INTERSECTION AND REFERENCE TYPES

DEDICATED TO HENK BARENDREGT ON THE OCCASION OF HIS 60TH BIRTHDAY

MARIANGIOLA DEZANI-CIANCAGLINI AND SIMONA RONCHI DELLA ROCCA

Dipartimento di Informatica, Università di Torino, corso Svizzera 185, Torino, Italy
e-mail address: dezani, ronchi@di.unito.it

Abstract. Aim of this paper is to understand the interplay between intersection and reference types. Putting together the standard typing rules for intersection types and reference types leads to loss of subject reduction. The problem comes from the invariance of the reference type constructor and the rule of intersection elimination, which is essentially a subsumption rule. We propose a solution which only allows intersection of non-reference types, and in which the rule of intersection introduction uses an operator on types pushing intersections under references in an iterative way. The so obtained type assignment system is shown to be sound and, when restricted to pure λ-calculus, as expressive as the standard type assignment system of intersection types.

Introduction

This paper deals with the problem of understanding the meaning of types built using both intersection and reference type constructors. Reference types are an essential tool for typing memory locations and the operations of reading and writing in memory. Intersection types allow for discrete polymorphism, so increasing both the typability and the type expressivity, and in particular giving a formal account to overloading. Putting together these two features is useful for typing in a significant way a programming language with imperative features. It is well-know 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 [Pie02] [page 198]. On the other hand, the intersection elimination typing rule is essentially a subsumption rule, being the intersection of two types contained in both types.

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

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

2000 ACM Subject Classification: F.3.3 [LOGICS AND MEANINGS OF PROGRAMS]: Studies of Program Constructs -*Type structure*; D.3.3 [PROGRAMMING LANGUAGES]: Language Constructs and Features - *Polymorphism*; F.4.1 [MATHEMATICAL LOGIC AND FORMAL LANGUAGES]: Mathematical Logic -*Lambda calculus and related systems* .

Key words and phrases: λ-calculus, intersection types, reference types, subtyping, store typings.

LOGICAL METHODS IN COMPUTER SCIENCE DOI:10.2168/LMCS-???

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

- the type $\texttt{Ref}\,\texttt{nat}$ for x in the typing of $x := 0$ getting the type `unit`;
- the type $\texttt{Ref}\,\texttt{pos}$ for x in the typing of $!x$ getting the type `pos`.

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

$(\lambda x.(\lambda y.!x)(x := 0))\texttt{ref}\,1 \sharp \emptyset \longrightarrow_{\mathrm{v}} (\lambda x.(\lambda y.!x)(x := 0))l \sharp (l = 1) \longrightarrow_{\mathrm{v}}$
$(\lambda y.!l)(l := 0) \sharp (l = 1) \longrightarrow_{\mathrm{v}} (\lambda y.!l)(\,) \sharp (l = 0) \longrightarrow_{\mathrm{v}} !l \sharp (l = 0) \longrightarrow_{\mathrm{v}} 0 \sharp (l = 0)$

and 0 does not have the type `pos`.

This example is a transcription of an example in [DP00], where the authors give a solution we will discuss in the Conclusion of the present paper comparing it with our proposal.

As suggested by the above example, a memory location typed by $\texttt{Ref}\,\texttt{pos} \wedge \texttt{Ref}\,\texttt{nat}$ must contain values which are both `pos` and `nat`, i.e. values of type $\texttt{pos} \wedge \texttt{nat}$. This can be better expressed by typing the memory location with the type $\texttt{Ref}\,(\texttt{pos} \wedge \texttt{nat})$. This clearly generalises to the case of two arbitrary types σ, σ' which are non-reference types, i.e. we claim it is sensible to type memory locations with the type $\texttt{Ref}\,(\sigma \wedge \sigma')$, but not with the type $\texttt{Ref}\,\sigma \wedge \texttt{Ref}\,\sigma'$.

Another observation is that it is meaningless the intersection between a reference type and a non-reference type, since they express incompatible properties. This remark is also substantiated in the Conclusion by means of an example.

By the above we think it is sensible to allow the intersection only between non-reference types, and to introduce an operator which applied to two reference types iterates the pushing of intersections under references. For example this operator applied to $\texttt{Ref}\,\texttt{pos}$ and $\texttt{Ref}\,\texttt{nat}$ returns $\texttt{Ref}\,(\texttt{pos} \wedge \texttt{nat})$.

Building on this idea we propose a type system for a λ-calculus with assignment statements and reference/dereference constructors. We show soundness, i.e. subject reduction and progress, of our type system. Lastly we observe that no expressive power is lost in comparison with the original system [CDC80] of intersection types when we restrict to the terms of pure λ-calculus.

1. Syntax and Reduction Rules

The language Λ_{imp} we are working with is a simplification of the language in [DP00], which in its turn belongs to the ML-family, the difference being in the lack of the *let* construction and of the binary strings. It is well know that the the *let* constructor is syntactic sugar [Pie02] [Section 11.5] and in presence of intersection types it does not increase the typability of the language, since intersection types allow to type the translation of *let* in pure λ-calculus [Cop80]. The only data types of Λ_{imp} are the numerals, but this is enough for discussing the typing problems cited in the introduction.
Terms of Λ_{imp} are defined by the following grammar:

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

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 writing and reading respectively, as it will be clear from the operational semantics given below. The set of values is the subset of Λ^0_{imp} defined as follows:

$$V \quad ::= \quad n \mid \lambda x.M \mid l \mid \texttt{op} \mid () \qquad (\lambda x.M \in \Lambda^0_{imp})$$

Values are results of the evaluation, and so they can be stored. In fact, the store can be modelled as a finite association between locations and values:

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

On Λ_{imp} we consider two reduction semantics, which differ by the parameter passing politics, through the notion of *evaluation context*. An evaluation context can be defined starting from the grammar for Λ_{imp}, where a new constant has been added, the *hole* ($[\,]$). The call-by-name evaluation contexts are defined as follows:

$$\begin{array}{rcl} E_{\mathrm{n}} & ::= & [\,] \mid E_{\mathrm{n}} M \mid \texttt{ref}\, E_{\mathrm{n}} \mid !E_{\mathrm{n}} \mid E_{\mathrm{n}} := M \mid \\ & & V := E_{\mathrm{n}} \mid \texttt{if}\, E_{\mathrm{n}}\, \texttt{then}\, M\, \texttt{else}\, M \mid \\ & & n E_{\mathrm{n}} M \mid n\, \texttt{op}\, E_{\mathrm{n}} \end{array}$$

The call-by-value evaluation contexts are obtained from the call-by-name ones by replacing E_{n} by E_{v} and by adding:

$$E_{\mathrm{v}} \quad ::= \quad V E_{\mathrm{v}}$$

The reduction semantics are given by two sets of rules, of the shape $E[M] \sharp \mu \longrightarrow E[N] \sharp \mu'$, where E is either E_{n} or E_{v}, M is a closed term, $[N/x]$ is the capture free substitution, and μ is a store. The call-by-name reduction rules are the following:

$$\begin{array}{lcll} E_{\mathrm{n}}[(\lambda x.M)N] \sharp \mu & \longrightarrow_{\mathrm{n}} & E_{\mathrm{n}}[M[N/x]] \sharp \mu & (\beta) \\ E_{\mathrm{n}}[\texttt{fix}\, x.M \sharp \mu] & \longrightarrow_{\mathrm{n}} & E_{\mathrm{n}}[M[\texttt{fix}\, x.M]/x] \sharp \mu & \\ E_{\mathrm{n}}[\texttt{ref}\, V] \sharp \mu & \longrightarrow_{\mathrm{n}} & E_{\mathrm{n}}[l] \sharp \mu, (l = V) & l \text{ fresh} \\ E_{\mathrm{n}}[!l] \sharp \mu, (l = V) & \longrightarrow_{\mathrm{n}} & E_{\mathrm{n}}[V] \sharp \mu, (l = V) & \\ E_{\mathrm{n}}[l := V] \sharp \mu, (l = V') & \longrightarrow_{\mathrm{n}} & E_{\mathrm{n}}[()] \sharp \mu, (l = V) & \\ E_{\mathrm{n}}[\texttt{if}\, 0\, \texttt{then}\, M\, \texttt{else}\, N] \sharp \mu & \longrightarrow_{\mathrm{n}} & E_{\mathrm{n}}[M] \sharp \mu & \\ E_{\mathrm{n}}[\texttt{if}\, n\, \texttt{then}\, M\, \texttt{else}\, N] \sharp \mu & \longrightarrow_{\mathrm{n}} & E_{\mathrm{n}}[N] \sharp \mu & n \neq 0 \\ E_{\mathrm{n}}[0 + 0] \sharp \mu & \longrightarrow_{\mathrm{n}} & E_{\mathrm{n}}[0] \sharp \mu & \\ E_{\mathrm{n}}[0 + 1] \sharp \mu & \longrightarrow_{\mathrm{n}} & E_{\mathrm{n}}[1] \sharp \mu & \\ \ldots & & & \end{array}$$

The call-by-value reduction rules are obtained simply by replacing, in the previous rules, E_{n} by E_{v} and the rule (β) by:

$$E_{\mathrm{v}}[(\lambda x.M)V] \sharp \mu \quad \longrightarrow_{\mathrm{v}} \quad E_{\mathrm{v}}[M[V/x]] \sharp \mu \qquad (\beta_v)$$

By $\Rightarrow_{\mathrm{n}}$ ($\Rightarrow_{\mathrm{v}}$) we will denote the transitive and reflexive closure of $\longrightarrow_{\mathrm{n}}$ ($\longrightarrow_{\mathrm{v}}$). We will use $\longrightarrow$ ($\Rightarrow$) for either $\longrightarrow_{\mathrm{n}}$ or $\longrightarrow_{\mathrm{v}}$ ($\Rightarrow_{\mathrm{n}}$ or $\Rightarrow_{\mathrm{v}}$) , when the difference is clear from the context or it is unimportant.

$$\frac{}{\texttt{pos} \leq \texttt{nat}}\,(pos) \qquad \frac{}{\tau \leq \tau}\,(id) \qquad \frac{\tau \leq \tau'' \quad \tau'' \leq \tau'}{\tau \leq \tau'}\,(trans)$$

$$\frac{}{\sigma \wedge \sigma' \leq \sigma}\,(\wedge_L) \qquad \frac{}{\sigma \wedge \sigma' \leq \sigma'}\,(\wedge_R) \qquad \frac{\tau \leq \tau_1 \quad \tau \leq \tau_2}{\tau \leq \tau_1 \cap \tau_2}\,(\cap)$$

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}\,(\to) \qquad \frac{\tau \leq \tau' \quad \tau' \leq \tau}{\texttt{Ref}\,\tau \leq \texttt{Ref}\,\tau'}\,(\texttt{Ref}) \qquad \frac{\tau_1 \cap \tau_2 \neq \bot}{(\tau \to \tau_1) \wedge (\tau \to \tau_2) \leq \tau \to \tau_1 \cap \tau_2}\,(\to \cap)$$

Figure 1: The preorder relation $\leq$ on types

2. Type System

Types are defined through the following grammar:

$$\begin{array}{lcll} \tau, \zeta & ::= & \sigma \mid \rho & \text{(types)} \\ \sigma & ::= & \texttt{pos} \mid \texttt{nat} \mid \texttt{unit} \mid \tau \to \tau \mid \sigma \wedge \sigma & \text{(non-reference types)} \\ \rho & ::= & \texttt{Ref}\,\tau & \text{(reference types)} \end{array}$$

We assume the following precedence relation between type constructs: $\texttt{Ref}\,, \wedge, \to$. As usual $\to$ associates to the right. `nat` and `pos` represent the sets of natural and positive numbers respectively, `unit` is the type of commands. Note that types are divided in two classes, reference types and non-reference types. and the constructor $\wedge$ is restricted to non-reference types. So $\texttt{Ref}\,(\texttt{pos} \wedge \texttt{nat})$ is a type, while $\texttt{Ref}\,\texttt{pos} \wedge \texttt{Ref}\,\texttt{nat}$ is not a type. $\texttt{Ref}\,(\texttt{pos} \wedge \texttt{nat})$ is the type of a location that can store values which are in the same time both natural and positive numbers. In order to deal in a uniform manner with the type syntax, we define a binary operator $\cap$ working on types, such that, when applied to two non-reference types it returns their intersection, when applied to two reference types it commutes with the `Ref` constructor, and it is undefined otherwise.

The operator $\cap$ is defined as follows:

$$\tau \cap \tau' = \begin{cases} \sigma \wedge \sigma' & \text{if } \tau = \sigma \text{ and } \tau' = \sigma', \\ \texttt{Ref}\,(\tau_1 \cap \tau_1') & \text{if } \tau = \texttt{Ref}\,\tau_1 \text{ and } \tau' = \texttt{Ref}\,\tau_1' \text{ and } \tau_1 \cap \tau_1' \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

$\cap$ binds weaker than $\wedge$ and stronger than $\to$.
The following property can be easily be proved by induction on the definition of $\cap$.

Property 2.1. *$\tau \cap \tau'$ defined implies $\tau = \underbrace{\texttt{Ref}\,(\texttt{Ref}\,\ldots(\texttt{Ref}}_{k}\,\sigma)\ldots)$, $\tau' = \underbrace{\texttt{Ref}\,(\texttt{Ref}\,\ldots(\texttt{Ref}}_{k}\,\sigma')\ldots)$, and $\tau \cap \tau' = \underbrace{\texttt{Ref}\,(\texttt{Ref}\,\ldots(\texttt{Ref}}_{k}(\sigma \wedge \sigma'))\ldots)$, for some σ, σ' and $k \geq 0$.*

A preorder relation $\leq$ is defined on types through the rules shown in Fig. 2.

Some comments are in order. Rule (*pos*) allows for overloading. Reference types and non-reference types are incomparable. Notice that Lemma 2.2(1) assures that $\tau_1 \cap \tau_2$ is always defined in rule ($\cap$). Rule (`Ref`) just allows $\wedge$ be commutative and associative also under the constructor `Ref`, i.e., $\texttt{Ref}\,(\tau_1 \wedge \tau_2)$ and $\texttt{Ref}\,(\tau_2 \wedge \tau_1)$ be equivalent. All the other

rules are the standard rules for intersection types, but rule $(\to \cap)$ which takes into account the fact that $\wedge$ is partially defined. From now on, we will consider types modulo commutativity and associativity of $\wedge$ and $\cap$.

Lemma 2.2. (1) *If $\tau \leq \tau_1$ and $\tau \leq \tau_2$, then $\tau_1 \cap \tau_2$ is defined.*
(2) *If $\tau_1 \leq \tau$ and $\tau_2 \leq \tau$, then $\tau_1 \cap \tau_2$ is defined.*
(3) *If $\bigwedge_{i\in I}(\tau_i \to \zeta_i) \leq \bigwedge_{j\in J}(\tau'_j \to \zeta'_j)$, then for all $j \in J$ there is $H_j \subseteq I$ such that $\tau_i \geq \tau'_j$ for all $i \in H_j$ and $\bigcap_{i\in H_j} \zeta_i \leq \zeta'_j$.*

Proof. Both points (1) and (2) are consequences of the following facts, that can be easily proved by induction on $\leq$ taking into account Property 2.1:

- $\tau \leq \tau'$ implies $\tau \cap \tau'$ is defined;
- $\tau \cap \tau'$ and $\tau' \cap \tau''$ defined imply $\tau \cap \tau''$ is defined.

(3) By induction on $(\leq)$. The only not immediate case is when the last applied rule is $(trans)$. Let $\bigwedge_{i\in I}(\tau_i \to \zeta_i) \leq \zeta$ and $\zeta \leq \bigwedge_{j\in J}(\tau'_j \to \zeta'_j)$. By inspecting the rules of $\leq$, it is immediate to see that $\to$-types are unrelated both with reference types and with constant types, so the general shape of ζ is $\bigwedge_{h\in H}(\tau''_h \to \zeta''_h)$. On one side, by induction, for all $h \in H$ there is $K_h \subseteq I$ such that $\tau_i \geq \tau''_h$ for all $i \in K_h$ and $\bigcap_{i\in K_h} \zeta_i \leq \zeta''_h$. On the other side, always by induction, for all $j \in J$ there is $R_j \subseteq H$ such that $\tau''_h \geq \tau'_j$ for all $h \in R_j$ and $\bigcap_{h\in R_j} \zeta''_h \leq \zeta'_j$. Let $H_j = \bigcup_{h\in R_j} K_h$: clearly H_j is a not empty subset of I. Then, by transitivity, for all $j \in J$, $\tau_i \geq \tau'_j$ for all $i \in H_j$ and $\bigcap_{i\in H_j} \zeta_i \leq \zeta'_j$. □

The typing system proves judgments of the shape:

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

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

$$\begin{array}{lcll} \Sigma & ::= & \emptyset \mid \Sigma, l : \tau & l \notin dom(\Sigma) \\ \Gamma & ::= & \emptyset \mid \Gamma, x : \tau & x \notin dom(\Gamma) \end{array}$$

where *dom* is the environment domain.

The typing rules are given in Fig. 2. Notice that the condition $\tau_1 \cap \tau_2 \neq \bot$ in rule $(\cap I)$ is necessary, since for example we can derive both

$$\emptyset; \{x : (\mathtt{nat} \to \mathtt{nat}) \wedge (\mathtt{nat} \to \mathtt{Ref\,nat}), y : \mathtt{nat}\} \vdash xy : \mathtt{nat}$$

and

$$\emptyset; \{x : (\mathtt{nat} \to \mathtt{nat}) \wedge (\mathtt{nat} \to \mathtt{Ref\,nat}), y : \mathtt{nat}\} \vdash xy : \mathtt{Ref\,nat},$$

but $\mathtt{nat} \cap \mathtt{Ref\,nat}$ is undefined.

It is easy to verify that the following rules are admissible in our typing system:

$$\frac{\Sigma; \Gamma \vdash M : \sigma \wedge \sigma'}{\Sigma; \Gamma \vdash M : \sigma}\,(\wedge E) \qquad \frac{\Sigma; \Gamma, x : \sigma \vdash M : \tau}{\Sigma; \Gamma, x : \sigma \wedge \sigma' \vdash M : \tau}\,(\wedge IL)$$

but the rules obtained from these by replacing non-reference types with reference types and $\wedge$ by $\cap$ are unsound. Also strengthening and weakening for both environments are admissible rules:

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

$$\frac{\Sigma;\Gamma,x:\tau\vdash M:\tau}{\Sigma;\Gamma\vdash \mathtt{fix}\,x.M:\tau}\,(\mathtt{fix}) \qquad \frac{}{\Sigma,l:\tau;\Gamma\vdash l:\mathtt{Ref}\,\tau}\,(cell) \qquad \frac{\Sigma;\Gamma\vdash M:\tau}{\Sigma;\Gamma\vdash \mathtt{ref}\,M:\mathtt{Ref}\,\tau}\,(\mathtt{Ref}\,I)$$

$$\frac{\Sigma;\Gamma\vdash M:\mathtt{Ref}\,\tau}{\Sigma;\Gamma\vdash !M:\tau}\,(\mathtt{Ref}\,E) \qquad \frac{}{\Sigma;\Gamma\vdash ():\mathtt{unit}}\,(\mathtt{unit}_{()}) \qquad \frac{\Sigma;\Gamma\vdash M:\mathtt{Ref}\,\tau \quad \Sigma;\Gamma\vdash N:\tau}{\Sigma;\Gamma\vdash M:=N:\mathtt{unit}}\,(\mathtt{unit})$$

$$\frac{\Sigma;\Gamma\vdash M:\mathtt{nat} \quad \Sigma;\Gamma\vdash N_1:\tau \quad \Sigma;\Gamma\vdash N_2:\tau}{\Sigma;\Gamma\vdash \mathtt{if}\,M\,\mathtt{then}\,N_1\,\mathtt{else}\,N_2:\tau}\,(\mathtt{if\ then\ else})$$

$$\frac{}{\Sigma;\Gamma\vdash 0:\mathtt{nat}}\,(\mathtt{nat}) \qquad \frac{}{\Sigma;\Gamma\vdash 1:\mathtt{pos}}\,(\mathtt{pos}) \qquad \frac{\Sigma;\Gamma\vdash M:\mathtt{nat} \quad \Sigma;\Gamma\vdash N:\mathtt{pos}}{\Sigma;\Gamma\vdash M+N:\mathtt{pos}}\,(+)$$

$$\ldots$$

$$\frac{\Sigma;\Gamma\vdash M:\tau \quad \Sigma;\Gamma\vdash M:\tau' \quad \tau\cap\tau'\neq\bot}{\Sigma;\Gamma\vdash M:\tau\cap\tau'}\,(\cap I) \qquad \frac{\Sigma;\Gamma\vdash M:\tau \quad \tau\leq\tau'}{\Sigma;\Gamma\vdash M:\tau'}\,(\leq)$$

Figure 2: The Typing Rules for Terms

$$\frac{\Sigma,l:\tau';\Gamma\vdash M:\tau \quad l\notin\mathcal{L}(M)}{\Sigma;\Gamma\vdash M:\tau}\,(streng\Sigma) \qquad \frac{\Sigma;\Gamma\vdash M:\tau \quad l\notin dom(\Sigma)}{\Sigma,l:\tau';\Gamma\vdash M:\tau}\,(weak\Sigma)$$

$$\frac{\Sigma;\Gamma,x:\tau'\vdash M:\tau \quad x\notin\mathcal{FV}(M)}{\Sigma;\Gamma\vdash M:\tau}\,(streng\Gamma) \qquad \frac{\Sigma;\Gamma\vdash M:\tau \quad x\notin dom(\Gamma)}{\Sigma;\Gamma,x:\tau'\vdash M:\tau}\,(weak\Gamma)$$

where $\mathcal{L}(M),\mathcal{FV}(M)$ are the sets of locations and free variables which occur in M.

As usual our type system enjoys a Generation Lemma, which allows to invert the typing rules. We omit the points concerning numerals and operators on numerals which are obvious.

Lemma 2.3 (Generation)**.** (1) $\Sigma;\Gamma\vdash x:\tau$ *implies* $x:\tau'\in\Gamma$ *for some* $\tau'\leq\tau$;

(2) $\Sigma;\Gamma\vdash\lambda x.M:\tau$ *implies* $\tau\geq\bigwedge_{i\in I}(\tau_i\to\tau_i')$ *and* $\Sigma;\Gamma,x:\tau_i\vdash M:\tau_i'$ *for some* I, τ_i *and* τ_i', *where* $i\in I$;

(3) $\Sigma;\Gamma\vdash MN:\tau$ *implies* $\tau\geq\bigcap_{i\in I}\tau_i'$, *and* $\Sigma;\Gamma\vdash M:\tau_i\to\tau_i'$ *and* $\Sigma;\Gamma\vdash N:\tau_i$, *for some* I, τ_i *and* τ_i', *where* $i\in I$;

(4) $\Sigma;\Gamma\vdash\mathtt{fix}\,x.M:\tau$ *implies* $\tau\geq\bigcap_{i\in I}\tau_i$, *and* $\Sigma;\Gamma,x:\tau_i\vdash M:\tau_i$ *for some* I, τ_i, *where* $i\in I$;

(5) $\Sigma;\Gamma\vdash l:\tau$ *implies* $\tau=\mathtt{Ref}\,\tau'$ *and* $l:\tau'\in\Sigma$, *for some* τ';

(6) $\Sigma;\Gamma\vdash\mathtt{ref}\,M:\tau$ *implies* $\tau=\mathtt{Ref}\,\tau'$ *and* $\Sigma;\Gamma\vdash M:\tau'$, *for some* τ';

(7) $\Sigma;\Gamma\vdash !M:\tau$ *implies* $\Sigma;\Gamma\vdash M:\mathtt{Ref}\,\tau$;

(8) $\Sigma;\Gamma\vdash M:=N:\tau$ *implies* $\tau=\mathtt{unit}$ *and* $\Sigma;\Gamma\vdash M:\mathtt{Ref}\,\tau'$ *and* $\Sigma;\Gamma\vdash N:\tau'$, *for some* τ';

(9) $\Sigma;\Gamma \vdash ( ) : \tau$ *implies* $\tau = \mathtt{unit}$*;*
(10) $\Sigma;\Gamma \vdash \mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2 : \tau$ *implies* $\Sigma;\Gamma \vdash M : \mathtt{nat}$ *and* $\Sigma;\Gamma \vdash N_1 : \tau$ *and* $\Sigma;\Gamma \vdash N_2 : \tau$*.*

Proof. All points can be proved by induction on derivations, the only interesting cases being when the last applied rule is $(\cap I)$.

For (2) we get:

$$\frac{\Sigma;\Gamma \vdash \lambda x.M : \tau \quad \Sigma;\Gamma \vdash \lambda x.M : \zeta \quad \tau \cap \zeta \neq \bot}{\Sigma;\Gamma \vdash \lambda x.M : \tau \cap \zeta}\ (\cap I)$$

By induction we have $\tau \geq \bigwedge_{i\in I}(\tau_i \to \tau'_i)$ and $\Sigma;\Gamma, x : \tau_i \vdash M : \tau'_i$ for some I, τ_i and τ'_i, where $i \in I$, and $\zeta \geq \bigwedge_{j\in J}(\zeta_j \to \zeta'_j)$ and $\Sigma;\Gamma, x : \zeta_j \vdash M : \zeta'_j$ for some J, ζ_j and ζ'_j, where $j \in J$. We conclude since $\tau \cap \zeta \geq \bigwedge_{i\in I}(\tau_i \to \tau'_i) \wedge \bigwedge_{j\in J}(\zeta_j \to \zeta'_j)$.

The proof for the other points is similar. □

Notice that, without reference types, points (2), (3) and (4) of the previous lemma hold with I a singleton set. The partiality of the $\wedge$ constructor is reflected in the necessity of allowing set of types of cardinality bigger than 1. For example for the identity term $\lambda x.x$ we can derive the type $(\mathtt{nat} \to \mathtt{nat}) \wedge (\mathtt{Ref}\,\mathtt{nat} \to \mathtt{Ref}\,\mathtt{nat})$, but there are no types τ_1, τ_2 such that $(\mathtt{nat} \to \mathtt{nat}) \wedge (\mathtt{Ref}\,\mathtt{nat} \to \mathtt{Ref}\,\mathtt{nat}) \geq \tau_1 \to \tau_2$ and $\emptyset; \{x : \tau_1\} \vdash x : \tau_2$. Similarly from $\{x : (\mathtt{nat} \to \mathtt{nat}) \wedge (\mathtt{nat} \to \mathtt{Ref}\,\mathtt{nat}), y : \mathtt{nat}, z : (\mathtt{nat} \to \mathtt{nat}) \wedge (\mathtt{Ref}\,\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat})\}$ 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 $xy : \tau_1$. Lastly we can derive $\emptyset; \{y : (\mathtt{Ref}\,\mathtt{nat} \to \mathtt{Ref}\,\mathtt{nat}) \wedge (\mathtt{Ref}\,(\mathtt{nat} \to \mathtt{nat}) \to \mathtt{Ref}\,(\mathtt{nat} \to \mathtt{nat}))\} \vdash \mathtt{fix}\, x.yx : \mathtt{Ref}\,(\mathtt{nat} \wedge (\mathtt{nat} \to \mathtt{nat}))$, but we cannot derive $\emptyset; \{x : \mathtt{Ref}\,(\mathtt{nat} \wedge (\mathtt{nat} \to \mathtt{nat})), y : (\mathtt{Ref}\,\mathtt{nat} \to \mathtt{Ref}\,\mathtt{nat}) \wedge (\mathtt{Ref}\,(\mathtt{nat} \to \mathtt{nat}) \to \mathtt{Ref}\,(\mathtt{nat} \to \mathtt{nat}))\} \vdash yx : \mathtt{Ref}\,(\mathtt{nat} \wedge (\mathtt{nat} \to \mathtt{nat}))$.

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

Lemma 2.4 (Substitution)**.** $\Sigma;\Gamma, x : \tau \vdash M : \tau'$ *and* $\Sigma;\Gamma \vdash N : \tau$ *imply* $\Sigma;\Gamma \vdash M[N/x] : \tau'$*.*

In order to state Subject Reduction for our type system we need to define the *agreement* between a store environment and a store [Pie02] [Definition 13.5.1]. 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 $\Sigma; \emptyset \vdash V : \tau$ for some τ;
- $l : \tau \in \Sigma$ implies $(l = V) \in \mu$ and $\Sigma; \emptyset \vdash V : \tau$ for some V.

Theorem 2.5 (Subject Reduction)**.** $\Sigma;\Gamma \vdash M : \tau$ *and* $\Sigma \vdash \mu$ *and* $M \sharp \mu \Rightarrow N \sharp \mu'$ *imply* $\Sigma';\Gamma \vdash N : \tau$ *and* $\Sigma' \vdash \mu'$ *for some* $\Sigma' \supseteq \Sigma$*.*

Proof. Since rule (β_v) is a restriction of rule (β), we will give the proof in the case of $\Rightarrow_{\mathrm{n}}$, and the result will hold obviously for $\Rightarrow_{\mathrm{v}}$ too. It is clearly enough to consider the case $M \sharp \mu \longrightarrow N \sharp \mu'$.

$M \sharp \mu \longrightarrow N \sharp \mu'$ implies $M = E_{\mathrm{n}}[P]$, $N = E_{\mathrm{n}}[Q]$, for some E_{n}, P, Q. The proof is by induction on E_{n} and by cases on the applied reduction rule. We consider only $E_{\mathrm{n}} = [\,]$, since the other cases come directly by induction, observing that all labels in the domain of Σ' which are not in the domain of Σ must be fresh for M.

Let the applied rule be (β). Then $P = (\lambda x.P')Q'$ and $Q = P'[Q'/x]$. By Generation Lemma 2.3(3), $\Sigma;\Gamma \vdash P : \tau$ implies, for some I and τ_i, τ'_i $(i \in I)$, $\Sigma;\Gamma \vdash \lambda x.P' : \tau_i \to \tau'_i$ and $\Sigma;\Gamma \vdash Q : \tau_i$, for all $i \in I$, and $\bigcap_{i\in I} \tau'_i \leq \tau$. By Generation Lemma 2.3(2), $\Sigma;\Gamma \vdash \lambda x.P' :$

$\tau_i \to \tau_i'$ implies, for some J_i and ζ_j, ζ_j' $(j \in J_i)$, $\Sigma;\Gamma, x : \zeta_j \vdash P' : \zeta_j'$ for all $j \in J_i$, and $\bigwedge_{j\in J_i}(\zeta_j \to \zeta_j') \leq \tau_i \to \tau_i'$. By Lemma 2.2(3) for all $i \in I$ there is $K_i \subseteq J_i$ such that $\zeta_j \geq \tau_i$ for all $j \in K_i$ and $\bigcap_{j\in K_i} \zeta_j' \leq \tau_i'$. By rule $(\leq)$ from $\Sigma;\Gamma \vdash Q : \tau_i$ we get $\Sigma;\Gamma \vdash Q : \zeta_j$ for all $j \in K_i$. This together with $\Sigma;\Gamma, x : \zeta_j \vdash P' : \zeta_j'$ gives $\Sigma;\Gamma \vdash P'[Q'/x] : \zeta_j'$ for all $j \in K_i$ by the Substitution Lemma 2.4. We conclude by applying the rules $(\cap I)$ and $(\leq)$.

The case $P = \mathtt{fix}\, x.P'$ follows easily from the Generation Lemma 2.3(4) and the Substitution Lemma 2.4.

If $P = \mathtt{ref}\, V$ and $Q = l$ by the Generation Lemma 2.3(6) $\tau = \mathtt{Ref}\, \tau'$ and $\Sigma;\Gamma \vdash V : \tau'$. We take $\Sigma' = \Sigma, l : \tau'$ and we get $\Sigma' \vdash \mu'$ since $\Sigma;\Gamma \vdash V : \tau'$ implies $\Sigma;\emptyset \vdash V : \tau'$, being $V \in \Lambda^0_{imp}$.

The case $P = l := V$ and $Q = ()$ follow by Generation Lemma 2.3(8), (5), and (9). The other cases are easier. □

In order to prove the progress of our type system we need a Canonical Form Lemma which can be easily proved by analysing the typing rules.

Lemma 2.6 (Canonical Forms)**.** (1) $\Sigma;\emptyset \vdash V : \mathtt{pos}$ *implies* $V \in \{1, 2, \ldots\}$.
(2) $\Sigma;\emptyset \vdash V : \mathtt{nat}$ *implies* $V \in \{0, 1, 2, \ldots\}$.
(3) $\Sigma;\emptyset \vdash V : \mathtt{unit}$ *implies* $V = ()$.
(4) $\Sigma;\emptyset \vdash V : \tau \to \tau'$ *implies* $V = \lambda x.M$ *and* $\Sigma; x : \tau \vdash M : \tau'$ *for some* x, M.
(5) $\Sigma;\emptyset \vdash V : \mathtt{Ref}\, \tau$ *implies* $V = l$ *and* $l : \tau \in \Sigma$ *for some* l.

Theorem 2.7 (Progress)**.** $\Sigma;\emptyset \vdash M : \tau$ *implies that either* M *is a value or* $M \sharp \mu \Rightarrow N \sharp \mu'$ *for some* N *and for all* μ *such that* $\Sigma \vdash \mu$.

Proof. The proof is by induction on the derivation $\Sigma;\Gamma \vdash M : \tau$.

If the last applied rule is $(\to I)$, (*cell*), $(\mathtt{unit}_{()})$, $(\mathtt{nat})$, or $(\mathtt{pos})$, then M is a value.

If the last applied rule is $(\mathtt{fix})$ or $(\mathtt{Ref}\, I)$, then M is immediately reducible.

If the last applied rule is $(\mathtt{Ref}\, E)$, $(\mathtt{unit})$, $(\mathtt{if\ then\ else})$, or $(+)$ the proof using the Canonical Form Lemma 2.6 is standard, see Theorem 13.5.7 in [Pie02].

If the last applied rule is $(\to E)$, and the considered evaluation is the call-by-value, the proof is standard, using the Canonical Form Lemma 2.6 (see Lemma 9.3.4 in [Pie02]). The call-by-name case is simpler, since rule (β_v) is a restriction of rule (β).

For rule $(\cap I)$ induction applies remarking that the evaluation strategies are deterministic.

The case of rule $(\leq)$ is immediate by induction. □

As far as the typability power is concerned, the system preserves the typability power of intersection types for the pure λ-calculus. In fact we can easily adapt to the current system the well-know characterisation of strongly normalising λ-terms by means of intersection types [Pot80].

Theorem 2.8. *A pure* λ*-term* M *is typable in the system of Fig. 2 if and only if it is strongly normalising.*

3. Conclusion

In this paper we discuss how to combine intersection types and reference types in a meaningful way. The naive use of intersection types is unsound in presence of references,

as shown in [DP00] and in the Introduction. Davies and Pfenning solve the problem by restricting both the the definition of the preorder relation $\leq$ between types, and the type assignment system. In the preorder relation $\leq$ between types they do not allow the standard rule:

$$\frac{}{(\tau \to \tau_1) \wedge (\tau \to \tau_2) \leq \tau \to \tau_1 \wedge \tau_2} \ (\to \wedge)$$

The type assignment system is restricted in such a way that the intersection 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. While in this way they solve the problem cited in the introduction, the system allows for some 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:

$$\frac{\dfrac{\emptyset; x : \mathtt{nat} \wedge \mathtt{Ref\ nat} \vdash x : \mathtt{nat} \wedge \mathtt{Ref\ nat}}{\emptyset; x : \mathtt{nat} \wedge \mathtt{Ref\ nat} \vdash x : \mathtt{Ref\ nat}} \ (\leq) \qquad \dfrac{\dfrac{\emptyset; x : \mathtt{nat} \wedge \mathtt{Ref\ nat} \vdash x : \mathtt{nat} \wedge \mathtt{Ref\ nat}}{\emptyset; x : \mathtt{nat} \wedge \mathtt{Ref\ nat} \vdash x : \mathtt{nat}} \ (\leq)}{\emptyset; x : \mathtt{nat} \wedge \mathtt{Ref\ nat} \vdash x + 1 : \mathtt{nat}} \ (+)}{\emptyset; x : \mathtt{nat} \wedge \mathtt{Ref\ nat} \vdash x := x + 1 : \mathtt{unit}} \ (\mathtt{unit})$$

We use a totally different approach. In our type language reference and non-reference types are two incomparable classes. As an immediate result, the intersection between reference and non-reference types is not allowed, so, for example, $\mathtt{nat} \wedge \mathtt{Ref\ nat}$ is not a type. Technically this is obtained through a partially defined function on types, $\cap$, that, when applied to two types, builds their intersection if it is a correct type, and it is undefined otherwise. All the standard $\leq$ rules and properties on types are valid, when replacing $\wedge$ by $\cap$. As a results, our system enjoys classical subject reduction (both in case a call-by-name or a call-by-value evaluation is used), and unsound terms as the one shown before cannot be typed (but the sound versions $x :=!x + 1$ and $\mathtt{ref}\, x := x + 1$ are typable). Moreover, when restricted to the pure functional part of the language, our typing system has a stronger typability power. As an example, consider the strongly normalising pure λ-term $(\lambda xy.(\lambda z.zz)(xy))(\lambda t.t)$: it is typable in our system (see Theorem 2.8), while it is not typable in the system of [DP00]. In fact for typing it is necessary to introduce an intersection between two subderivations whose subject is xy, and in the system of [DP00] 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 $\emptyset; \{x : \sigma_1, y : \sigma_2\} \vdash xy : \sigma_2$, which requires to apply rule $(\cap I)$ to xy.

We plan to investigate how the present approach can be extended for dealing with parametric polymorphism and reference types. We think that a meaningful restriction would forbid universal quantification of reference types by introducing a suitable operator which plays the role of $\cap$ for intersection types.

Acknowledgement

The authors gratefully acknowledge fruitful discussions with Frank Pfenning and Betti Venneri.

References

[CDC80] Mario Coppo and Mariangiola Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980.

[Cop80] Mario Coppo. An Extended Polymorphic Type System for Applicative Languages. In Piotr Dembinski, editor, *MFCS'80*, volume 88 of *LNCS*, pages 194–204. Springer-Verlag, 1980.

[DP00] Rowan Davies and Frank Pfenning. Intersection Types and Computational Effects. In Philip Wadler, editor, *ICFP'00*, volume 35(9) of *SIGPLAN Notices*, pages 198–208. ACM Press, 2000.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pot80] Garrel Pottinger. A Type Assignment for the Strongly Normalizable λ-terms. In Roger Hindley and Jonathan P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, London, 1980.