then the set $\overline{\mathcal{E}}$ 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 $\overline{\mathcal{E}} \subseteq \overline{\mathcal{E}_2}$.*

Instance of the Class of Calculi. We have the expected result:

Lemma 9. *The calculus is an instance of the class specified in Sect. 2.2.*

Orthogonality Functions-Arguments. Just as we defined an orthogonality relation between terms and contexts in Sect. 3.2, 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

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

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

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

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

Lemma 10 (Function Orthogonality). *The closure induced by function orthogonality is strictly finer than the closure induced by context orthogonality: for all sets of terms $\mathcal{E}$, we have*

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

but the converse inclusion does not always hold. As a consequence,

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

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 11 (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 $\langle c \rangle \in \mathcal{E} \boxed{\rightarrow} \mathcal{E}_0$.*

4 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 [6].

4.1 Types

The syntax of types is given by the following grammar.

$$\begin{array}{rll@{\qquad}rll}
\tau ::= & \chi & \text{constructed type} & \chi ::= & \tau \to \tau & \text{function type} \\
 & \bot & \text{bottom type} & & \tau \times \tau & \text{pair type} \\
 & \tau \cup \tau & \text{union type} & & \kappa & \text{constant type}
\end{array}$$

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

$$\begin{array}{ll@{\qquad}ll}
[\![\tau \to \tau']\!] & = [\![\tau]\!] \boxed{\Rightarrow} [\![\tau']\!] & [\![\bot]\!] & = \boxed{\bot} \\
[\![\tau \times \tau']\!] & = [\![\tau]\!] \boxed{\times} [\![\tau']\!] & [\![\tau \cup \tau']\!] & = [\![\tau]\!] \boxed{\cup} [\![\tau']\!] \\
[\![\kappa]\!] & = \boxed{\kappa} & &
\end{array}$$

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

4.2 Subtyping Relation

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

$$\begin{array}{cccc}
\dfrac{\tau_1 <: \tau_1' \qquad \tau_2' <: \tau_2}{\tau_2 \to \tau_1 <: \tau_2' \to \tau_1'}\ \text{FUNCTION} &
\dfrac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}\ \text{PAIR} &
\kappa <: \kappa\ \ \text{CONSTANT} &
\bot <: \tau\ \ \text{BOTTOM}
\end{array}$$

$$\begin{array}{ccc}
\dfrac{\tau <: \tau'' \qquad \tau' <: \tau''}{\tau \cup \tau' <: \tau''}\ \text{UNION-LEFT} &
\dfrac{\chi <: \tau}{\chi <: \tau \cup \tau'}\ \text{UNION-RIGHT-1} &
\dfrac{\chi <: \tau'}{\chi <: \tau \cup \tau'}\ \text{UNION-RIGHT-2}
\end{array}$$

Fig. 2. Subtyping Rules

4.3 Soundness of the Subtyping Relation

The soundness of the subtyping relation is straightforward.

Theorem 12 (Soundness). *If $\tau <: \tau'$, then $[\![\tau]\!] \subseteq [\![\tau']\!]$.*

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

- 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: $[\![\tau']\!] \subseteq [\![\tau]\!] \boxed{\cup} [\![\tau']\!]$. □

4.4 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 13 (Homogeneity). *The set of values $\mathcal{V}(\llbracket\chi\rrbracket)$ of a constructed type χ is homogeneous: $\mathcal{V}(\llbracket\tau' \to \tau\rrbracket)$ only contain functions, $\mathcal{V}(\llbracket\tau \times \tau'\rrbracket)$ only contain pairs, $\mathcal{V}(\llbracket\kappa\rrbracket)$ only contain the constant κ.*

Lemma 14 (Directed Set). *The set of values $\mathcal{V}(\llbracket\chi\rrbracket)$ is directed.*

These two lemmas are illustrated below, respectively for function types, pair types and constant types. Values are underlined. The value just above `diverge` is included in all constructed types of the corresponding kind. Given two values in $\mathcal{V}(\llbracket\chi\rrbracket)$, one of their upper bounds in $\mathcal{V}(\llbracket\chi\rrbracket)$ is given.

$$\begin{tikzcd}
 & \underline{\lambda x.(e_1 \sqcup e_2)} & \\
 & (\lambda x.e_1) \sqcup (\lambda x.e_2) \arrow[u, dash] & \\
\underline{\lambda x.e_1} \arrow[ur, dash] & & \underline{\lambda x.e_2} \arrow[ul, dash] \\
 & \underline{\lambda x.\texttt{diverge}} \arrow[ul, dash] \arrow[ur, dash] & \\
 & \texttt{diverge} \arrow[u, dash] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \underline{((e_1 \sqcup e_2), (e_1 \sqcup e_2))} & \\
 & (e_1, e_1') \sqcup (e_2, e_2') \arrow[u, dash] & \\
\underline{(e_1, e_1')} \arrow[ur, dash] & & \underline{(e_2, e_2')} \arrow[ul, dash] \\
 & \underline{(\texttt{diverge}, \texttt{diverge})} \arrow[ul, dash] \arrow[ur, dash] & \\
 & \texttt{diverge} \arrow[u, dash] &
\end{tikzcd}$$

$$\begin{tikzcd}
\underline{\kappa} \\
\texttt{diverge} \arrow[u, dash]
\end{tikzcd}$$

4.5 Completeness of the Subtyping Relation

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

Theorem 15. *If $\llbracket\tau\rrbracket \subseteq \llbracket\tau'\rrbracket$ for the calculus of Sect. 3, then $\tau <: \tau'$.*

Corollary 16 (Completeness). *If $\llbracket\tau\rrbracket \subseteq \llbracket\tau'\rrbracket$ for all calculi, then $\tau <: \tau'$.*

At several points in the proof of completeness, we need to prove an inclusion $\llbracket\tau_1\rrbracket \subseteq \llbracket\tau_1'\rrbracket$ assuming that an inclusion between the semantics of two types built from τ_1 and τ_1' (for instance, $\llbracket\tau_1 \times \tau_2\rrbracket \subseteq \llbracket\tau_1' \times \tau_2'\rrbracket$) 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 types τ and all terms e, $e \in \llbracket\tau\rrbracket$ if and only if $f(e) \in \llbracket F(\tau)\rrbracket$. Then, it is easy to see that, if (F, f) is a typed transformation and $\llbracket F(\tau)\rrbracket \subseteq \llbracket F(\tau')\rrbracket$, then $\llbracket\tau\rrbracket \subseteq \llbracket\tau'\rrbracket$, We thus define three families of typed transformations.

Lemma 17 (Typed Transformations). *The following families of pairs of functions are typed transformations (for the calculus of Sect. 3).*

$$\begin{array}{ll}
F_1(\tau') : \tau \mapsto \tau \times \tau' & \quad f_1 : e \mapsto (e, \texttt{diverge}) \\
F_2(\tau') : \tau \mapsto \tau' \times \tau & \quad f_2 : e \mapsto (\texttt{diverge}, e) \\
F_3(\tau') : \tau \mapsto \tau' \to \tau & \quad f_3 : e \mapsto \lambda x.e
\end{array}$$

Proof (of Theorem 15). We interpret the semantics of types in the calculus defined in Sect. 3. In order to handle the contravariance of the function type, we simultaneously prove by induction on τ and τ' that if $[\![\tau]\!] \subseteq [\![\tau']\!]$ then $\tau <: \tau'$, and if $[\![\tau']\!] \subseteq [\![\tau]\!]$ then $\tau' <: \tau$. For each pair of type τ and τ', we prove that if $[\![\tau]\!] \subseteq [\![\tau']\!]$, then there exists a subtyping rule whose conclusion is $\tau <: \tau'$ and whose premises are a consequence of the induction hypothesis.

– Case $[\![\bot]\!] \subseteq [\![\tau]\!]$. By rule Bottom, we have $\bot <: \tau$.
– Case $[\![\tau \cup \tau']\!] \subseteq [\![\tau'']\!]$. This implies $[\![\tau]\!] \subseteq [\![\tau'']\!]$ and $[\![\tau']\!] \subseteq [\![\tau'']\!]$. Hence, by induction hypothesis, $\tau <: \tau''$ and $\tau' <: \tau''$. Finally, by rule Union-Left, $\tau \cup \tau' <: \tau''$.
– Case $[\![\chi]\!] \subseteq [\![\bot]\!]$. By lemma 6 (Least Semantic Type), the set $[\![\bot]\!]$ does not contain any value. By Lemma 14 (Directed Set), $[\![\chi]\!]$ contains at least one value. Thus, this case is not possible.
– Case $[\![\chi]\!] \subseteq [\![\tau \cup \tau']\!]$. This is a direct corollary of Lemmas 14 (Directed Set) and 8 (Prime when Directed).
– Case $[\![\chi]\!] \subseteq [\![\chi']\!]$ where χ and χ' are distinct constructed types. By Lemmas 14 (Directed Set) and 13 (Homogeneity), constructed types all contain at least a value, and their values are homogeneous. Hence, $[\![\chi]\!]$ contains a value which is not in $[\![\chi']\!]$. This case is not possible.
– Case $[\![\tau_2 \rightarrow \tau_1]\!] \subseteq [\![\tau_4 \rightarrow \tau_3]\!]$. We prove that $[\![\tau_1]\!] \subseteq [\![\tau_3]\!]$ and $[\![\tau_4]\!] \subseteq [\![\tau_2]\!]$. This allow us to conclude by induction hypothesis and rule Function.
  The inclusion $[\![\tau_1]\!] \subseteq [\![\tau_3]\!]$ is a direct consequence of Lemma 17 (Typed Transformations).
  Let us prove that $[\![\tau_4]\!] \subseteq [\![\tau_2]\!]$. It is sufficient to show that $[\![\tau_2]\!]^\perp \subseteq [\![\tau_4]\!]^\perp$. Let c in $[\![\tau_2]\!]^\perp$. By Lemma 11 (Context as Function), $\langle c \rangle \in [\![\tau_2]\!] \boxed{\rightarrow} [\![\tau_1]\!] = [\![\tau_2 \rightarrow \tau_1]\!] \subseteq [\![\tau_4 \rightarrow \tau_3]\!] = [\![\tau_4]\!] \boxed{\rightarrow} [\![\tau_3]\!]$. Hence, by this lemma again, $c \in [\![\tau_4]\!]^\perp$.
– Case $[\![\tau_1 \times \tau_2]\!] \subseteq [\![\tau_3 \times \tau_4]\!]$. By Lemma 17 (Typed Transformations), $[\![\tau_1]\!] \subseteq [\![\tau_3]\!]$ and $[\![\tau_2]\!] \subseteq [\![\tau_4]\!]$. We conclude by induction and 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 $\tau_1 \rightarrow \tau <: \tau_2 \rightarrow \tau$, 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 $\tau_1 \rightarrow \tau$ but not $\tau_2 \rightarrow \tau$. Given that the term e has type τ_2, a natural way to prove that the function e' does not have type $\tau_2 \rightarrow \tau$ 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 $\tau_1 \rightarrow \tau$ 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 $\tau_1 \rightarrow \tau$ 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 18. *For the calculus of Sect. 3, if τ is not a subtype of τ', then there exists a term e in $[\![\tau]\!]$ and a function e' in $[\![\tau' \rightarrow \bot]\!]$ such that $e'\,e \Downarrow$ `error`.*

5 Related Work

This work is a continuation of our work with Melliès on semantic types [4, 3]. 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* [7], that is, sets that are downward closed and closed under directed limits. *Reducibility candidates* [5] are also closed sets of terms. Girard [8] reformulates the candidates as sets of terms closed by biorthogonality in his proof of cut elimination for linear logic. Meanwhile, Krivine [9, 10] has developed a comprehensive framework based on orthogonality, in order to analyze types as *specification* of terms. In semantics, Pitts [11] uses relations closed by biorthogonality to study parametric polymorphism in an operational setting.

Damm [1] studies subtyping for a deterministic calculus with recursive types with union and intersection. He takes a domain theoretic approach based on the ideal model [7]. 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 [12] 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).

6 Extensions and Future Work

Polymorphism and Type Constructors. In an extended version of the paper [2], we present a refined type system with ML-style polymorphism and type constructors and we similarly prove its soundness and completeness. This is omitted here for lack of space.

Strict Pairs and Recursive Types. The type system presented here is not as rich as the type systems of XDuce [13] and CDuce [12] for two reasons. First, for the sake of simplicity, we have not considered recursive types. In previous work [4, 3], we have developed some tools to deal with them. 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.

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 $(\tau_1 \times \tau_3) \cap (\tau_2 \times \tau_4) <: (\tau_1 \cap \tau_2) \times (\tau_3 \cap \tau_4)$. Another issue is that 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. Thus, it is not clear how union and intersection interact as far as subtyping is concerned.

References

1. Damm, F.: Subtyping with union types, intersection types and recursive types II. Research Report 2259, INRIA Rennes (1994)
2. Vouillon, J.: Subtyping union types (extended version). Manuscript; available from `http://www.pps.jussieu.fr/~vouillon/publi/\#union`. (2004)
3. Vouillon, J., Melliès, P.A.: Semantic types: A fresh look at the ideal model for types. In: Proceedings of the 31th ACM Conference on Principles of Programming Languages, Venezia, Italia, ACM Press (2004) 52–63
4. Melliès, P.A., Vouillon, J.: Recursive polymorphic types and parametricity in an operational framework. Preprint PPS//04/06//n°30; available from `http://www.pps.jussieu.fr/~vouillon/publi/\#semtypes2`. (2004)
5. Girard, J.Y.: Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse de doctorat d'État, University of Paris VII (1972)
6. Coq Development Team: The Coq Proof Assistant Reference Manual – Version V7.4. (2003) Available from `http://coq.inria.fr/doc/main.html`.
7. MacQueen, D., Plotkin, G., Sethi, R.: An ideal model for recursive polymorphic types. Information and Control **71** (1986) 95–130
8. Girard, J.Y.: Linear logic. Theoretical Computer Science **50** (1987) 1–102
9. Danos, V., Krivine, J.L.: Disjunctive tautologies and synchronisation schemes. In: Computer Science Logic'00. Volume 1862 of Lecture Notes in Computer Science., Springer (2000) 292–301
10. Krivine, J.L.: Typed lambda-calculus in classical Zermelo-Fraenkel set theory. Archive of Mathematical Logic **40** (2001) 189–205
11. Pitts, A.M.: Parametric polymorphism and operational equivalence. Mathematical Structures in computer Science **10** (2000) 321–359
12. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping. In: 17th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (2002) 137–146
13. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. In: Proceedings of the International Conference on Functional Programming (ICFP). (2000)