Intersection, Universally Quantified, and Reference Types

Mariangiola Dezani-Ciancaglini[1], Paola Giannini[2] and Simona Ronchi Della Rocca[1]

[1] Dipartimento di Informatica, Univ. di Torino, Italy — www.di.unito.it**
[2] Dipartimento di Informatica, Univ. del Piemonte Orientale, Italy — www.di.unipmn.it

Abstract. The aim of this paper is to understand the interplay between intersection, universally quantified, and reference types. Putting together the standard typing rules for intersection, universally quantified, and reference types leads to loss of subject reduction. The problem comes from the invariance of the reference type constructor and the rules of intersection and/or universal quantification elimination, which are subsumption rules. We propose a solution in which types have a kind saying whether the type is (or contains in the case of intersection) a reference type or not. Intersection elimination is limited to intersections not containing reference types, and the reference type constructor can only be applied to closed types. The type assignment is shown to be safe, and when restricted to pure λ-calculus, as expressive as the full standard type assignment system with intersection and universally quantified types.

Introduction

This paper deals with the problem of understanding the interplay between types built using intersection, universal quantification, and reference type constructors. Reference types, [7] and [15], are an essential tool for typing memory locations and the operations of reading and writing in memory. Parametric polymorphism of universally quantified types, introduced by Girard in [5] and Reynolds in [12], enhances the expressivity of typing in a uniform way. Intersection types, introduced in [2], allow for discrete polymorphism, increase the typability, and in particular give a formal account to overloading. Putting together these type constructs is useful for typing in a significant way a programming language with imperative features. It is well known that reference types must be invariant, since they represent both reading and writing of values, and therefore they should be both covariant and contra-variant [10] [page 198]. On the other hand, the standard intersection and universal quantifier elimination typing rules are subsumption rules, since the intersection of two types is contained in both types, and the instantiation of a universally quantified variable specializes the type.

As already remarked in [3] a naive typing with reference and intersection types may lead to loss of subject reduction as the following example shows. We can derive type $\mathtt{Pos}$ for the term

$$(\lambda x.(\lambda y.!x)(x := 0))\,\mathtt{ref}\,1$$

by assuming type $\mathtt{Ref}\,\mathtt{Pos} \wedge \mathtt{Ref}\,\mathtt{Nat}$ for the variable x. In fact $\mathtt{ref}\,1$ has type $\mathtt{Ref}\,\mathtt{Pos} \wedge \mathtt{Ref}\,\mathtt{Nat}$ since 1 is both $\mathtt{Pos}$ and $\mathtt{Nat}$. By intersection elimination we can use:

** Work partially supported by MIUR PRIN'06 EOS DUE, and MIUR PRIN'07 CONCERTO projects. The funding bodies are not responsible for any use that might be made of the results presented here.

– the type $\texttt{Ref}\,\texttt{Nat}$ for x in the typing of $x := 0$ getting the type $\texttt{Unit}$;
– the type $\texttt{Ref}\,\texttt{Pos}$ for x in the typing of $!x$ getting the type $\texttt{Pos}$.

Reducing this term starting from the empty memory, with the call-by-value strategy we get

$$\begin{aligned}(\lambda x.(\lambda y.!x)\ (x := 0))\ \texttt{ref}\,1 \sharp \emptyset &\longrightarrow (\lambda x.(\lambda y.!x)\ (x := 0))\ l \sharp (l = 1)\\ &\longrightarrow (\lambda y.!l)\ (l := 0) \sharp (l = 1)\\ &\longrightarrow (\lambda y.!l)\ () \sharp (l = 0)\\ &\longrightarrow !l \sharp (l = 0)\\ &\longrightarrow 0 \sharp (l = 0)\end{aligned}$$

and 0 does not have the type $\texttt{Pos}$. Note that this term evaluates to 1 in the memory $(l = 1)$ under the call-by-name reduction strategy. So the soundness of typing depends on the evaluation strategy used.

This example is a transcription of an example in [3]. The solution given in [3] is discussed in the conclusion, where it is compared with our proposal.

A variant of this examples shows that also a naive use of universally quantified types may lead to loss of subject reduction. Consider the term:

$$M = ((\lambda x.(\lambda y.!x)\ (x := \lambda z.0))\ (\texttt{ref}\,(\lambda z.z)))\ 1$$

We can derive type $\texttt{Pos}$ for this term by assuming type $\forall t.\texttt{Ref}\,(t \to t)$ for the variable x. In fact, $(\texttt{ref}\,(\lambda z.z))$ can be given type $\forall t.\texttt{Ref}\,(t \to t)$ and therefore also $\texttt{Ref}\,(\texttt{Pos} \to \texttt{Pos})$. By forall elimination we can use:
– the type $\texttt{Ref}\,(\texttt{Nat} \to \texttt{Nat})$ for x in the typing of $x := \lambda z.0$ getting the type $\texttt{Unit}$;
– the type $\texttt{Ref}\,(\texttt{Pos} \to \texttt{Pos})$ for x in the typing of $!x$ getting the type $\texttt{Pos} \to \texttt{Pos}$.

Reducing this term starting from the empty memory, with the call-by-value strategy we get

$$\begin{aligned}M \sharp \emptyset &\longrightarrow ((\lambda x.(\lambda y.!x)\ (x := \lambda z.0))\ l)\ 1 \sharp (l = \lambda z.z)\\ &\longrightarrow ((\lambda y.!l)\ (l := \lambda z.0))\ 1 \sharp (l = \lambda z.z)\\ &\longrightarrow ((\lambda y.!l)())\ 1 \sharp (l = \lambda z.0)\\ &\longrightarrow !l\ 1 \sharp (l = \lambda z.0)\\ &\longrightarrow (\lambda z.0)\ 1 \sharp (l = \lambda z.0)\\ &\longrightarrow 0 \sharp (l = \lambda z.0)\end{aligned}$$

and 0 does not have the type $\texttt{Pos}$. Note that using the call-by-name reduction strategy we get 1.

As suggested by the above examples, a memory location typed by $\texttt{Ref}\,\texttt{Pos} \wedge \texttt{Ref}\,\texttt{Nat}$ must contain values which are both $\texttt{Pos}$ and $\texttt{Nat}$, i.e. values of type $\texttt{Pos} \wedge \texttt{Nat}$. For the case of quantified types a memory location typed by $\forall t.\texttt{Ref}\,(t \to t)$ must contain a function of type $\forall t.t \to t$. This can be better expressed by typing the memory location with the type $\texttt{Ref}\,(\texttt{Pos} \wedge \texttt{Nat})$ in the first case and $\texttt{Ref}\,(\forall t.t \to t)$ in the second. Therefore, when a value is assigned to it it must have type $\texttt{Pos}$ in the first case and $\forall t.t \to t$ in the second.

Building on this idea we propose a type system for a λ-calculus with assignment statements and reference/dereference constructors. Intersections and universally quantified types are assigned to terms, via introduction rules, but elimination of intersection is limited to non reference types, and the $\texttt{Ref}$ type constructor can only be applied to

closed types. We show safety, i.e. subject reduction and progress, of our type system. Lastly we observe that no expressive power is lost in comparison with the original systems of universally quantified types [5], intersection types [2], and both intersection and universally quantified types [8].

A strongly related paper is [4] which proposes a different type assignment system with both reference and intersection types. We will compare the present solution and that one discussed in the paper [4] in the conclusion.

Outline of the paper. Section 1 presents the syntax and reduction rules of the language. Types with their relevant properties are introduced in Section 2, and Section 3 defines the type assignment system and proves its safety. We conclude, in Section 4, by comparing our approach with the ones of [3] and [4], and outlining possible directions for further work.

1 Syntax and Reduction Rules

The language Λ_{imp} we are working with is a simplification of the language in [3], which in its turn belongs to the ML-family, the difference being the lack of the *let* construction and of the binary strings. It is well know that the *let* constructor is syntactic sugar [10] [Section 11.5] and in presence of intersection or universally quantified types does not increase the typability of the language, since with either intersection or universally quantified types we can type the translation of *let* in pure λ-calculus [1], [6]. The only data types of Λ_{imp} are the numerals, that is enough for discussing the typing problems shown in the introduction.

Terms of Λ_{imp} are defined by the following grammar:

$$\begin{array}{rl} M & ::= n \mid x \mid \lambda x.M \mid MM \mid \mathtt{fix}x.M \\ & \quad\; l \mid \mathtt{ref}\,M \mid !M \mid M := M \mid () \\ & \quad\; \mathtt{if}\ M\ \mathtt{then}\ M\ \mathtt{else}\ M \mid M\,\mathtt{op}\,M \mid \ldots \\ n & ::= 0 \mid 1 \mid 2 \mid \ldots \\ \mathtt{op} & ::= + \mid \times \mid \ldots \end{array}$$

where x ranges over a countable set of variables, and l ranges over a countable set of *locations*. Free and bound variables are defined as usual. A term is closed if it does not contain free variables. The set of closed terms is denoted by Λ^0_{imp}.

The syntactical constructs with an imperative operational behaviour are the locations, denoting memory addresses, and the operators `ref`, !, and :=, denoting the operations of allocation, dereferencing, and assignment, whose behaviour is given below. The set of values is the subset of Λ_{imp} defined as follows:

$$V ::= n \mid \lambda x.M \mid l \mid ()$$

The value $()$ is the result of the evaluation of an assignment, whose purpose is the side-effect of changing the store. The store is modeled as a finite association between locations and values:

$$\mu ::= \emptyset \mid \mu, (l = V)$$

On Λ_{imp} we consider a call-by-value reduction semantics. The operational semantics is given by defining reductions inside evaluations contexts, that, as usual, are terms with a *hole*, [ ], specifying which subterm must be reduced.

$$\begin{array}{rl} \mathscr{E} ::= & [\,] \mid \mathscr{E}\,M \mid V\,\mathscr{E} \mid \mathtt{ref}\,\mathscr{E} \mid !\mathscr{E} \mid \mathscr{E} := M \mid V := \mathscr{E} \mid \\ & \mathtt{if}\ \mathscr{E}\ \mathtt{then}\ M\ \mathtt{else}\ M \mid \mathscr{E}\ \mathtt{op}\ M \mid n\ \mathtt{op}\ \mathscr{E} \end{array}$$

$$
\begin{array}{lll}
\mathscr{E}[(\lambda x.M)\ V] \sharp \mu & \longrightarrow \mathscr{E}[M[V/x]] \sharp \mu & (\beta_v) \\
\mathscr{E}[\mathtt{fix}x.M] \sharp \mu & \longrightarrow \mathscr{E}[M[\mathtt{fix}x.M/x]] \sharp \mu & (\mathit{fixR}) \\
\mathscr{E}[\mathtt{ref}\,V] \sharp \mu & \longrightarrow \mathscr{E}[l] \sharp \mu, (l = V) \quad l \text{ fresh} & (\mathit{refR}) \\
\mathscr{E}[!l] \sharp \mu, (l = V) & \longrightarrow \mathscr{E}[V] \sharp \mu, (l = V) & (\mathit{locR}) \\
\mathscr{E}[l := V] \sharp \mu, (l = V') & \longrightarrow \mathscr{E}[()] \sharp \mu, (l = V) & (\mathit{unitR}) \\
\mathscr{E}[\mathtt{if}\ 0\ \mathtt{then}\ M\ \mathtt{else}\ N] \sharp \mu & \longrightarrow \mathscr{E}[M] \sharp \mu & (\mathit{ifZR}) \\
\mathscr{E}[\mathtt{if}\ n\ \mathtt{then}\ M\ \mathtt{else}\ N] \sharp \mu & \longrightarrow \mathscr{E}[N] \sharp \mu \quad n \neq 0 & (\mathit{ifPR}) \\
\mathscr{E}[0 + 0] \sharp \mu & \longrightarrow \mathscr{E}[0] \sharp \mu & (+\mathit{ZZR}) \\
\mathscr{E}[0 + 1] \sharp \mu & \longrightarrow \mathscr{E}[1] \sharp \mu & (+\mathit{ZOR}) \\
\ldots &&
\end{array}
$$

Fig. 1. Reduction Rules

As one can see the evaluation is left to right and for an application we evaluate both terms. The reduction semantics is given by the sets of rules in Fig. 1 where $[N/x]$ is the capture free substitution of x with N, and μ is a store.

2 Types and Type Theory

Types, τ, σ, ρ, are defined by the following syntax:

$$
\begin{array}{lll}
\tau, \sigma, \rho & ::= & \mathtt{Pos} \mid \mathtt{Nat} \mid \mathtt{Unit} \mid t \mid \tau \to \tau \mid \tau \wedge \tau \mid \forall t : \kappa.\tau \mid \mathtt{Ref}\,\tau \\
\kappa & ::= & \mathbf{S} \mid \mathbf{R}
\end{array}
$$

where t belongs to a countable set of type variables (ranged over by t, u, v, w). Kinds (ranged over by κ) say whether the type is (or contains in the case of intersection) a reference type. The *simple kind* **S** is the kind of types which are constants, arrows or intersections of two types both of kind **S**. The *reference kind* **R** is the kind of types which are references, or intersections of two types at least one of them being of kind **R**. An universally quantified type inherits the kind from the type obtained by erasing the quantification.

The type of natural and positive numbers is denoted respectively by `Nat` and `Pos`, `Unit` is the type of assignments and $()$. The arrow constructor, $\tau \to \sigma$, is the type of functions from type τ to type σ, and intersection, $\tau \wedge \sigma$, is the type of expressions that have both type τ and type σ. Universal quantification specifies the kind of the bound variable, since the variable can be replaced only by a type of the same kind. Finally $\mathtt{Ref}\,\tau$ is the type of a reference to a value of type τ. We assume the following precedence relation between constructs: $\forall$, `Ref`, $\wedge$, $\to$. As usual $\to$ associates to the right. We use $\forall \overline{t} : \overline{\kappa}.\tau$ as an abbreviation for $\forall t_1 : \kappa_1 \ldots \forall t_n : \kappa_n.\tau$, where $n \geq 0$.

The set of free type variables of a type, $\mathscr{T}V(\tau)$, is defined in the usual way. A term without occurrences of free type variables is said *closed*.

A *kind environment* Δ is an association between type variables and kinds, defined as follows:

$$
\Delta \vdash \mathtt{Pos} :: \mathbf{S} \qquad \Delta \vdash \mathtt{Nat} :: \mathbf{S} \qquad \Delta \vdash \mathtt{Unit} :: \mathbf{S} \qquad \Delta, t : \kappa \vdash t :: \kappa
$$

$$
\frac{\Delta \vdash \tau :: \kappa \quad \mathscr{T}V(\tau) = \emptyset}{\Delta \vdash \mathtt{Ref}\,\tau :: \mathbf{R}} \qquad
\frac{\Delta \vdash \tau :: \kappa \quad \Delta \vdash \tau' :: \kappa'}{\Delta \vdash \tau \to \tau' :: \mathbf{S}} \qquad
\frac{\Delta \vdash \tau :: \kappa \quad \Delta \vdash \tau' :: \kappa'}{\Delta \vdash \tau \wedge \tau' :: \kappa \curlyvee \kappa'} \qquad
\frac{\Delta, t : \kappa \vdash \tau :: \kappa'}{\Delta \vdash \forall t : \kappa.\tau :: \kappa'}
$$

Fig. 2. Kind Assignment

$$\tau \equiv \tau \wedge \tau \qquad \sigma \wedge \tau \equiv \tau \wedge \sigma \qquad (\tau \wedge \sigma) \wedge \tau' \equiv \tau \wedge (\sigma \wedge \tau')$$

$$(\sigma \to \tau) \wedge (\sigma \to \tau') \equiv \sigma \to \tau \wedge \tau' \qquad \texttt{Ref}\,(\tau \wedge \sigma) \equiv \texttt{Ref}\,\tau \wedge \texttt{Ref}\,\sigma$$

$$\forall t : \kappa.\forall t' : \kappa'.\tau \equiv \forall t' : \kappa'.\forall t : \kappa.\tau$$

$$t \notin \mathscr{T}V(\tau) \quad \Rightarrow \quad \begin{cases} \forall t' : \kappa.\tau \equiv \forall t : \kappa.\tau[t/t'] \\ \forall t : \kappa.\tau \equiv \tau \\ \forall t : \kappa.(\tau \to \sigma) \equiv \tau \to \forall t : \kappa.\sigma \\ \forall t : \kappa.(\tau \wedge \sigma) \equiv \tau \wedge \forall t : \kappa.\sigma \end{cases}$$

Fig. 3. The Congruence $\equiv$ on Types

$$\Delta ::= \emptyset \mid \Delta, t : \kappa \qquad t \notin dom(\Delta)$$

where *dom* is the environment domain.
We use $\Delta, \overline{t} : \overline{\kappa}$ as an abbreviation for the kind environment $\Delta, t_1 : \kappa_1, ..., t_n : \kappa_n$, where $n \geq 0$.

A type τ *has kind* κ *w.r.t.* Δ if the judgment $\Delta \vdash \tau :: \kappa$ can be derived from the rules in Fig . 2. Note that only closed types can be arguments of the `Ref` type constructor. As we can see from the rules of Fig. 2 the kind of an arrow is always **S** and the kind of an intersection is **R** if at least one of its types has kind **R**, since we define:

$$\kappa \curlyvee \kappa' = \begin{cases} \mathbf{S} & \text{if } \kappa = \kappa' = \mathbf{S}, \\ \mathbf{R} & \text{otherwise.} \end{cases}$$

We abbreviate $\Delta \vdash \tau_1 :: \kappa_1 \ldots, \Delta \vdash \tau_n :: \kappa_n$, where $n \geq 0$, by $\Delta \vdash \overline{\tau} :: \overline{\kappa}$.

In the following we will only consider types to which a kind can be assigned from a suitable environment.

On types, we define a congruence relation, $\equiv$, identifying types that denote the same property of terms. The relation $\equiv$ is the minimal equivalence relation which is a congruence and which satisfies the axioms given in Fig. 3. Regarding intersection we have idempotence, commutativity, associativity, distribution of intersection on the right side of arrows with the same left side, and distribution of `Ref` over intersection. For quantified types we have commutativity of quantification, α-conversion, the fact that quantifying on a variable not free in a type is irrelevant, and the standard distribution rules for quantifiers on arrow and intersection connectives. We consider types modulo $\equiv$, so we write $\bigwedge_{i \in I} \tau_i$, and $\bigwedge_{1 \leq i \leq n} \tau_i$ for denoting $\tau_1 \wedge \cdots \wedge \tau_n$, where $I = \{1, \ldots, n\}$, and none of the τ_i, $1 \leq i \leq n$, is an intersection.

It is easy to check that if $\Delta \vdash \tau :: \kappa$ and $\tau \equiv \sigma$, then $\Delta \vdash \sigma :: \kappa$. It is important to notice that $\texttt{Ref}\,\tau$ has a kind implies τ is closed, so in particular $\forall t.\texttt{Ref}\,\tau \equiv \texttt{Ref}\,\tau$.

In the following it is handy to single out the types whose top quantification is meaningless.

Definition 1. *A type* τ *is* $\forall$-top-free *if there are no* t, κ, *and* σ *such that* $\tau \equiv \forall t : \kappa.\sigma$ *and* $t \in \mathscr{T}V(\sigma)$.

For example, $\forall t.Nat$ is $\forall$-top-free, since $\forall t.Nat \equiv Nat$. Instead $Nat \to \forall t.t$ is not $\forall$-top-free, since $Nat \to \forall t.t \equiv \forall t.Nat \to t$.

A preorder relation $\leq$ is defined on types through the rules shown in Fig. 4. Rule

$$\Delta \vdash \texttt{Pos} \leq \texttt{Nat}\ (pos) \qquad \frac{\Delta \vdash \tau \wedge \sigma :: \mathbf{S}}{\Delta \vdash \tau \wedge \sigma \leq \tau}\ (\wedge E) \qquad \frac{\Delta \vdash \forall t : \kappa.\tau :: \kappa' \quad \Delta \vdash \sigma :: \kappa}{\Delta \vdash \forall t : \kappa.\tau \leq \tau[\sigma/t]}\ (\forall E)$$

$$\frac{\Delta \vdash \tau' \leq \tau \quad \Delta \vdash \sigma \leq \sigma'}{\Delta \vdash \tau \to \sigma \leq \tau' \to \sigma'}\ (\to) \qquad \frac{\Delta \vdash \tau \leq \tau' \quad \Delta \vdash \sigma \leq \sigma'}{\Delta \vdash \tau \wedge \sigma \leq \tau' \wedge \sigma'}\ (\wedge) \qquad \frac{\Delta, t : \kappa \vdash \tau \leq \sigma}{\Delta \vdash \forall t : \kappa.\tau \leq \forall t : \kappa.\sigma}\ (\forall)$$

$$\frac{\Delta \vdash \tau :: \kappa}{\Delta \vdash \tau \leq \tau}\ (id) \qquad \frac{\Delta \vdash \tau \leq \rho \quad \Delta \vdash \rho \leq \sigma}{\Delta \vdash \tau \leq \sigma}\ (trans) \qquad \frac{\tau \equiv \tau' \quad \Delta \vdash \tau' \leq \sigma' \quad \sigma' \equiv \sigma}{\Delta \vdash \tau \leq \sigma}\ (congr)$$

Fig. 4. The Preorder Relation $\leq$ on Types

(pos) says that a positive is also a natural. Rules ($\wedge E$) and ($\forall E$) are the elimination rules. Note that for eliminating intersection we require that the intersection does not contain reference types. This is a crucial restriction, along with the facts that `Ref` can only be applied to closed types and there is no rule for applying $\leq$ inside `Ref`, to get subject reduction. Rules ($\to$), ($\wedge$), and ($\forall$) extend $\leq$ to the specific constructor, and they are standard. Rule (id), and ($trans$), make $\leq$ a preorder, and rule ($congr$) makes $\leq$ a partial order when we identify congruent types.

Note that $\Delta \vdash \tau :: \kappa$ and $\tau \equiv \sigma$ imply both $\Delta \vdash \tau \leq \sigma$ and $\Delta \vdash \sigma \leq \tau$. On the other side $\Delta \vdash \tau \leq \sigma$ implies $\Delta \vdash \tau :: \kappa$ and $\Delta \vdash \sigma :: \kappa$ for some κ.

Weakening holds for kind environments in all the considered judgements, i.e. $\Delta \vdash \tau :: \kappa$ implies $\Delta, t : \kappa \vdash \tau :: \kappa$ if $t \notin dom(\Delta)$ and similarly for $\Delta \vdash \tau \leq \sigma$.

By induction on the definition of $\leq$ we can show that the preorder is preserved by replacing type variables by types of the same kinds.

Lemma 1. *$\Delta, t : \kappa \vdash \tau \leq \sigma$, and $\Delta \vdash \rho :: \kappa$ imply $\Delta \vdash \tau[\rho/t] \leq \sigma[\rho/t]$.*

The next technical lemma is the key tool for proving the subject reduction property in case the used reduction rule is the β_v-rule. It states that if a quantified intersection of arrows is less than the arrow $\tau \to \sigma$, then there are instances of domains and co-domains of some arrows in the intersection which are related by the preorder to τ and σ.

Lemma 2. *If $\Delta \vdash \forall \overline{t} : \overline{\kappa}. \bigwedge_{i \in I}(\tau_i \to \sigma_i) \leq \tau \to \sigma$, where σ_i $(i \in I)$ and σ are $\forall$-top-free, then there are $\overline{\rho}$, and $J \subseteq I$, such that:*

- *$\Delta \vdash \overline{\rho} :: \overline{\kappa}$,*
- *$\Delta \vdash \tau \leq \tau_j[\overline{\rho}/\overline{t}]$ $(j \in J)$, and*
- *$\Delta \vdash \bigwedge_{j \in J} \sigma_j[\overline{\rho}/\overline{t}] \leq \sigma$.*

Proof. By induction on the definition of $\leq$. In order to prove the result for rule ($trans$) we show the more general assert that follows:
If $\tau \equiv \forall \overline{u} : \overline{\kappa}. \bigwedge_{i \in I}(\tau_i \to \sigma_i)$ where σ_i $(i \in I)$ are $\forall$-top-free, and $\Delta \vdash \tau \leq \sigma$, then there are $\overline{v}$, $\overline{\kappa'}$, J, τ'_j, $\forall$-top-free σ'_j $(j \in J)$, $\overline{\rho}$ such that:

- *$\sigma \equiv \forall \overline{v} : \overline{\kappa'}. \bigwedge_{j \in J}(\tau'_j \to \sigma'_j)$*
- *$\Delta, \overline{v} : \overline{\kappa'} \vdash \overline{\rho} :: \overline{\kappa}$, and*
- *for all $j \in J$ there is $H_j \subseteq I$ with:*
  - *$\Delta, \overline{v} : \overline{\kappa'} \vdash \tau'_j \leq \tau_h[\overline{\rho}/\overline{u}]$ for all $h \in H_j$, and*
  - *$\Delta, \overline{v} : \overline{\kappa'} \vdash \bigwedge_{h \in H_j} \sigma_h[\overline{\rho}/\overline{u}] \leq \sigma'_j$.*

The proof is by induction on the derivation of $\leq$. We only consider the most difficult case, that is when the statement is the consequent of rule $(trans)$. Let $\Delta \vdash \tau \leq \tau'$ and $\Delta \vdash \tau' \leq \sigma$ be the premises of the application of rule $(trans)$.

By induction hypothesis on $\tau \leq \tau'$ there are $\overline{w}$, $\overline{\kappa''}$, L, τ''_l, $\forall$-top-free σ''_l ($l \in L$), $\overline{\rho'}$ such that:

(a) $\tau' \equiv \forall \overline{w} : \overline{\kappa''}. \bigwedge_{l \in L} (\tau''_l \to \sigma''_l)$

(b) $\Delta, \overline{w} : \overline{\kappa''} \vdash \overline{\rho'} :: \overline{\kappa}$, and

(c) for all $l \in L$ there is $H'_l \subseteq I$ with:

$(c.1)$ $\Delta, \overline{w} : \overline{\kappa''} \vdash \tau''_l \leq \tau_h[\overline{\rho'}/\overline{u}]$ for all $h \in H'_l$, and

$(c.2)$ $\Delta, \overline{w} : \overline{\kappa''} \vdash \bigwedge_{h \in H'_l} \sigma_h[\overline{\rho'}/\overline{u}] \leq \sigma''_l$.

By induction hypothesis on $\tau' \leq \sigma$ there are $\overline{v}$, $\overline{\kappa'}$, J, τ'_j, $\forall$-top-free σ'_j ($j \in J$), $\overline{\rho''}$ such that:

(a') $\sigma \equiv \forall \overline{v} : \overline{\kappa'}. \bigwedge_{j \in J} (\tau'_j \to \sigma'_j)$

(b') $\Delta, \overline{v} : \overline{\kappa'} \vdash \overline{\rho''} :: \overline{\kappa''}$, and

(c') for all $j \in J$ there is $H''_j \subseteq L$ with:

$(c'.1)$ $\Delta, \overline{v} : \overline{\kappa'} \vdash \tau'_j \leq \tau''_h[\overline{\rho''}/\overline{w}]$ for all $h \in H''_j$, and

$(c'.2)$ $\Delta, \overline{v} : \overline{\kappa'} \vdash \bigwedge_{h \in H''_j} \sigma''_h[\overline{\rho''}/\overline{w}] \leq \sigma'_j$.

Note that we can assume that the sets of variables $\overline{u}$, $\overline{v}$, and $\overline{w}$ are fresh and pairwise disjoint. Define $\overline{\rho} = \overline{\rho'[\overline{\rho''}/\overline{w}]}$ and $H_j = \bigcup_{k \in H''_j} H'_k$ $(j \in J)$. It is easy to verify that:

- $\Delta, \overline{v} : \overline{\kappa'} \vdash \overline{\rho} :: \overline{\kappa}$ (from Lemma 1, weakening, (b) and (b')), and
- $H_j \subseteq I$.

Moreover $(c.1)$ and (b') imply by Lemma 1 and weakening that

$$\Delta, \overline{v} : \overline{\kappa'} \vdash \tau''_l[\overline{\rho''}/\overline{w}] \leq \tau_h[\overline{\rho'}/\overline{u}][\overline{\rho''}/\overline{w}] \text{ for all } h \in H'_l.$$

Note that $\tau_h[\overline{\rho'}/\overline{u}][\overline{\rho''}/\overline{w}] = \tau_h[\overline{\rho}/\overline{u}]$ for all $h \in H_j$ since $\overline{w}$ cannot occur in τ_h. So by $(c'.1)$, and transitivity of $\leq$ we get for all $j \in J$:

$$\Delta, \overline{v} : \overline{\kappa'} \vdash \tau'_j \leq \tau_h[\overline{\rho}/\overline{u}] \text{ for all } h \in H_j.$$

Similarly from $(c.2)$, (b'), Lemma 1 and weakening we get

$$\Delta, \overline{v} : \overline{\kappa'} \vdash \bigwedge_{h \in H'_l} \sigma_h[\overline{\rho}/\overline{u}] \leq \sigma''_l[\overline{\rho''}/\overline{w}] \text{ for all } l \in L.$$

This together with $(c'.2)$, using rule $(\wedge)$, transitivity of $\leq$, and the congruence $\sigma \wedge \sigma \equiv \sigma$ implies for all $j \in J$:

$$\Delta, \overline{v} : \overline{\kappa'} \vdash \bigwedge_{h \in H_j} \sigma_h[\overline{\rho}/\overline{u}] \leq \sigma'_j. \quad \square$$

3 The Typing System

The typing system proves judgements of the shape:

$$\Delta; \Sigma; \Gamma \vdash M : \tau$$

where Δ is a kind environment, Σ and Γ are a *store environment* and a *type environment* respectively, M is a term and τ is a type. Store and type environments are defined as follows:

$$\begin{array}{lll} \Sigma ::= \emptyset \mid \Sigma, l : \tau & l \notin dom(\Sigma) & \tau \text{ is a closed type} \\ \Gamma ::= \emptyset \mid \Gamma, x : \tau & x \notin dom(\Gamma). & \end{array}$$

A store (type) environment is *well formed* with respect to a kind environment Δ if all its predicates have a kind, i.e., Σ (Γ) is such that if $l : \tau \in \Sigma$ ($x : \tau \in \Gamma$) then

$$\Delta;\Sigma;\Gamma \vdash 0 : \mathtt{Nat}\ \ (\mathtt{Nat}) \qquad \frac{n \neq 0}{\Delta;\Sigma;\Gamma \vdash n : \mathtt{Pos}}\ (\mathtt{Pos}) \qquad \Delta;\Sigma;\Gamma \vdash () : \mathtt{Unit}\ \ (\mathtt{Unit}_{()})$$

$$\Delta;\Sigma;\Gamma, x:\tau \vdash x:\tau\ \ (\mathtt{var}) \qquad\qquad \Delta;\Sigma, l:\tau;\Gamma \vdash l : \mathtt{Ref}\,\tau\ \ (\mathtt{loc})$$

$$\frac{\Delta;\Sigma;\Gamma \vdash M:\tau \quad \mathscr{TV}(\tau)=\emptyset}{\Delta;\Sigma;\Gamma \vdash \mathtt{ref}\,M : \mathtt{Ref}\,\tau}\ (\mathtt{Ref}\,I) \qquad \frac{\Delta;\Sigma;\Gamma \vdash M : \mathtt{Ref}\,\tau}{\Delta;\Sigma;\Gamma \vdash !M : \tau}\ (\mathtt{Ref}\,E)$$

$$\frac{\Delta;\Sigma;\Gamma, x:\tau \vdash M:\sigma}{\Delta;\Sigma;\Gamma \vdash \lambda x.M : \tau\to\sigma}\ (\to I) \qquad \frac{\Delta;\Sigma;\Gamma \vdash M:\tau\to\sigma \quad \Delta;\Sigma;\Gamma \vdash N:\tau}{\Delta;\Sigma;\Gamma \vdash MN:\sigma}\ (\to E)$$

$$\frac{\Delta, t:\kappa;\Sigma;\Gamma \vdash M:\tau \quad t \notin \mathscr{TV}(\Sigma,\Gamma)}{\Delta;\Sigma;\Gamma \vdash M : \forall t:\kappa.\tau}\ (\forall I) \qquad \frac{\Delta;\Sigma;\Gamma \vdash M:\tau \quad \Delta;\Sigma;\Gamma \vdash M:\sigma}{\Delta;\Sigma;\Gamma \vdash M : \tau\wedge\sigma}\ (\wedge I)$$

$$\frac{\Delta;\Sigma;\Gamma \vdash M:\tau \quad \Delta \vdash \tau \leq \sigma}{\Delta;\Sigma;\Gamma \vdash M:\sigma}\ (\leq) \qquad \frac{\Delta;\Sigma;\Gamma \vdash M : \mathtt{Ref}\,\tau \quad \Delta;\Sigma;\Gamma \vdash N:\tau}{\Delta;\Sigma;\Gamma \vdash M := N : \mathtt{Unit}}\ (\mathtt{Unit})$$

$$\frac{\Delta;\Sigma;\Gamma, x:\tau \vdash M:\tau}{\Delta;\Sigma;\Gamma \vdash \mathtt{fix}\,x.M:\tau}\ (\mathtt{fix}) \qquad \frac{\Delta;\Sigma;\Gamma \vdash M : \mathtt{Nat} \quad \Delta;\Sigma;\Gamma \vdash N_1:\tau \quad \Delta;\Sigma;\Gamma \vdash N_2:\tau}{\Delta;\Sigma;\Gamma \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : \tau}\ (\mathtt{if})$$

Fig. 5. The Typing Rules for Terms

$\Delta \vdash \tau :: \kappa$ for some kind κ. When we write a typing judgement $\Delta;\Sigma;\Gamma \vdash M : \tau$ we always assume that Σ and Γ are well formed with respect to Δ.

The typing rules, given in Fig. 5, are standard. We omit the typing rules dealing with arithmetic operators which are obvious. Note that the elimination rules of both $\wedge$ and $\forall$ are particular cases of rule $(\leq)$.

It is easy to verify that strengthening and weakening for all the environments are admissible rules. Fig. 6 shows these rules, where $\mathscr{L}(M)$ is the set of locations and $\mathscr{FV}(M)$ is the set of free variables occurring in M.

The proof that deductions remain valid under the substitution of type variables by types respecting kinds by induction on deductions is standard.

Proposition 1. *If* $\Delta, t:\kappa;\Sigma;\Gamma \vdash M:\tau$ *and* $\Delta \vdash \sigma :: \kappa$, *then*
$$\Delta;\Sigma[\sigma/t];\Gamma[\sigma/t] \vdash M : \tau[\sigma/t].$$

The type system enjoys a Generation Lemma, which relates the shapes of terms with the shapes of their possible derivations. We omit the obvious points concerning numerals and operators on numerals.

Lemma 3 (Generation). *Let* $\Delta;\Sigma;\Gamma \vdash M:\tau$. *Then* $\Delta \vdash \tau \geq \forall \overline{t}:\overline{\kappa}.\bigwedge_{i\in I}\tau_i$, *for some* I, $\overline{t}$, $\overline{\kappa}$, $\forall$*-top-free* τ_i $(i \in I)$, *and the followings hold, where* $\Delta' = \Delta, \overline{t}:\overline{\kappa}$:

1. $\underline{M = x}$ *implies that* $x:\sigma \in \Gamma$ *for some* σ *such that* $\sigma \leq \tau$;
2. $\underline{M = \lambda x.P}$ *implies that there are* σ_i, ρ_i $(i \in I)$, *such that:*
   (a) $\tau_i = \sigma_i \to \rho_i$, *and*
   (b) $\Delta';\Sigma;\Gamma, x:\sigma_i \vdash P : \rho_i$ $(i \in I)$;
3. $\underline{M = PN}$ *implies that there are* σ_i $(i \in I)$ *such that:*

$$\frac{\Delta, t:\kappa; \Sigma; \Gamma \vdash M:\tau \quad t \notin \mathscr{T}V(\Sigma,\Gamma,\tau)}{\Delta; \Sigma; \Gamma \vdash M:\tau}\ (s\Delta) \qquad \frac{\Delta; \Sigma; \Gamma \vdash M:\tau \quad t \notin \mathscr{T}V(\Sigma,\Gamma,\tau)}{\Delta, t:\kappa; \Sigma; \Gamma \vdash M:\tau}\ (w\Delta)$$

$$\frac{\Delta; \Sigma, l:\tau'; \Gamma \vdash M:\tau \quad l \notin \mathscr{L}(M)}{\Delta; \Sigma; \Gamma \vdash M:\tau}\ (s\Sigma) \qquad \frac{\Delta; \Sigma; \Gamma \vdash M:\tau \quad l \notin dom(\Sigma) \quad \Delta \vdash \sigma :: \kappa \quad \text{for some } \kappa \quad \mathscr{T}V(\sigma) = \emptyset}{\Delta; \Sigma, l:\sigma; \Gamma \vdash M:\tau}\ (w\Sigma)$$

$$\frac{\Delta; \Sigma; \Gamma, x:\tau' \vdash M:\tau \quad x \notin \mathscr{T}V(M)}{\Delta; \Sigma; \Gamma \vdash M:\tau}\ (s\Gamma) \qquad \frac{\Delta; \Sigma; \Gamma \vdash M:\tau \quad x \notin dom(\Gamma) \quad \Delta \vdash \sigma :: \kappa \quad \text{for some } \kappa}{\Delta; \Sigma; \Gamma, x:\sigma \vdash M:\tau}\ (w\Gamma)$$

Fig. 6. Admissible Rules

   (a) $\Delta'; \Sigma; \Gamma \vdash P : \sigma_i \to \tau_i$ $(i \in I)$, *and*
   (b) $\Delta'; \Sigma; \Gamma \vdash N : \sigma_i$ $(i \in I)$;
4. $\underline{M = \mathtt{fix}x.N}$ *implies that* $\Delta'; \Sigma; \Gamma, x:\tau_i \vdash N : \tau_i$ $(i \in I)$;
5. $\underline{M = l}$ *implies that* $l:\sigma \in \Sigma$, *for some closed* σ *such that* $\mathtt{Ref}\,\sigma \equiv \tau$;
6. $\underline{M = !N}$ *implies that* $\Delta'; \Sigma; \Gamma \vdash N : \mathtt{Ref}\,\tau_i$ $(i \in I)$;
7. $\underline{M = \mathtt{ref}\,N}$ *implies that there are closed* σ_i *such that:*
   (a) $\tau_i = \mathtt{Ref}\,\sigma_i$, *and*
   (b) $\Delta; \Sigma; \Gamma \vdash N : \sigma_i$ $(i \in I)$;
8. $\underline{M = P := N}$ *implies* $\tau \equiv \mathtt{Unit}$, *and for some closed* σ *we have* $\Delta'; \Sigma; \Gamma \vdash P : \mathtt{Ref}\,\sigma$ *and* $\Delta'; \Sigma; \Gamma \vdash N : \sigma$;
9. $\underline{M = ()}$ *implies* $\tau \equiv \mathtt{Unit}$;
10. $\underline{M = \mathtt{if}\ P\ \mathtt{then}\ N\ \mathtt{else}\ N'}$ *implies that* $\Delta'; \Sigma; \Gamma \vdash P : \mathtt{Nat}$ *and* $\Delta'; \Sigma; \Gamma \vdash N : \tau_i$ *and* $\Delta'; \Sigma; \Gamma \vdash N' : \tau_i$ $(i \in I)$.

Proof. For all points, the proof is by induction on derivations. We will consider only the case in which the last rule applied is $(\wedge I)$, and we will show it for Points 2, 3 and 5. All the other cases are simpler.

2. If the last applied rule is:
$$\frac{\Delta; \Sigma; \Gamma \vdash \lambda x.P : \tau \quad \Delta; \Sigma; \Gamma \vdash \lambda x.P : \tau'}{\Delta; \Sigma; \Gamma \vdash \lambda x.P : \tau \wedge \tau'}\ (\wedge I)$$
by induction $\Delta \vdash \tau \geq \forall \overline{t} : \overline{\kappa}. \bigwedge_{i\in I}(\sigma_i \to \rho_i)$, and $\Delta \vdash \tau' \geq \forall \overline{t'} : \overline{\kappa'}. \bigwedge_{j\in J}(\sigma'_j \to \rho'_j)$, and $\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma, x:\sigma_i \vdash P : \rho_i$, for $i \in I$ and $\Delta, \overline{t'} : \overline{\kappa'}; \Sigma; \Gamma, x:\sigma'_j \vdash P : \rho'_j$, for $j \in J$. By the monotonicity of $\leq$ with respect to $\wedge$ we get
$$\tau \wedge \tau' \geq \forall \overline{t} : \overline{\kappa}. \bigwedge_{i\in I}(\sigma_i \to \rho_i) \wedge \forall \overline{t'} : \overline{\kappa'}. \bigwedge_{j\in J}(\sigma'_j \to \rho'_j)$$
and, since types are considered modulo $\equiv$, we can assume that $\overline{t}$ and $\overline{t'}$ are disjoint, so we have $\tau \wedge \tau' \geq \forall \overline{t} : \overline{\kappa}. \forall \overline{t'} : \overline{\kappa'}. (\bigwedge_{i\in I}(\sigma_i \to \rho_i) \wedge \bigwedge_{j\in J}(\sigma'_j \to \rho'_j))$. Moreover, by the admissible rule $(w\Delta)$, we obtain $\Delta, \overline{t} : \overline{\kappa}, \overline{t'} : \overline{\kappa'}; \Sigma; \Gamma, x:\sigma_i \vdash P : \rho_i$, for $i \in I$ and $\Delta, \overline{t} : \overline{\kappa}, \overline{t'} : \overline{\kappa'}; \Sigma; \Gamma, x:\sigma'_j \vdash P : \rho'_j$, for $j \in J$.

3. If the last used rule is:
$$\frac{\Delta; \Sigma; \Gamma \vdash PN : \tau \quad \Delta; \Sigma; \Gamma \vdash PN : \tau'}{\Delta; \Sigma; \Gamma \vdash PN : \tau \wedge \tau'}\ (\wedge I)$$
then by induction, $\Delta \vdash \tau \geq \forall \overline{t} : \overline{\kappa}. \bigwedge_{i\in I} \tau_i$ and $\Delta \vdash \tau' \geq \forall \overline{t'} : \overline{\kappa'}. \bigwedge_{j\in J} \tau'_j$, and:

$\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma \vdash P : \sigma_i \to \tau_i$ and $\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma \vdash N : \sigma_i$ $(i \in I)$
$\Delta, \overline{t'} : \overline{\kappa'}; \Sigma; \Gamma \vdash P : \sigma'_j \to \tau'_j$ and $\Delta, \overline{t'} : \overline{\kappa'}; \Sigma; \Gamma \vdash N : \sigma'_j$ $(j \in J)$.
Then, by $(w\Delta)$:
$\Delta, \overline{t} : \overline{\kappa}, \overline{t'} : \overline{\kappa'}; \Sigma; \Gamma \vdash P : \sigma_i \to \tau_i$ and $\Delta, \overline{t} : \overline{\kappa}, \overline{t'} : \overline{\kappa'}; \Sigma; \Gamma \vdash N : \sigma_i$ $(i \in I)$
$\Delta, \overline{t} : \overline{\kappa}, \overline{t'} : \overline{\kappa'}; \Sigma; \Gamma \vdash P : \sigma'_j \to \tau'_j$ and $\Delta, \overline{t} : \overline{\kappa}, \overline{t'} : \overline{\kappa'}; \Sigma; \Gamma \vdash N : \sigma'_j$ $(i \in J)$.
The proof follows from $\Delta \vdash \tau \wedge \tau' \geq \forall \overline{t} : \overline{\kappa}. \forall \overline{t'} : \overline{\kappa'}. (\bigwedge_{i \in I} \tau_i \wedge \bigwedge_{i \in J} \tau'_j)$, since we can assume that $\overline{t}$ and $\overline{t'}$ are disjoint.

5. If the last applied rule is:
$$\frac{\Delta; \Sigma; \Gamma \vdash l : \tau \quad \Delta; \Sigma; \Gamma \vdash l : \tau'}{\Delta; \Sigma; \Gamma \vdash l : \tau \wedge \tau'} \ (\wedge I)$$
by induction $\tau \equiv \mathtt{Ref}\, \sigma$ and $\tau' \equiv \mathtt{Ref}\, \sigma'$, and the proof follows from the fact that $\mathtt{Ref}\, \sigma \wedge \mathtt{Ref}\, \sigma' \equiv \mathtt{Ref}\, (\sigma \wedge \sigma')$. $\square$

Note that, without reference types, Points 3 and 4 of previous lemma hold with I a singleton set. The restriction on rule $(\wedge E)$ is reflected in the necessity of having sets of types of cardinality bigger than 1. For example from $\{x : (\mathtt{Nat} \to \mathtt{Nat}) \wedge (\mathtt{Nat} \to \mathtt{Ref\,Nat}), y : \mathtt{Nat}, z : (\mathtt{Nat} \to \mathtt{Nat}) \wedge (\mathtt{Ref\,Nat} \to \mathtt{Nat} \to \mathtt{Nat})\}$ we can derive $z(xy) : \mathtt{Nat} \wedge (\mathtt{Nat} \to \mathtt{Nat})$, but there are no types τ_1, τ_2 such that from the same environment we can derive $z : \tau_1 \to \tau_2$ and $xy : \tau_1$. Instead we have $\emptyset; \emptyset; \{x : (\sigma \to \sigma) \wedge (\sigma \to \tau), y : \sigma, z : (\sigma \to \sigma) \wedge (\tau \to \sigma \to \sigma)\} \vdash z : \sigma \wedge \tau \to \sigma \wedge (\sigma \to \tau)$ and $\emptyset; \emptyset; \{x : (\sigma \to \sigma) \wedge (\sigma \to \tau), y : \sigma, z : (\sigma \to \sigma) \wedge (\tau \to \sigma \to \sigma)\} \vdash x : \sigma \wedge \tau$ for all σ, τ of kind $\mathbf{S}$. Similarly we can derive $\emptyset; \emptyset; \{y : (\mathtt{Ref\,Nat} \to \mathtt{Ref\,Nat}) \wedge (\mathtt{Ref}\,(\mathtt{Nat} \to \mathtt{Nat}) \to \mathtt{Ref}\,(\mathtt{Nat} \to \mathtt{Nat}))\} \vdash \mathtt{fix}\, x.yx : \mathtt{Ref}\,(\mathtt{Nat} \wedge (\mathtt{Nat} \to \mathtt{Nat}))$, but we cannot derive $\emptyset; \emptyset; \{x : \mathtt{Ref}\,(\mathtt{Nat} \wedge (\mathtt{Nat} \to \mathtt{Nat})), y : (\mathtt{Ref\,Nat} \to \mathtt{Ref\,Nat}) \wedge (\mathtt{Ref}\,(\mathtt{Nat} \to \mathtt{Nat}) \to \mathtt{Ref}\,(\mathtt{Nat} \to \mathtt{Nat}))\} \vdash yx : \mathtt{Ref}\,(\mathtt{Nat} \wedge (\mathtt{Nat} \to \mathtt{Nat}))$.

The typing system enjoys the standard Substitution Property, that can be proved by induction on derivations.

Lemma 4 (Substitution). *If $\Delta; \Sigma; \Gamma, x : \tau \vdash M : \sigma$ and $\Delta; \Sigma; \Gamma \vdash N : \tau$, then $\Delta; \Sigma; \Gamma \vdash M[N/x] : \sigma$.*

In order to prove subject reduction for our type system we need to show that typing is preserved under the replacement of a type by a smaller one in the type environment.

Lemma 5. *Let $\Delta; \Sigma; \Gamma, x : \sigma \vdash M : \tau$ and $\Delta \vdash \sigma' \leq \sigma$. Then $\Delta; \Sigma; \Gamma, x : \sigma' \vdash M : \tau$.*

The *agreement* between a store environment and a store is defined as usual [10] [Definition 13.5.1].

Definition 2. *We say that a store environment Σ agrees with a store μ (notation $\Sigma \vdash \mu$) if:*
- *$(l = V) \in \mu$ implies $l : \tau \in \Sigma$ and $\Delta; \Sigma; \emptyset \vdash V : \tau$ for some τ;*
- *$l : \tau \in \Sigma$ implies $(l = V) \in \mu$ and $\Delta; \Sigma; \emptyset \vdash V : \tau$ for some V.*

Now we can prove subject reduction.

Theorem 1 (Subject Reduction). *$\Delta; \Sigma; \Gamma \vdash M : \tau$ and $\Sigma \vdash \mu$ and $M \sharp \mu \longrightarrow N \sharp \mu'$ imply $\Delta; \Sigma'; \Gamma \vdash N : \sigma$ and $\Sigma' \vdash \mu'$ for some $\Sigma' \supseteq \Sigma$.*

Proof. $M \sharp \mu \longrightarrow N \sharp \mu'$ implies that $M = \mathscr{E}[M']$ and $N = \mathscr{E}[N']$, for some evaluation context $\mathscr{E}$. The proof is given by induction on $\mathscr{E}$. We consider the most interesting cases for $\mathscr{E} = [\,]$ since the induction cases are straightforward.

If the rule applied is (β_v), then $M = (\lambda x.P)V$, and $N = P[V/x]$. From Lemma 3(3), for some $\overline{t}$, $\overline{\kappa}$, I, σ_i and $\forall$-top-free τ_i $(i \in I)$,

(1) $\Delta \vdash \forall \overline{t} : \overline{\kappa}. \bigwedge_{i\in I} \tau_i \leq \tau$,

(2) $\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma \vdash \lambda x.P : \sigma_i \to \tau_i$ $(i \in I)$,

(3) $\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma \vdash V : \sigma_i$ $(i \in I)$,

From Lemma 3(2), and Point (2), for all $i \in I$, there are $\overline{\nu^{(i)}}$, $\overline{\kappa^{(i)}}$, H_i, $\sigma_j^{(i)}$, and $\tau_j^{(i)}$ $(j \in H_i)$ such that $\sigma_j^{(i)} \to \tau_j^{(i)}$ is $\forall$-top-free and

(a) $\Delta, \overline{t} : \overline{\kappa} \vdash \forall \overline{\nu^{(i)}} : \overline{\kappa^{(i)}}. \bigwedge_{j\in H_i} (\sigma_j^{(i)} \to \tau_j^{(i)}) \leq \sigma_i \to \tau_i$,

(b) $\Delta, \overline{t} : \overline{\kappa}, \overline{\nu^{(i)}} : \overline{\kappa^{(i)}}; \Sigma; \Gamma, x : \sigma_j^{(i)} \vdash P : \tau_j^{(i)}$ $(j \in H_i)$.

Note that $\sigma_j^{(i)} \to \tau_j^{(i)}$ $\forall$-top-free implies $\tau_j^{(i)}$ $\forall$-top-free. Then Lemma 2 and Point (a) imply that there are $\overline{\rho^{(i)}}$, and $J_i \subseteq H_i$, such that:

(α) $\Delta, \overline{t} : \overline{\kappa} \vdash \overline{\rho^{(i)}} :: \overline{\kappa^{(i)}}$ $(i \in I)$,

(β) $\Delta, \overline{t} : \overline{\kappa} \vdash \sigma_i \leq \sigma_j^{(i)}[\overline{\rho^{(i)}}/\overline{\nu^{(i)}}]$ $(j \in J_i)$, and

(γ) $\Delta, \overline{t} : \overline{\kappa} \vdash \bigwedge_{j\in J_i} \tau_j^{(i)}[\overline{\rho^{(i)}}/\overline{\nu^{(i)}}] \leq \tau_i$.

From Points (b), (α), and Proposition 1, for all $j \in J_i$, we derive

$$\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma, x : \sigma_j^{(i)}[\overline{\rho^{(i)}}/\overline{\nu^{(i)}}] \vdash P : \tau_j^{(i)}[\overline{\rho^{(i)}}/\overline{\nu^{(i)}}]$$

and by Point (β), and Lemma 5 we get:

$$\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma, x : \sigma_i \vdash P : \tau_j^{(i)}[\overline{\rho^{(i)}}/\overline{\nu^{(i)}}].$$

Applying rules $(\wedge I)$, $(\leq)$ by Point (γ) we derive

$$\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma, x : \sigma_i \vdash P : \tau_i$$

which by Lemma 4, and Point (3), implies that $\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma \vdash P[V/x] : \tau_i$ for all $i \in I$. With multiple applications of rule $(\wedge I)$ we get $\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma \vdash P[V/x] : \bigwedge_{i\in I} \tau_i$, and then applying many times rule $(\forall I)$ (note that we can assume $\overline{t} \notin \mathscr{T}V(\Sigma, \Gamma)$ since we take types modulo $\equiv$) we derive $\Delta; \Sigma; \Gamma \vdash P[V/x] : \forall \overline{t} : \overline{\kappa}. \bigwedge_{i\in I} \tau_i$. Finally from Point (1) and rule $(\leq)$ we conclude $\Delta; \Sigma; \Gamma \vdash P[V/x] : \tau$.

If the rule applied is (*fixR*), then $M = \mathtt{fix}x.P$, and $N = P[\mathtt{fix}x.P/x]$. From Lemma 3(4), for some $\overline{t}$, $\overline{\kappa}$, I, τ_i $(i \in I)$, we get $\Delta \vdash \forall \overline{t} : \overline{\kappa}. \bigwedge_{i\in I} \tau_i \leq \tau$ and $\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma, x : \tau_i \vdash P : \tau_i$ $(i \in I)$. Therefore, rule $(\mathtt{fix})$ of Fig. 5 implies $\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma \vdash \mathtt{fix}x.P : \tau_i$ $(i \in I)$. From the Substitution Lemma 4 we derive $\Delta, \overline{t} : \overline{\kappa}; \Sigma; \Gamma \vdash P[\mathtt{fix}x.P/x] : \tau_i$ $(i \in I)$. Applying $(\wedge I)$'s, $(\forall I)$'s, and $(\leq)$ we conclude $\Delta; \Sigma; \Gamma \vdash P[\mathtt{fix}x.P/x] : \tau$.

If the rule applied is (*refR*), then $M = \mathtt{ref}\, V$, $N = l$, and $\mu' = \mu, (l = V)$. From Lemma 3(7), for some $\overline{t}$, $\overline{\kappa}$, I, closed τ_i $(i \in I)$, we get $\Delta \vdash \forall \overline{t} : \overline{\kappa}. \bigwedge_{i\in I} \mathtt{Ref}\, \tau_i \leq \tau$ and $\Delta; \Sigma; \Gamma \vdash V : \tau_i$ $(i \in I)$. Let $\Sigma' = \Sigma, l : \bigwedge_{i\in I} \tau_i$, we have that $\Delta; \Sigma'; \Gamma \vdash l : \mathtt{Ref} \bigwedge_{i\in I} \tau_i$. Therefore, since τ_i are closed, we have that $\mathtt{Ref} \bigwedge_{i\in I} \tau_i \equiv \forall \overline{t} : \overline{\kappa}. \bigwedge_{i\in I} \mathtt{Ref}\, \tau_i$. Applying rule $(\leq)$ we conclude $\Delta; \Sigma'; \Gamma \vdash l : \tau$. From $\Sigma \vdash \mu$ and $\Delta; \Sigma; \Gamma \vdash V : \bigwedge_{i\in I} \tau_i$ we also get $\Sigma' \vdash \mu'$.

If the rule applied is (*locR*), then result derives directly from the fact that $\Sigma \vdash \mu$.

If the rule applied is (*unitR*), then $M = l := V$, $\mu = \mu'', (l = V')$, and $N = ()$, $\mu' = \mu'', (l = V)$. From Lemma 3(8), $\tau \equiv \mathtt{Unit}$, and for some closed σ we have $\Delta; \Sigma; \Gamma \vdash l : \mathtt{Ref}\, \sigma$, and $\Delta; \Sigma; \Gamma \vdash V : \sigma$. The typing rule $(\mathtt{Unit}_{()})$ gives $\Delta; \Sigma; \Gamma \vdash () : \mathtt{Unit}$. From $\Delta; \Sigma; \Gamma \vdash l : \mathtt{Ref}\, \sigma$,

and Lemma 3(5), $l : \sigma' \in \Sigma$ for some σ', and $\texttt{Ref}\,\sigma \equiv \texttt{Ref}\,\sigma'$, which implies $\sigma \equiv \sigma'$. From $\Sigma \vdash \mu$ in order to show $\Sigma \vdash \mu'$ we have only to prove that $\Delta;\Sigma;\Gamma \vdash V : \sigma'$, which is immediate since $\Delta;\Sigma;\Gamma \vdash V : \sigma$ and $\sigma \equiv \sigma'$. $\square$

Remark 1. Note that the proof of subject reduction for the case of rule (β_v) extends without modifications to rule (β). Moreover, it is easy to check that the proof works for arbitrary contexts. So we can conclude that subject reduction for our type assignment system holds independently from the used reduction strategy.

In order to prove the progress of our type system we need a Canonical Form Lemma which can be proved in a standard way, see [10], by analyzing the typing rules and the syntax of values.

Lemma 6 (Canonical Forms).
1. $\Delta;\Sigma;\emptyset \vdash V : \texttt{Pos}$ *implies* $V \in \{1,2,\ldots\}$.
2. $\Delta;\Sigma;\emptyset \vdash V : \texttt{Nat}$ *implies* $V \in \{0,1,2,\ldots\}$.
3. $\Delta;\Sigma;\emptyset \vdash V : \texttt{Unit}$ *implies* $V = ()$.
4. $\Delta;\Sigma;\emptyset \vdash V : \tau \to \sigma$ *implies* $V = \lambda x.M$.
5. $\Delta;\Sigma;\emptyset \vdash V : \texttt{Ref}\,\tau$ *implies* $V = l$ *and* $l : \sigma \in \Sigma$ *for some* σ.

Theorem 2 (Progress). *Let $M \in \Lambda^0_{imp}$. Then $\Delta;\Sigma;\emptyset \vdash M : \sigma$ implies that either M is a value or for all μ such that $\Sigma \vdash \mu$ we have that $M \sharp \mu \longrightarrow N \sharp \mu'$ for some N, μ'.*

Proof. The proof is by induction on the derivation $\Sigma;\Gamma \vdash M : \tau$.
If the last applied rule is $(\to I)$, $(\texttt{Unit}_{()})$, $(\texttt{loc})$, $(\texttt{Nat})$, or $(\texttt{Pos})$, then M is a value.
If the last applied rule is $(\texttt{fix})$, then M is immediately reducible.
If the last applied rule is $(\to E)$, then M is NP, and $\Delta;\Sigma;\emptyset \vdash N : \tau \to \rho$, and $\Delta;\Sigma;\emptyset \vdash P : \tau$.
If N is a value, then by the Canonical Form Lemma 6(4), $N = \lambda x.Q$. If also P is a value, rule (β_v) applies. Otherwise, by induction hypothesis on $\Delta;\Sigma;\emptyset \vdash P : \tau$, for all μ such that $\Sigma \vdash \mu$ we have that $P \sharp \mu \longrightarrow P' \sharp \mu'$ for some P' and μ'. Therefore, for some $\mathscr{E}$, R and R', we get $P = \mathscr{E}[R]$ and $P' = \mathscr{E}[R']$. Consider the evaluation context $\mathscr{E}' = (\lambda x.Q)\mathscr{E}$. We have that $\mathscr{E}'[R] = M$ and $\mathscr{E}'[R] \sharp \mu \longrightarrow \mathscr{E}'[R'] \sharp \mu'$.
If N is not a value, by induction hypothesis on $\Delta;\Sigma;\emptyset \vdash N : \tau \to \rho$, for all μ such that $\Sigma \vdash \mu$ we have that $N \sharp \mu \longrightarrow N' \sharp \mu'$ for some N' and μ'. Therefore, for some $\mathscr{E}$, R and R', we get $N = \mathscr{E}[R]$ and $N' = \mathscr{E}[R']$. Consider the evaluation context $\mathscr{E}' = \mathscr{E}\,P$. We have that $\mathscr{E}'[R] = M$ and $\mathscr{E}'[R] \sharp \mu \longrightarrow \mathscr{E}'[R'] \sharp \mu'$.
If the last applied rule is $(\texttt{Unit})$ then M is $N := P$, and $\Delta;\Sigma;\emptyset \vdash N : \texttt{Ref}\,\tau$ and $\Delta;\Sigma;\emptyset \vdash P : \tau$.
If N is a value, from the Canonical Form Lemma 6(5), $N = l$, and $l : \sigma \in \Sigma$, for some σ. Moreover, from $\Sigma \vdash \mu$, we have that $(l = V) \in \mu$ for some V. If also P is a value, then rule $(unitR)$ is applicable. Otherwise, if P is not a value, we apply the induction hypothesis to $\Delta;\Sigma;\emptyset \vdash P : \tau$, and derive that for all μ such that $\Sigma \vdash \mu$ we have that $P \sharp \mu \longrightarrow P' \sharp \mu'$ for some P' and μ'. Therefore, for some $\mathscr{E}$, R and R', we get $P = \mathscr{E}[R]$ and $P' = \mathscr{E}[R']$. Consider the evaluation context $\mathscr{E}' = l := \mathscr{E}$. We have that $\mathscr{E}'[R] = M$ and $\mathscr{E}'[R] \sharp \mu \longrightarrow \mathscr{E}'[R'] \sharp \mu'$.
If N is not a value, by induction hypothesis on $\Delta;\Sigma;\emptyset \vdash N : \texttt{Ref}\,\tau$, for all μ such that $\Sigma \vdash \mu$ we have that $N \sharp \mu \longrightarrow N' \sharp \mu'$ for some N' and μ'. Therefore, for some $\mathscr{E}$, R

and R', we get $N = \mathscr{E}[R]$ and $N' = \mathscr{E}[R']$. Consider the evaluation context $\mathscr{E}' = \mathscr{E} := P$. We have that $\mathscr{E}'[R] = M$ and $\mathscr{E}'[R] \sharp \mu \longrightarrow \mathscr{E}'[R'] \sharp \mu'$.
The proof for the cases $(\mathtt{Ref}\, E)$, $(\mathtt{Ref}\, I)$, $(\mathtt{if})$, and $(+)$ are similar.
For rules $(\wedge I)$, $(\forall I)$ and $(\leq)$ the result follows directly by induction. $\square$

Let us restrict the language to the pure λ-calculus. Then our type assignment system preserves the typability power of intersection types, i.e., it gives types to all and only the strongly normalizing terms. As far as the expressive power is concerned, we can compare our system with System F [5], in its type assignment version [9], with the intersection type assignment system of [11], and with the system defined in [8], where both intersection and universally quantified types are present. Let $\vdash_F$ denote derivability in the type assignment version of system F [9] and $\vdash_P$ in the intersection type assignment system of [11]. The system in [8] can give types to all terms, since it contains the universal type ω, and a rule that assign ω to all terms. Let us consider a restriction of this system, obtained from it by erasing both the type ω and the related rule, for whose derivability we use $\vdash_{MZ}$. In order to prove that our system preserves the expressive power of $\vdash_F$, $\vdash_P$ and $\vdash_{MZ}$, we define a decorating function *dec*, transforming every type in [8] non containing occurrences of ω in a type of our system, in the following way:

$$dec(\tau \;\mathtt{op}\; \sigma) = dec(\tau)\; \mathtt{op}\; dec(\sigma) \quad (\mathtt{op} \in \{\rightarrow, \wedge\}) \qquad dec(\forall t.\tau) = \forall t : \mathbf{S}.\tau$$

The function *dec* can be obviously applied also to the set of System F types and to the set of intersection types, which are proper subsets of the types in [8].

Theorem 3. *Let M be a term of the pure λ-calculus, σ a type, Γ a type environment and Δ the kind environment which gives kind* **S** *to all the type variables occurring in σ and Γ.*

1. *If $\Gamma \vdash_{MZ} M : \sigma$, then $\Delta;\emptyset;\Gamma \vdash M : dec(\sigma)$.*
2. *If $\Gamma \vdash_F M : \sigma$, then $\Delta;\emptyset;\Gamma \vdash M : dec(\sigma)$.*
3. *If $\Gamma \vdash_P M : \sigma$, then $\Delta;\emptyset;\Gamma \vdash M : dec(\sigma)$.*
4. *M is typable in the system of Fig. 5 if and only if it is strongly normalizing.*

Proof. Point 1 is immediate, since the rules of $\vdash_{MZ}$ are a proper subset of our rules, and also the $\leq$ relation on types is the same, when reference types are not present.

Points 2 and 3 follow from Point 1, since the rules of $\vdash_F$ and of $\vdash_P$ are a proper subsets of the rules of $\vdash_{MZ}$.

For Point 4 since all strongly normalising terms are typable in the system of [11] we get from Point 3 that all strongly normalising terms are typable in our system. The vice versa can be proved by a standard use of the computability technique as done in [11]. $\square$

4 Conclusion

In this paper we discuss how to combine intersection, universally quantified and reference types in a meaningful way. The naive use of intersection and universally quantified types is unsound in presence of references, as shown in [3] and in the introduction of this paper. Davies and Pfenning solve the problem by restricting both the definition of the preorder relation $\leq$ between types, and the type assignment system. In the preorder relation $\leq$ between types they do not have the standard rules:

$$(\to \wedge) \qquad (\tau \to \sigma) \wedge (\tau \to \rho) \leq \tau \to \sigma \wedge \rho$$
$$(\to \forall) \qquad \forall t.\tau \to \sigma \leq \tau \to \forall t.\sigma \qquad t \notin \mathscr{T}V(\tau)$$

The type assignment system is restricted in such a way that the intersection and the universal quantification can be introduced just in case the subject is a value. Then the subject reduction property holds, for a call-by-value reduction semantics of terms. As already noticed in [4], while in this way they solve the problem described in the introduction, in the system there are unsound typings. In fact the term $x := x+1$ can be typed in their system, extended with the standard typing rule for the sum, through the following derivation:

$$\dfrac{\dfrac{\Delta;\emptyset;x:\mathtt{Nat}\wedge\mathtt{Ref}\,\mathtt{Nat}\vdash x:\mathtt{Nat}\wedge\mathtt{Ref}\,\mathtt{Nat}}{\Delta;\emptyset;x:\mathtt{Nat}\wedge\mathtt{Ref}\,\mathtt{Nat}\vdash x:\mathtt{Ref}\,\mathtt{Nat}}(\leq) \qquad \dfrac{\dfrac{\Delta;\emptyset;x:\mathtt{Nat}\wedge\mathtt{Ref}\,\mathtt{Nat}\vdash x:\mathtt{Nat}\wedge\mathtt{Ref}\,\mathtt{Nat}}{\Delta;\emptyset;x:\mathtt{Nat}\wedge\mathtt{Ref}\,\mathtt{Nat}\vdash x:\mathtt{Nat}}(\leq)}{\Delta;\emptyset;x:\mathtt{Nat}\wedge\mathtt{Ref}\,\mathtt{Nat}\vdash x+1:\mathtt{Nat}}(+)}{\Delta;\emptyset;x:\mathtt{Nat}\wedge\mathtt{Ref}\,\mathtt{Nat}\vdash x:=x+1:\mathtt{Unit}}(\mathtt{Unit})$$

In [4] a different solution is proposed, for a system with reference and intersection types only, which does restrict neither the definition of the $\leq$ relation between types nor the type assignment system rules. In this system there cannot be intersections between reference and non-reference types, so, for example, $\mathtt{Nat}\wedge\mathtt{Ref}\,\mathtt{Nat}$ is not a type. Syntactically, this is realized through a partial intersection operator, $\cap$, which applied to two non-reference types returns their intersection, and applied to two reference types commutes with the `Ref` constructor pushing the operator inside the `Ref`. The system is shown to be sound and no expressive power is lost in comparison with the original system [2] [11] of intersection types when we restrict to the terms of pure λ-calculus.

In this paper, we consider a system with intersection, universally quantified and reference types. Our aim is to design a sound system having minimal restrictions. The solution adopted to avoid unsoundness is different from both [3] and [4], and leads to a more elegant system. The only restriction we impose on types is that the `Ref` constructor can be applied only to closed types. So the quantification on reference types becomes meaningless, since $\forall t.\mathtt{Ref}\,\tau$ is equivalent to $\mathtt{Ref}\,\tau$. Regarding intersection types we do not have restrictions. In particular, we may have intersection between reference and non-reference types. With a notion of kind and a kind assignment we keep track of potential reference types. The soundness is reached by limiting the definition of $\leq$ relation between types in the rule for intersection elimination, which may only be applied if the intersection does not contain reference types. For our types rules $(\to\wedge)$ and $(\to\forall)$ hold in both directions. As a results, our system enjoys subject reduction independently from the reduction strategy. In fact the critical term $(\lambda x.(\lambda y.!x)(x:=0))\mathtt{ref}\,1$, showed in the introduction, in our system has only types equivalent to `Nat`, which is the type of both 0 and 1, so the typing is preserved under any reduction strategy. Moreover unsound terms as the one shown before cannot be typed (but their sound versions $x :=!x+1$ and $\mathtt{ref}\,x := x+1$ are typable).

When restricted to the pure functional part of the language, our typing system has a stronger typability power than the system of [3]. As an example, consider the strongly normalizing pure λ-term $(\lambda xy.(\lambda z.zz)(xy))(\lambda t.t)$ which is typable in our system (see Theorem 3(4)), while it is not typable in the system of [3]. For typing this term it is necessary to introduce an intersection between two subderivations whose subject is

xy, and in the system of [3] this is not possible, since this subterm is not a value. More precisely, if $\sigma_1 = (\mathtt{Nat} \to \mathtt{Nat}) \wedge ((\mathtt{Nat} \to \mathtt{Nat}) \to \mathtt{Nat} \to \mathtt{Nat})$ and $\sigma_2 = \mathtt{Nat} \wedge (\mathtt{Nat} \to \mathtt{Nat})$, it is easy to verify that $\lambda t.t$ has type σ_1 and $\lambda z.zz$ has type $\sigma_2 \to \mathtt{Nat}$. Therefore, in order to type the above term we need to derive $\Delta; \emptyset; \{x : \sigma_1, y : \sigma_2\} \vdash xy : \sigma_2$, which requires the application of rule $(\wedge I)$ to *xy*.

The system we define is clearly undecidable, as the subtyping itself is undecidable also when restricted to the types of System F, as proved in [13]. Moreover, type inference for the systems of [5] and [11] is undecidable, as proved in [14] and [11], respectively. Therefore, the present system cannot be proposed for real programming. The interest of this paper is merely foundational, since it explores the difficulties in putting together different type constructs, and formalizes a sound proposal which enhances previous solutions. A future work will be to tailor a proper subsystem of our typing system, possibly using bidirectional type checking as proposed in [3], with the property of being decidable while preserving a good expressive power.

Acknowledgements We gratefully acknowledge fruitful discussions with Frank Pfenning and Betti Venneri. We also thank the referees for their comments.

References

1. M. Coppo. An Extended Polymorphic Type System for Applicative Languages. In P. Dembinski, editor, *MFCS'80*, volume 88 of *LNCS*, pages 194–204. Springer-Verlag, 1980.
2. M. Coppo and M. Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.
3. R. Davies and F. Pfenning. Intersection Types and Computational Effects. In P. Wadler, editor, *ICFP'00*, volume 35(9) of *SIGPLAN Notices*, pages 198–208. ACM Press, 2000.
4. M. Dezani-Ciancaglini and S. Ronchi Della Rocca. Intersection and Reference Types. In E. Barendsen, V. Capretta, H. Geuvers, and M. Niqui, editors, *Reflections on Type Theory, Lambda Calculus, and the Mind*, pages 77–86. Radboud University Nijmegen, 2007.
5. J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse d'Etat, Université de Paris VII, 1972.
6. D. Leivant. Polymorphic Type Inference. In A. Demers and T. Teitelbaum, editors, *POPL'83*, pages 88–98. ACM Press, 1983.
7. J. M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. Ph. d. thesis, Massachusets Institute of Technology, 1987.
8. I. Margaria and M. Zacchi. Principal Typing in a $\forall\wedge$-Discipline. *Journal of Logic and Computation*, 5(3):367–381, 1995.
9. J. Mitchell. Polymorphic Type Inference and Containment. *Information and Computation*, 76(2/3):211–249, 1988.
10. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
11. G. Pottinger. A Type Assignment for the Strongly Normalizable λ-terms. In R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, London, 1980.
12. J. C. Reynolds. Towards a Theory of Type Structure. In J. Loecks, editor, *Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.
13. J. Tiuryn and P. Urzyczyn. The Subtyping Problem for Second-Order Types Is Undecidable. *Information and Computation*, 179(1):1–18, 2002.
14. J. B. Wells. Typability and Type Checking in System F are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999.
15. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115:38–94, 1994.