# On the Semantics of Non-Interference Type-based Analyses

Frédéric Prost

*Laboratoire Leibniz-IMAG,*
*46 Av. Félix Viallet,*
*38 031 Grenoble Cedex 1, France*
Frederic.Prost@imag.fr

**Résumé**

Les analyses de programmes basées sur le typage sont courantes. Dans ces systèmes les types sont décorés avec des annotations. Nous nous intéressons au problème de la sémantique de ces annotations. Nous définissons une extension du système $F$ de Girard permettant de faire une analyse de non-interférence. Nous donnons une sémantique à base de relations d'équivalences partielles de ce nouveau système. À travers cette sémantique nous montrons que les annotations doivent être vues comme des *classes* de relation d'équivalences partielles (par exemple les relations triviales et diagonales sont deux classes différentes de relation d'équivalences partielles). Nous montrons aussi que l'analyse de programmes par le typage revient à abstraire par rapport aux classes de relation d'équivalences partielles que l'on utilise pour interpréter le programme.

## 1. Introduction

### 1.1. Static functional program analysis

Two approaches are commonly used for static analysis in the functional programming world : semantic based approach, via abstract interpretation (see [CC77, TJ92]), and type based approach (it has to be noted that according to [Mon95] type inference is an abstract interpretation). Since the early 90s a lot of type-based systems have been proposed (see for instance [Hei95, Sol95, NSN94, HM94, DP98, TJ92, ABHR99]) for many analyses like Binding Time, Strictness, Dead Code, Control Flow etc. Common features can be found in all these systems. A typed programming language is given. The type system is then modified by the addition of annotations on types or term constructors. Those annotations denote semantic properties of programs (for a survey see the introduction of [Sol95]). The general picture of this approach is as follows : the programmer writes a typed program, then his program is automatically re-typed under a slightly different type system which includes annotations.

The shape of typing rules becomes more or less the following :

$$\frac{\Gamma \vdash f : A^{\alpha} \to B^{\beta} \quad \Gamma \vdash a : A^{\alpha}}{\Gamma \vdash (f \quad a) : B^{\beta}}$$

where $\alpha$, $\beta$ are annotations, that is to say properties independent from the underlying type. For instance, an annotation may mean that a term is known at compile time or unknown. In such systems, inference rules are double purpose :

1. To insure terms well-formedness.
2. To analyse dependencies inside a term by generating constraints over annotations.

In [Pro99a, Pro99b, Pro00] we have shown that annotations may be understood, from a *logical point of view*, as sorts (the type of all types). Using this idea, it is possible to build a framework for type-based systems : it is shown in [ABHR99] that non-interference is a key property for various analyses. The aim of this work is to give, from a *semantic point of view*, an explanation of what is performed in such a framework. In other words, it is an attempt to understand how the typing discipline is able to find dependencies inside programs.

### 1.2. Overview of the paper

In section 2 we introduce a type system $\lambda_{\mathcal{S}_I}$. This type system can be seen as a framework to define many type-based analyses, indeed it is possible to give a non-interference result for it, as we see in section 3. In section 4 we give a semantics, based on PER semantics of system $F$, for the typing rules of $\lambda_{\mathcal{S}_I}$. Finally, we conclude in section 5.

## 2. A type system for non-interference : $\lambda_{\mathcal{S}_I}$

In this section we define a type system : $\lambda_{\mathcal{S}_I}$. It is an extended version of Girard's System $F$ in which there are several "types of all types" (called sorts). It is also possible to abstract terms relatively to sorts, thus there is a kind of polymorphism at the level of sorts. We also study the relationships between $\lambda_{\mathcal{S}_I}$ and System $F$.

### 2.1. $\lambda_{\mathcal{S}_I}$ definition

We define a type system, $\lambda_{\mathcal{S}_I}$, parameterised by a set $\mathcal{S}_I = \{*_i\}_{i \in I}$. In the following of this section we suppose given, although non specified a non empty set $\mathcal{S}_I$ having at least an element $*_\top$.

**Definition 1** *We inductively define 3 syntactical families ;* terms, types *and* sorts *by :*

$$
\begin{aligned}
t ::= \;& x \mid (t \;\; t) \mid (t \;\; A) \mid (t \;\; k) \mid \\
& \lambda x{:}A.t \mid \lambda X{:}k.t \mid \lambda\alpha{:}\Box.t \\
A ::= \;& X \mid A \to A \mid \Pi X{:}k.A \mid \Pi\alpha{:}\Box.A \\
k ::= \;& \alpha \mid *_i \mid \Box
\end{aligned}
$$

*where $x$, $X$, and $\alpha$ are term, type and sort variables, and $*_i \in \mathcal{S}_I$.*

We only consider $\beta$-reduction as computation rule :

**Definition 2 (Typed reduction)** *We have three kinds of redex :*

$$(\lambda x{:}A.t \;\; t') \;\to_\beta\; t[x := t']$$

$$(\lambda X{:}k.t \;\; A) \;\to_\beta\; t[X := A] \qquad (\lambda\alpha{:}\Box.t \;\; k') \;\to_\beta\; t[\alpha := k']$$

*With $\to_\beta^*$ we intend the reflexive, transitive $\to_\beta$ closure. $=_\beta$ is the smallest equivalence relation generated by $\to_\beta^*$.*

In this setting sort abstraction is natural, indeed :

- Unlike traditional type systems, where there is a unique type of all types (namely $*$), $\lambda_{\mathcal{S}_I}$ may have different "types of all types" (namely the elements of $\mathcal{S}_I$).
- There is a notion of sort *variable*.

therefore, it is possible to build terms depending on sort variables that can be instantiated by more than one value. Hence, the motivation for sort abstraction.

We define contexts : for each free term, type or sort variable a context associates a type or a sort.

**Definition 3 (Contexts)** *Contexts are defined by :* $\Gamma ::= <> \mid \Gamma, x : A \mid \Gamma, X : k \mid \Gamma, \alpha : \Box$ *where* $<>$ *is the empty list,* $x$*,* $X$ *and* $\alpha$ *are term, type and sort variables,* $A$ *is a type, and* $k$ *is a sort which is not* $\Box$*.*

For each syntactical family we define corresponding well-formedness rules. We start with contexts. A context $\Gamma$ is well-formed, if : $\Gamma \vdash ok$, can be derived from the following rules :

Context rules

$$\frac{}{<> \vdash ok}$$

$$\frac{\Gamma \vdash ok}{\Gamma, \alpha : \Box \vdash ok} \quad (\alpha \notin \Gamma)$$

$$\frac{\Gamma \vdash k : \Box}{\Gamma, X : k \vdash ok} \quad (X \notin \Gamma)$$

$$\frac{\Gamma \vdash A : k \quad \Gamma \vdash k : \Box}{\Gamma, x : A \vdash ok} \quad (x \notin \Gamma)$$

where $\alpha \notin \Gamma$ (resp. $X \notin \Gamma$, resp. $x \notin \Gamma$) means that $\alpha$ (resp. $X$, resp. $x$) does not occur freely in $\Gamma$. A sort $k$ is well-formed under context $\Gamma$ if : $\Gamma \vdash k : \Box$, can be derived from the following rules :

$\lambda_{\mathcal{S}_I}$ sorts formation rules

$$\frac{\Gamma \vdash ok}{\Gamma \vdash *_i : \Box} \qquad \frac{\Gamma \vdash ok}{\Gamma \vdash \alpha : \Box} \quad (\alpha : \Box \in \Gamma)$$

Therefore, the only well formed sorts are either sort variables $\alpha$, or elements of $\mathcal{S}_I$.

A type $A$ is well-formed of sort $k$ under context $\Gamma$ if : $\Gamma \vdash A : k$, can be derived from the following rules :

$\lambda_{\mathcal{S}_I}$ type formation rule

$$\frac{\Gamma \vdash ok}{\Gamma \vdash X : k} \quad X : k \in \Gamma$$

$$\frac{\Gamma, x : A, \Gamma' \vdash B : k \quad \Gamma, \Gamma' \vdash ok}{\Gamma, \Gamma' \vdash A \to B : k}$$

$$\frac{\Gamma, X : k, \Gamma' \vdash A : k' \quad \Gamma, \Gamma' \vdash ok}{\Gamma, \Gamma' \vdash (\Pi X : k . A) : k'}$$

$$\frac{\Gamma, \alpha : \Box, \Gamma' \vdash A : k \quad \Gamma, \Gamma' \vdash ok}{\Gamma, \Gamma' \vdash (\Pi \alpha : \Box . A) : k} \quad k \neq \alpha$$

$$\frac{\Gamma, \alpha : \Box, \Gamma' \vdash A : \alpha \quad \Gamma, \Gamma' \vdash ok}{\Gamma, \Gamma' \vdash (\Pi \alpha : \Box . A) : *_\top}$$

The last typing rule is is needed to avoid inconsistency. Indeed, if it were not present, it would be possible to have judgements of the form :

$$\Gamma \vdash (\Pi \alpha : \Box . A) : \alpha$$

Since $\alpha$ is linked in $\Pi\alpha{:}\Box.A$, it is possible through renaming to have the judgement :

$$\Gamma \vdash (\Pi\beta{:}\Box.A[\alpha := \beta]) : \alpha$$

which is nonsense relatively to the first one.

In $\lambda_{\mathcal{S}_I}$ it is possible to build types on different sorts. For instance the impredicative encoding of the type of natural numbers can be written :

$$\mathcal{N}^{*_i} \stackrel{def}{=} \Pi X{:}{*_i}.X \to (X \to X) \to X$$

therefore in $\lambda_{\mathcal{S}_I}$ it is possible to have many types that represent the type of natural numbers (one variation for each element of $\mathcal{S}_I$). The idea behind $\lambda_{\mathcal{S}_I}$ is to use different sorts to *annotate* programs. Let us suppose that $\mathcal{S}_I = \{*_\perp, *_\top\}$. Then $\mathcal{N}^{*_\perp}$ could be used to denote a sub-term of type $\mathcal{N}$ which is dead code. On the other hand, $\mathcal{N}^{*_\top}$ could be used to denote a sub-term of type $\mathcal{N}$ which is useful for the computation.

A term $t$ is well formed of type $A$ under context $\Gamma$ if : $\Gamma \vdash t : A$, can be derived from the following rules :

$\lambda_{\mathcal{S}_I}$ term typing rules

$$\frac{\Gamma \vdash ok}{\Gamma \vdash x : A} \quad x : A \in \Gamma$$

$$\frac{\Gamma, x : A, \Gamma' \vdash t : B \qquad \Gamma, \Gamma' \vdash ok}{\Gamma, \Gamma' \vdash (\lambda x{:}A.t) : A \to B}$$

$$\frac{\Gamma, X : k, \Gamma' \vdash t : B \qquad \Gamma, \Gamma' \vdash ok}{\Gamma, \Gamma' \vdash (\lambda X{:}k.t) : (\Pi X{:}k.B)}$$

$$\frac{\Gamma, \alpha : \Box, \Gamma' \vdash t : B \qquad \Gamma, \Gamma' \vdash ok}{\Gamma, \Gamma' \vdash (\lambda \alpha{:}\Box.t) : (\Pi \alpha{:}\Box.B)}$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash (t \;\; t') : B}$$

$$\frac{\Gamma \vdash t : \Pi X{:}k.A \qquad \Gamma \vdash B : k}{\Gamma \vdash (t \;\; B) : A[X := B]}$$

$$\frac{\Gamma \vdash t : \Pi\alpha{:}\Box.A \quad \Gamma \vdash k : \Box}{\Gamma \vdash (t \;\; k) : A[\alpha := k]}$$

As we have already seen, it is possible to build several types of natural numbers. For each of these types we can define the natural number $n$ as follows :

$$\overline{n}^i \stackrel{def}{=} \lambda X{:}{*_i}.\lambda x{:}X.\lambda f{:}X \to X.\;(\overbrace{f \ldots f}^{n} \quad x)$$

$\overline{n}^i$ is simply the Church encoding of numeral $n$ on sort $*_i$. To become more familiar with $\lambda_{\mathcal{S}_I}$ we recommend to the reader to see how multiple sorts influence traditional encodings of lists, trees, pairs, projections (see [GLT89] for instance)...

$\lambda_{\mathcal{S}_I}$ enjoys good properties (subject reduction, confluence, strong normalisation etc.). Since the aim of this paper is not the theoretical study of $\lambda_{\mathcal{S}_I}$, we refer to [Pro99b] for $\lambda_{\mathcal{S}_I}$ meta-properties.

## 2.2. relation between $\lambda_{\mathcal{S}_I}$ and System $F$

$\lambda_{\mathcal{S}_I}$ has strong links with Girard's system $F$ (see [Gir72] or [GLT89]). Actually, $\lambda_{\mathcal{S}_I}$ is a system $F$ extension. In order to precisely study the interactions between $\lambda_{\mathcal{S}_I}$ and $F$, we give a rapid presentation of the latter one.

System $F$ terms are a subset of $\lambda_{\mathcal{S}_I}$ terms. We give system $F$ typing rules in the following figure :

System $F$ typing rules

$$\frac{}{\Gamma \vdash_F x : A} \quad x : A \in \Gamma$$

$$\frac{\Gamma, x : A \vdash_F t : B}{\Gamma \vdash_F (\lambda x\!:\!A.t) : A \to B}$$

$$\frac{\Gamma, X : * \vdash_F t : A}{\Gamma \vdash_F (\lambda X\!:\!*.t) : (\Pi X\!:\!*.A)}$$

$$\frac{\Gamma \vdash_F t : A \to B \quad \Gamma \vdash_F t' : A}{\Gamma \vdash_F (t \ \ t') : B}$$

$$\frac{\Gamma \vdash_F t : (\Pi X\!:\!*.A)}{\Gamma \vdash_F (t \ \ B) : A[X := B]}$$

We now introduce a notion of skeleton. It is not hard to see that if we forget about sorts, $\lambda_{\mathcal{S}_I}$ is nothing more than System $F$. Actually for all $\lambda_{\mathcal{S}_I}$ terms there exist an underlying System $F$ term : its skeleton.

**Definition 4 (Skeleton)** *Function $|.|$, from $\lambda_{\mathcal{S}_I}$ terms (resp. types) toward system $F$ terms (resp. type), is inductively defined by :*

– *For types :*

$$\begin{array}{rcl} |X| & = & X \\ |\Pi X\!:\!k.A| & = & \Pi X\!:\!*.|A| \\ |A \to B| & = & |A| \to |B| \\ |\Pi\alpha\!:\!\Box.A| & = & |A| \end{array}$$

– *For terms :*

$$|x| = x$$

$$\begin{array}{rlrl} |(t \ \ t')| = & (|t| \ \ |t'|) & |(t \ \ A)| = & (|t| \ \ |A|) \\ |(t \ \ k)| = & |t| & |\lambda\alpha\!:\!\Box.t| = & |t| \\ |\lambda x\!:\!A.t| = & \lambda x\!:\!|A|.|t| & |\lambda X\!:\!k.t| = & \lambda X\!:\!*.|t| \end{array}$$

*We naturally extend $|.|$ to contexts :*

$$\begin{array}{rlrl} |\langle\rangle| = & \langle\rangle & |\Gamma, \alpha : k| = & |\Gamma| \\ |\Gamma, x : A| = & |\Gamma|, x : |A| & |\Gamma, X : k| = & |\Gamma|, X : * \end{array}$$

*If $t$ is a $\lambda_{\mathcal{S}_I}$ term, we say that $|t|$ is the skeleton of $t$ (similarly we talk about types or contexts skeleton).*

**Theorem 1** $\Gamma \vdash t : A \implies |\Gamma| \vdash_F |t| : |A|$

The proof is an induction on the form of the derivation.

Now, from a computational point of view we also have a link between system $F$ and $\lambda_{\mathcal{S}_I}$.

**Theorem 2** *If $t \ \to_\beta \ t'$ then $|t| \ \to_\beta^* \ |t'|$.*

The proof of this theorem is done by induction on the form of $t$. One has to remark that if the redex reduced from $t$ to $t'$ is of the form $(\lambda\alpha\!:\!\Box.t_1 \ \ t_2)$, then $|t| = |t'|$, so there is no reduction step in System $F$.

The meaning of theorems 1 and 2 is that $\lambda_{\mathcal{S}_I}$ extra features do not deeply modify System $F$. Indeed, sorts can be seen as annotations and sort abstractions as constraints between annotations, nothing more.

## 3. Non-interference result for $\lambda_{\mathcal{S}_I}$

In this section we show how $\lambda_{\mathcal{S}_I}$ can be viewed as a framework for non-interference type-based analyses. We give a result of non-interference for $\lambda_{\mathcal{S}_I}$. In [ABHR99], it is shown that non-interference is a cornerstone of many analyses like binding time analysis, dead code analysis, strictness analysis etc.

The problem of non-interference is the following. Given a term we address the question of what parts of this term are irrelevant regarding its evaluation. In other words we look at sub-terms that can be erased without affecting the evaluation of a term.

Since we are looking for two different kind of terms, we consider from now until the end of this paper that $\mathcal{S}_I$ is limited to two elements that is

$$\mathcal{S}_I \stackrel{def}{=} \{*_\bot, *_\top\}$$

#### 3.0.1. Scheme for non-interference result in $\lambda_{\mathcal{S}_I}$ :

How can we prove that some parts of a term are independent each one from another one in $\lambda_{\mathcal{S}_I}$ ? We solve this problem in a similar way to [Ber93] or [ALL96].

The idea can be divided into three steps :

1. We define dummy constants that are used to replace sub-terms that don't interfere with the evaluation of the program (they denote typed holes in terms).
2. We define an order relation on terms, based on dummy constants. We say that a term $u$ is smaller than another term $t$, if $u$ may be obtained from $t$ by replacing some of its sub-terms by dummy constants.
3. We define a notion of equivalence between programs and study how this equivalence behaves relatively to the order relation. Indeed, if two terms are related by the order, say $u \leq t$, and if those terms are equivalent, then all the sub-terms removed in $t$ to obtain $u$ do not interfere with the computation of $t$.

We start by the introduction of $i$-types. Those types are used to identify sub-terms that are going to be replaced with dummy constants. Then, we precisely define a notion of program equivalence, namely the observational equivalence. Finally, we state the non interference result for $\lambda_{\mathcal{S}_I}$.

#### 3.0.2. $i$-Types and $i$-pruning

**Definition 5 ($i$-types)** *A type $A$ is an $i$-type if there exists a context $\Gamma$ such that $\Gamma \vdash A : *_i$ and $A$ is :*

- *a variable.*
- *$A_1 \to A_2$ for some $i$-types $A_1$, $A_2$.*
- *$\Pi X : *_i . A_X$ for some $i$-type $A_X$.*

For each $\bot$-type $A$, we introduce a program constant. This constant is a canonical element denoting all programs of the same type.

**Definition 6 ($i$-constants)** *For any $i$-type $A$ we define constant $\mathbf{d}_A$. We add the typing rule*

$$<>\vdash \mathbf{d}_A : A$$

$\mathbf{d}_{\mathcal{N}^\perp}$, for instance, is the constant representing *any* naturals of type $\mathcal{N}^\perp$. Those constants are intended to be used to simulate the erasure of a sub-term. If it is possible to erase a sub-term without changing the global behavior of a program, then it means that this sub-term does not interfere with the program. One can see $i$-constant as typed holes (as the holes of [ALL96] prefixes for instance). We now define a relation between terms and types : the pruning relation.

**Definition 7 (Pruning relation)** *We inductively define the pruning relation $\leq_i$ for $i \in \{\top, \bot\}$ by :*

- *If $t : A$ and $A$ is an $i$-type, then $\mathbf{d}_A \leq_i t$.*
- *$x \leq_i x$ for all term variable $x$.*
- *If $t_1 \leq_i t_2$ and $t'_1 \leq_i t'_2$, then :*

$$(t_1 \;\; t'_1) \leq_i (t_2 \;\; t'_2) \qquad \lambda x\!:\!A_x.t_1 \leq_i \lambda x\!:\!A_x.t_2$$
$$\lambda X\!:\!k.t_1 \leq_i \lambda x\!:\!k.t_2 \qquad \lambda\alpha\!:\!\Box.t_1 \leq_i \lambda\alpha\!:\!\Box.t_2$$

In other words if $t_1 \leq_i t_2$, it means that $t_1$ matches $t_2$ except for some parts of $t_2$ that have been replaced with $i$-constants.

The reader has to be aware of the following peculiarity : terms like $(\mathbf{d}_{A\to B} \;\; \mathbf{d}_A)$ are not relevant. Indeed, the intended meaning of $i$-constants is to formalise the notion of "erasure". Therefore terms like $(\mathbf{d}_{A\to B} \;\; \mathbf{d}_A)$ should be replaced by $\mathbf{d}_B$ (one can notice that by definition if $A \to B$ is an $i$-type, so is $B$). We will only consider acceptable terms, that is terms where *maximal* sub-terms have been replaced by $i$-constants. More formally :

**Definition 8 ($i$-free terms)** *Let $t$ be a well formed $\lambda_{\mathcal{S}_I}$ term. $t$ is told $i$-free if for all $i$-type $A$, $\mathbf{d}_A$ does not occur in $t$. $t$ is told free if $t$ is $i$-free for all $i \in I$.*

**Definition 9 ($i$-saturated terms)** *Let $t$ be a well formed $\lambda_{\mathcal{S}_I}$ term of type $A$. $t$ is told $i$-saturated if*

- *If $A$ is an $i$-type : $t$ is $\mathbf{d}_A$.*
- *$A$ is not an $i$-type :*
  - *$t$ is a variable,*
  - *$t$ is of the form $(t_1 \;\; t_2)$, and $t_1$, $t_2$ are $i$-saturated,*
  - *$t$ is of the form $(t_1 \;\; A')$, and $t_1$ is $i$-saturated.*
  - *$t$ is of the form $(t_1 \;\; k)$, and $t_1$ is $i$-saturated.*
  - *$t$ is of the form $\lambda x\!:\!A.t_1$, and $t_1$ is $i$-saturated.*
  - *$t$ is of the form $\lambda X\!:\!k.t_1$, and $t_1$ is $i$-saturated.*
  - *$t$ is of the form $\lambda\alpha\!:\!\Box.t_1$, and $t_1$ is $i$-saturated.*

**Definition 10 ($i$-acceptable terms)** *Let $t$ be a well formed $\lambda_{\mathcal{S}_I}$ term $t$ is told $i$-acceptable if it is either $i$-free or $i$-saturated. If for all $i \in I$, $t$ is $i$-acceptable, then $t$ is acceptable.*

In the following of this paper we only consider acceptable terms.

### 3.0.3. Observational equivalence

We use the impredicative encoding of types as in [GLT89, Gir72]. A noticeable feature of $\lambda_{\mathcal{S}_I}$ lies in the fact that :

$$\mathsf{Bool}^i \stackrel{def}{=} \Pi X\!:\!*_i.X \to X \to X$$

It is the "standard" impredicative encoding of booleans, but we can choose a sort on which booleans are built. We define the two canonical inhabitants of booleans built on sort $*_i$ by :

$$\mathsf{True}^i \stackrel{def}{=} \lambda X\!:\!*_i.\lambda x_1\!:\!X.\lambda x_2\!:\!X.x_1$$

$$\mathsf{False}^i \stackrel{def}{=} \lambda X\!:\!*_i.\lambda x_1\!:\!X.\lambda x_2\!:\!X.x_2$$

We are now able to define the observational equivalence.

**Definition 11 (Observational Equivalence)** *Suppose $<>\vdash t, t' : A : *_\top$, we have $t =_{obs} t'$ if and only if :*
- *If $A = \mathsf{Bool}^i$ then, $t =_\beta t'$.*
- *Else for all $t_{obs}$ such that, $<>\vdash t_{obs} : A \to \mathsf{Bool}^i$, we have $(t_{obs} \ \ t) =_\beta (t_{obs} \ \ t')$.*

In other words, two programs are observationally equivalent if no experience (any function that takes the programs and returns a “yes/no” answer) is able to distinguish them.

#### 3.0.4. Non-interference result

**Theorem 3 (Main theorem)** *Let $<>\vdash t, t' : A$, with $A$ a $\top$-type, then if $t \leq_\perp t'$, one has : $t =_{obs} t'$*

The proof of this theorem is standard (regarding [Ber93] and [ALL96]). It can be found in [Pro99b].

From theorem 3 it is easy to show a non-interference property. In other words a term having a $\top$-type does not depend on its sub-terms of $\perp$-type.

**Theorem 4 (Non-interference theorem)** *Suppose $x : A \vdash t : B$, with $A$ a $\perp$-type and $B$ a $\top$-type, then for all terms $t_1$, $t_2$ such that $<>\vdash t_1, t_2 : A$ one has :*

$$t[x := t_1] =_{obs} t[x := t_2]$$

At this point the reader may notice that $*_\perp$ and $*_\top$ are interchangeable. Theorems 3 and 4 simply express the fact that sub-terms with different $i$-types live in different spaces : there are no interaction between them. So the analysis of a term in $\lambda_{\mathcal{S}_I}$ amounts to use as many different sort variables as possible. Then, regarding the analysis performed, one will instantiate them by appropriate sorts. Consider the following example :

**Example 1** *Program $t$ is well typed of type $\mathcal{N}^\top$ under the environment*

$$\Gamma = g : (\mathcal{N}^\top \to \mathcal{N}^\top) \to \mathcal{N}^\top \to \mathcal{N}^\top$$

$$\begin{aligned} t = (\ &\lambda f : (\Pi\alpha,\beta : \Box.\mathcal{N}^\alpha \to \mathcal{N}^\beta)(g \quad \begin{array}{l}(f \ \ *_\top \ \ *_\top) \\ ((f \ \ *_\perp \ \ \ *_\top) \ \ t'))\end{array} \\ &\lambda\alpha,\beta : \Box.\lambda x : \mathcal{N}^\alpha.\overline{5}^\beta) \end{aligned}$$

*In $t$, $g$ is a function taking two arguments : a function from naturals to naturals and a natural, both on sort $*_\top$. $f$ is a constant function taking a natural and giving 5 as result. The salient feature of $\lambda_{\mathcal{S}_I}$, abstraction over sorts, is enlighten in $f$. Indeed, $f$'s argument is supposed to be of $\alpha$-type which can be different from $f$'s result which has a $\beta$-type. The abstraction of $f$'s body relatively to $\alpha$, $\beta$ might be seen as a non-interference analysis of $f$. It is clear that one can now bind $\alpha$ to a given sort and $\beta$ to another one. In $t$, $f$ occurs in two different places. In the first one, $g$ “needs” a function of type $\mathcal{N}^\top \to \mathcal{N}^\top$, whereas in the second one $g$ only needs a term of type $\mathcal{N}^\top$. Therefore in the second occurrence we can use the fact that $f$'s output does not depend on $f$'s input to prove that $t'$ is useless (i.e. has a $*_\perp$-type). Now applying theorem 3 we have that $t =_{obs} t_{opt}$ with*

$$\begin{aligned} t_{opt} = (\ &\lambda f : (\Pi\alpha,\beta : \Box.\mathcal{N}^\alpha \to \mathcal{N}^\beta)(g \quad \begin{array}{l}(f \ \ *_\top \ \ *_\top) \\ ((f \ \ *_\perp \ \ *_\top) \ \ \mathbf{d}_{\mathcal{N}_\perp}))\end{array} \\ &\lambda\alpha,\beta : \Box.\lambda x : \mathcal{N}^\alpha.\overline{5}^\beta) \end{aligned}$$

# 4. Semantics of $\lambda_{\mathcal{S}_I}$ typing rules

In this section we give an interpretation of $\lambda_{\mathcal{S}_I}$ typing rules. The idea is to use a semantics of system $F$ based on Partial Equivalence Relations (PERs) (see for instance [MM85, BM84]). Indeed, we have seen that $\lambda_{\mathcal{S}_I}$ and system $F$ are close relatives so it is quite natural to rely on system $F$ semantics.

We adopt a presentation in the style of [Wad89]. We start by giving a semantics for system $F$ then, we show how $\lambda_{\mathcal{S}_I}$ extra features can be interpreted.

## 4.1. Frame semantics for system $F$

A type model is a tuple consisting in a universe $\mathcal{U}$ of type values, and two operations $\Rightarrow$ and $\forall$ that constructs types from over types. Let $[\mathcal{U} \to \mathcal{U}]$ denote the set of function (in a set theoretical sense) from $\mathcal{U}$ to $\mathcal{U}$. If $A,\ B$ are in $\mathcal{U}$, $A \Rightarrow B$ is in $\mathcal{U}$, and if $F$ is in $[\mathcal{U} \to \mathcal{U}]$, then $\forall F$ might be in $\mathcal{U}$.

**Definition 12 (Type environment)** *Let $\Gamma \vdash_F A : *$, we call* type environment*, any function (written $\mathfrak{E}$) from free $\Gamma$ type variables to $\mathcal{U}$.*

*$\mathfrak{E}[X := B]$, is the type environment such that $\mathfrak{E}[X := B](Y) = \mathfrak{E}(Y)$ for all $Y \neq X$, and $\mathfrak{E}[X := B](X) = B$.*

Type interpretation of $\Gamma \vdash_F A : *$ under type environment $\mathfrak{E}$ is defined as follows :

$$\boxed{\begin{array}{lcl} [\![X]\!]\mathfrak{E} & = & \mathfrak{E}(X) \\ [\![A \to B]\!]\mathfrak{E} & = & [\![A]\!]\mathfrak{E} \Rightarrow [\![B]\!]\mathfrak{E} \\ [\![\Pi X{:}*.A]\!]\mathfrak{E} & = & \forall(\lambda B.[\![A]\!]\mathfrak{E}[X := B]) \end{array}}$$

For each element $A$ of $\mathcal{U}$ we associate a corresponding domain of value : $\mathcal{V}_A$. The domain of all values is $\mathcal{V} = \cup_{A \in \mathcal{U}} \mathcal{V}_A$.
Let $A,\ B$ be in $\mathcal{U}$. Elements of $\mathcal{V}_{A \Rightarrow B}$, *represent* functions from $\mathcal{V}_A$ to $\mathcal{V}_B$. One has to be aware that, elements of $\mathcal{V}_{A \Rightarrow B}$ are not functions. Indeed, $[\mathcal{V}_A \to \mathcal{V}_B]$ is the set of functions (from a set theoretical point of view) from $\mathcal{V}_A$ to $\mathcal{V}_B$. Moreover there exists functions :

$$\phi_{A,B} : \mathcal{V}_{A \Rightarrow B} \to (\mathcal{V}_A \to \mathcal{V}_B) \qquad \psi_{A,B} : (\mathcal{V}_A \to \mathcal{V}_B) \to \mathcal{V}_{A \Rightarrow B}$$

such that $\phi_{A,B} \circ \psi_{A,B}$ is identity on $[\mathcal{V}_A \to \mathcal{V}_B]$. In the following we will omit subscripts and write $\phi$ and $\psi$.

Let $F$ be a function of $[\mathcal{U} \to \mathcal{U}]$, $\mathcal{V}_{\forall F}$ elements represent function that given a type $A$ associates an element of $\mathcal{V}_{F(A)}$. The set of function that for all $A$ in $\mathcal{U}$ associates an element of $\mathcal{V}_{F(A)}$ is written $[\forall A : \mathcal{U}.\mathcal{V}_{F(A)}]$. Moreover there exists functions :

$$\Phi_F : \mathcal{V}_{\forall A} \to [\forall B : \mathcal{U}.\mathcal{V}_{F(A)}] \qquad \Psi_F : [\forall A : \mathcal{U}.\mathcal{V}_{F(A)}] \to \mathcal{V}_{\forall A}$$

such that $\Phi_F \circ \Psi_F$ is identity on $[\forall A : \mathcal{U}.\mathcal{V}_{F(A)}]$. In the following we will omit the subscripts and just write $\Phi$ and $\Psi$.

**Definition 13 (Term environment)** *Let $\Gamma \vdash_F t : A$, we call* term environment*, any function $\mathfrak{e}$ associating $\mathcal{V}$ elements to $\Gamma$ free term variables.*

*$\mathfrak{e}[x := t']$ is the term environment such that $\mathfrak{e}(y)[x := t'] = \mathfrak{e}(y)$ and $\mathfrak{e}(x)[x := t'] = t'$.*

Let $\Gamma \vdash_F t : A$. We say that environments $\mathfrak{E},\ \mathfrak{e}$ are respecting $\Gamma$ if for all $x_i : A_i$ in $\Gamma$, we have that $\mathfrak{E}(x_i) \in \mathcal{V}_{[\![A_i]\!]\mathfrak{E}}$.
Interpretation of a term $\Gamma \vdash_F t : A$ under type and term environment $\mathfrak{E}$, $\mathfrak{e}$ is defined as follows :

$$
\begin{array}{|lcl|}
\hline
[\![x]\!]\mathfrak{E}\mathfrak{e} & = & \mathfrak{e}(x) \\
[\![\lambda x\!:\!A.t]\!]\mathfrak{E}\mathfrak{e} & = & \psi(\lambda a.[\![t]\!]\mathfrak{E}\mathfrak{e}[x := a]) \\
[\![(t \;\; A)]\!]\mathfrak{E}\mathfrak{e} & = & \phi([\![t]\!]\mathfrak{E}\mathfrak{e})([\![A]\!]\mathfrak{E}\mathfrak{e}) \\
[\![\lambda X\!:\!*.t]\!]\mathfrak{E}\mathfrak{e} & = & \Psi(\lambda A.[\![t]\!]\mathfrak{E}[X := A]\mathfrak{e}) \\
[\![(t \;\; t')]\!]\mathfrak{E}\mathfrak{e} & = & \Phi([\![t]\!]\mathfrak{E}\mathfrak{e})([\![t']\!]\mathfrak{E}\mathfrak{e}) \\
\hline
\end{array}
$$

We write $\Gamma \models t : A$ if for all environments $\mathfrak{E}$, $\mathfrak{e}$ respecting $\Gamma$ we have $[\![t]\!]\mathfrak{E}\mathfrak{e} \in [\![A]\!]\mathfrak{E}$. It is then possible to show type soundness for system $F$ :

**Theorem 5 (System $F$ type soundness)** *If $\Gamma \vdash_F t : A$ then $\Gamma \models t : A$.*

## 4.2. Types seen as relations

In the previous section we have given an interpretation of system $F$ types as sets. In fact it is also possible to see types as *relations* (see [Wad89]). In this section we recall definitions of logical relations (relation that represent types). We start by giving our notations and then give definitions for type constructors.

### 4.2.1. Relations

**Lexicography 1 (Relation notation)** *In order to improve the readability of this paper we adopt the following conventions :*
- *Relations, $\overline{A}$, are over-lined.*
- *Sets, $\mathbf{A}$ are written in bold-file.*

A relation $\overline{A}$ between sets $\mathbf{A}$, $\mathbf{A'}$ is a subset of $\mathbf{A} \times \mathbf{A'}$. We write $\overline{A} : \mathbf{A} \leftrightarrow \mathbf{A'}$. If $(x, x') \in \overline{A}$, we write $x \overset{\overline{A}}{\longleftrightarrow} x'$.

Let $\mathbf{A}$ be a set. An equivalence relation (ER), $\overline{A}$, on $\mathbf{A}$ is transitive, symmetric and reflexive. A partial equivalence relation (PER) is only transitive and symmetric. One can consider PER $\overline{A}$ on a set $\mathbf{A}$ as an ER on a subset of $\mathbf{A}$ : this subset is called the domain of $\overline{A}$ and is written $\mathcal{D}(\overline{A})$.

For any set $\mathbf{A}$, we define :
- $\overline{\Delta}(\mathbf{A})$, the diagonal relation on $\mathbf{A}$ defined by $\overline{\Delta}(\mathbf{A}) = \{(x, x) \mid x \in \mathbf{A}\}$,
- $\overline{\Theta}(\mathbf{A})$, the trivial relation $\mathbf{A}$, defined by $\overline{\Theta}(\mathbf{A}) = \{(x, y) \mid x, y \in \mathbf{A}\}$,
- $\mathsf{REL}(\mathbf{A})$, (resp. $\mathsf{PER}(\mathbf{A})$) the set of all relations (resp. PER) on $\mathbf{A}$,
- $\mathsf{REL}_\Theta(\mathbf{A})$, (resp. $\mathsf{REL}_\Delta(\mathbf{A})$) the set of all trivial (resp. diagonal) relations on subsets of $A$.

### 4.2.2. Relation for type constructors

We now give definitions for all type constructors of system $F$, that is $\rightarrow$ and $\Pi$. We give relational alternatives that are respectively $\Rightarrow$ and $\forall$.

**Definition 14 (Relation $\Rightarrow$)** *Let $\overline{A} : \mathbf{A} \leftrightarrow \mathbf{A'}$, $\overline{B} : \mathbf{B} \leftrightarrow \mathbf{B'}$ be two relations. We define relation $\overline{A} \Rightarrow \overline{B} : [\mathbf{A} \rightarrow \mathbf{B}] \leftrightarrow [\mathbf{A'} \rightarrow \mathbf{B'}]$ as the relation defined by :*

$$\overline{A} \Rightarrow \overline{B} = \{(f, f') \mid \forall (a, a') \in \overline{A}, (\phi(f)(a), \phi(f')(a')) \in \overline{B}\}$$

In other words two functions are related by a $\Rightarrow$ relation if they map related arguments into related results.

**Definition 15 (Relation $\forall$)** *Let $F$, $F'$ be two elements of $[\mathcal{U} \to \mathcal{U}]$. For each $\mathbf{A}$, $\mathbf{A}'$ in $\mathcal{U}$, let $\overline{F}$ be a function taking a relation $\overline{A} : \mathcal{V}_\mathbf{A} \leftrightarrow \mathcal{V}_{\mathbf{A}'}$ as argument and giving back a relation $\overline{F}(\overline{A}) : \mathcal{V}_{F(\mathbf{A})} \leftrightarrow \mathcal{V}_{F'(\mathbf{A}')}$. We define relation $\forall\overline{F} : \forall F \leftrightarrow \forall F'$ as the relation defined by :*

$$\forall\overline{F} = \begin{array}{ll} \{(f,f') \mid & \textit{for all } \mathbf{A},\mathbf{A}',\ \textit{and } \overline{A} : \mathbf{A} \leftrightarrow \mathbf{A}', \\ & (\Phi_F(f)(\mathbf{A}), \Phi_F(f')(\mathbf{A}')) \in \overline{F}(\overline{A})\} \end{array}$$

In other words, type abstraction are related if they map related types into related results.

## 4.3. Frame semantics for $\lambda_{\mathcal{S}_I}$

We have seen in section 2.2 that system $F$ and $\lambda_{\mathcal{S}_I}$ are close relatives. They principally differ on the two following points :

1. There are many sorts to build types in $\lambda_{\mathcal{S}_I}$ whereas there is only one in system $F$.
2. It is possible to abstract sorts over terms in $\lambda_{\mathcal{S}_I}$.

The aim of this section is to give an interpretation for those two points.

In the previous section we have shown how system $F$ types can be interpreted. We have also shown how types can be viewed as relations. Now, since $*$ is the type of all type, it would seem natural to interpret $*$ as the union of all type interpretations. If we were to interpret types as relations over $\mathcal{V}$, we would set $[\![*]\!] = \mathsf{PER}(\mathcal{V})$ (indeed logical relations are PERs, see [Wad89]). It is then easy to give a semantics for type building rules : $\Gamma \vdash_F A : *$ because $[\![A]\!]\mathfrak{E}$ is trivially an element of $[\![*]\!]$.

Now the question is the following : what meaning can we give to two different sorts $*_\top$ and $*_\perp$? What are the differences between types of same skeleton $\mathsf{Bool}^\top$ and $\mathsf{Bool}^\perp$? The idea is that each $*_i$ is interpreted as a *class* of PERs. Let $\mathbf{B}$ denote the domain of $[\![\mathsf{Bool}]\!]$. Then, there are several PERs on $\mathbf{B}$ that are able to interpret type $\mathsf{Bool}$. One can say that $*_\top$ types are types where each element is equal to itself. On the other hand one can say that $*_\perp$ types are dummy types where all elements are equated together. So $*_\top$ is interpreted as $\overline{\Delta}(\mathcal{V})$ and $*_\perp$ as $\overline{\Theta}(\mathcal{V})$. Interpretations of $\mathsf{Bool}^\top$ and $\mathsf{Bool}^\perp$ are *PERs having a common domain while being different.*

Now what can be the interpretation of sort abstraction? In $\lambda_{\mathcal{S}_I}$ one can build terms like : $t = \lambda\alpha{:}\Box.\lambda x{:}\mathcal{N}^\alpha.x$. The only way to instantiate $\alpha$ is to replace it with a sort $*_i$. We have seen that $*_i$ is interpreted as *some PER class.* Thus, abstracting over sort variables amount to abstract relatively to the PER class used to interpret types. Since the domain of $\alpha$ is finite, we can use a naive interpretation of products and say that $\Pi\alpha{:}\Box.\Pi X{:}\alpha.X \to X$ (the type of $t$) must be a type for all $\alpha$. One has just to consider the intersection of all PERs on the domain of type $\Pi X{:}*.X \to X$. It is in fact the most precise PER : the diagonal one (the one identifies by $*_\top$).

Term interpretation remains the same : roughly the interpretation of a $\lambda_{\mathcal{S}_I}$ term will be the one of its skeleton.

We suppose given although non specified a frame semantic for system $F$.

### 4.3.1. Sorts interpretation

We now give an interpretation of $\lambda_{\mathcal{S}_I}$ sorts. Since sorts are the types of all types, and since we want to interpret types as relations, sorts are going to be seen as sets of relations. We distinguish two class of relations : trivial and diagonal relations.

**Definition 16 (Sort Context)** *Let* $\Gamma \vdash k : \Box$*, we call* sort environment *any function (written* $\mathfrak{s}$*) from free* $\Gamma$ *sort variables to the set* $\{\mathsf{REL}_\Delta(\mathcal{V}), \mathsf{REL}_\Theta(\mathcal{V})\}$*.*

$\mathfrak{s}[\alpha := \mathsf{R}]$ *is the sort environment such that* $\mathfrak{s}(\alpha')[\alpha := \mathsf{R}] = \mathfrak{s}(\alpha')$*, for all* $\alpha \neq \alpha'$*, and* $\mathfrak{s}[\alpha := \mathsf{R}](\alpha) = \mathsf{R}$*.*

Interpretation of a sort $\Gamma \vdash k : \Box$ under a sort environment $\mathfrak{s}$ is defined by :

$$
\begin{array}{lcllcl}
[\![\alpha]\!]\mathfrak{s} & = & \mathfrak{s}(\alpha) & [\![*_\top]\!]\mathfrak{s} & = & \mathsf{REL}_\Delta(\mathcal{V}) \\
[\![*_\bot]\!]\mathfrak{s} & = & \mathsf{REL}_\Theta(\mathcal{V}) & [\![\Box]\!]\mathfrak{s} & = & \mathsf{PER}(\mathcal{V})
\end{array}
$$

We write $\Gamma \models k : \Box$ if for all sort environment $\mathfrak{s}$, one has $[\![k]\!]\mathfrak{s} \in [\![\Box]\!]\mathfrak{s}$.

**Theorem 6** $\Gamma \vdash k : \Box$ *implies* $\Gamma \models k : \Box$*.*

### 4.3.2. $\lambda_{\mathcal{S}_I}$ type interpretation

We now give an interpretation of $\lambda_{\mathcal{S}_I}$ types. For this interpretation we introduce the notion of relation environment.

**Definition 17 (relation environment)** *A* relation environment *maps type variable to relation over* $\mathcal{V}$*. Let* $\mathfrak{E}$*,* $\mathfrak{E}'$ *be two type environments for context* $\Gamma$*. We write* $\overline{\mathfrak{R}} : \mathfrak{E} \leftrightarrow \mathfrak{E}'$ *the relation such that for all type variable* $X$ *in* $\Gamma$*, we have* $\overline{\mathfrak{R}(X)} : \mathcal{V}_{\mathfrak{E}(\mathbf{X})} \leftrightarrow \mathcal{V}_{\mathfrak{E}'(\mathbf{X})}$*.*

Let $\Gamma \vdash A : k$, we say that relation and sort environments $\mathfrak{R}$, $\mathfrak{s}$ respect $\Gamma$ if for all $X_i : k_i$ we have $\mathfrak{R}(X_i) \in \mathfrak{s}(k_i)$. Indeed, each type variable $X_i$ defined in $\Gamma$ belongs to a sort $k_i$ and thus cannot be interpreted as any relation.

Interpretation of a type, $\Gamma \vdash A : k$, with sort and relation environments $\mathfrak{s}$, $\mathfrak{R}$ is defined by :

$$
\begin{array}{lcl}
[\![X]\!]\mathfrak{s}\mathfrak{R} & = & \mathfrak{R}(X) \\
[\![A \to B]\!]\mathfrak{s}\mathfrak{R} & = & [\![A]\!]\mathfrak{s}\mathfrak{R} \Rightarrow [\![B]\!]\mathfrak{s}\mathfrak{R} \\
[\![\Pi X : k.A]\!]\mathfrak{s}\mathfrak{R} & = & \forall(\lambda \overline{B} \in [\![k]\!]\mathfrak{s}.[\![A]\!]\mathfrak{s}\mathfrak{R}[X := \overline{B}]) \\
[\![\Pi \alpha : \Box.A]\!]\mathfrak{s}\mathfrak{R} & = & [\![A]\!]\mathfrak{s}[\alpha := \mathsf{REL}_\Delta(\mathcal{V})]\mathfrak{R}
\end{array}
$$

We write $\Gamma \models A : k$ if for all environments $\mathfrak{s}$, $\mathfrak{R}$ respecting $\Gamma$, one has $[\![A]\!]\mathfrak{s}\mathfrak{R} \in [\![k]\!]\mathfrak{s}$.

**Theorem 7** $\Gamma \vdash A : k \implies \Gamma \models A : k$

### 4.3.3. Terms frame semantics

Roughly, $\lambda_{\mathcal{S}_I}$ term interpretation corresponds to the interpretation of skeletons. Indeed, as we have seen in section 2.2 terms that sorts do not influence the computational behaviour of terms.

Let $\mathfrak{e}$ be a term valuation, interpretation of a term is defined as follows :

$$
\begin{array}{lcl}
[\![x]\!]\mathfrak{E}\mathfrak{e} & = & \mathfrak{e}(x) \\
[\![\lambda x : A.t]\!]\mathfrak{E}\mathfrak{e} & = & \psi(\lambda a.[\![t]\!]\mathfrak{E}\mathfrak{e}[x := a]) \\
[\![(t \;\; t')]\!]\mathfrak{E}\mathfrak{e} & = & \phi([\![t]\!]\mathfrak{E}\mathfrak{e})([\![t']\!]\mathfrak{E}\mathfrak{e}) \\
[\![\lambda X : k.t]\!]\mathfrak{E}\mathfrak{e} & = & \Psi(\lambda A.[\![t]\!]\mathfrak{E}[X := A]\mathfrak{e}) \\
[\![(t \;\; A)]\!]\mathfrak{E}\mathfrak{e} & = & \Phi([\![t]\!]\mathfrak{E}\mathfrak{e})([\![A]\!]\mathfrak{E}\mathfrak{e}) \\
[\![\lambda \alpha : \Box.t]\!]\mathfrak{E}\mathfrak{e} & = & [\![t]\!]\mathfrak{E}\mathfrak{e} \\
[\![(t \;\; k)]\!]\mathfrak{E}\mathfrak{e} & = & [\![t]\!]\mathfrak{E}\mathfrak{e}
\end{array}
$$

Environments $\mathfrak{E}$, $\mathfrak{e}$ respect context $\Gamma$, if for all $x : A \in \Gamma$ one has $\mathfrak{e}(x)$ an element of $\mathcal{V}_{[\![A]\!]\mathfrak{E}}$, . In this case we write $\mathfrak{E}, \mathfrak{e} \vdash_F \Gamma$.

Let $\mathfrak{e}$ and $\mathfrak{e}'$ be two term environments, and $\mathfrak{E}$, $\mathfrak{E}'$ two type environments both respecting $\Gamma$, and $\mathfrak{s}$ a sort environment. $\mathfrak{R}$, $\mathfrak{E}$, $\mathfrak{e}$, $\mathfrak{E}'$, $\mathfrak{e}'$, $\mathfrak{s}$ *respect* $\Gamma$ if for all $(x : A) \in \Gamma$, $([\![x]\!]\mathfrak{E}'\mathfrak{e}', [\![x]\!]\mathfrak{E}\mathfrak{e}) \in [\![A]\!]\mathfrak{s}\mathfrak{R}$, and for all $(X : k) \in \Gamma$, $[\![X]\!]\mathfrak{s}\mathfrak{R} \in [\![k]\!]\mathfrak{s}$.

**Theorem 8 ($\lambda_{\mathcal{S}_I}$ type soundness)** *If* $\Gamma \vdash t : A$ *then* $\Gamma \models t : A$.

### 4.4. Discussion

We have given an interpretation for $\lambda_{\mathcal{S}_I}$ typing rules in the particular case when $\mathcal{S}_I = \{*_\top, *_\bot\}$. This interpretation has obvious links with previous ones [BB95, HS91, CDG98], but it is done here in a polymorphic second order language (whereas the previous ones were for simply typed calculi). The non-interference property is well represented : a term built on sort $*_\bot$ is undistinguishable from any other term with same type. $i$-constants introduced in section 3.0.2 can now be understood as canonical element of an equivalence class. On the other hand a term built on sort $*_\top$ is only equal to itself.

Now, uniquely by semantic means we can understand why $\lambda_{\mathcal{S}_I}$ can be used as a non interference calculus. Suppose that you have a term $t$ having a type built on sort $*_\top$ and having a sub-term $s$ having a type built on type $*_\bot$. Then $s$ can be equal to every term of the same type *without changing the semantics of* $t$. Therefore, $s$ do not interfere with $t$.

The semantics also gives a meaning to types built on variable sorts. It means that the interpretation of the type is not yet fixed. To abstract sorts amounts to abstract relatively to the interpretation of types. And indeed, it is exactly what happens when one analyses a program. For instance, let $t$ be of type $\mathcal{N}$. Now, in the following terms : $(\lambda x.5 \;\; t)$ and $(\lambda x.x \;\; t)$, $t$ can be both relevant and irrelevant regarding its context. Thus it has to be typed on both sorts $*_\bot$ and $*_\top$.

One can sum up the results of this section into the saying :

**To analyse a program = to abstract over its interpretation**

Indeed, as we have seen in section 3, the type-based analyses of non-interference can be seen as abstraction over sorts. We have shown in this section that abstraction over sorts may be seen as abstraction over class of PERs used to interpret a type. So our conclusions are that, from a conceptual point of view, all analyses performed via type-based systems are more or less similar : in all of them the addition of annotations over types corresponds to the choice between PERs interpreting this type.

## 5. Conclusion

We have presented a system to analyse non-interference properties for polymorphic second order $\lambda$-calculus. This system can be viewed as a framework to define many type-based analyses like dead code, binding time, slicing etc. We have given a semantics for $\lambda_{\mathcal{S}_I}$ typing rules based on PER interpretation of types.

This semantics precisely explains what means to perform an analysis using a type-based approach : the addition of annotation to types (or variable as in polyvariant analyses) amounts to choose between several interpretations. From a philosophical point of view it is quite natural. In itself a piece of code is neither dead-code nor useful code. It depends on the context where this piece of code occurs. The only thing we know for sure is that this piece of code, for instance, gives a natural number as result. But out of any context, we have to abstract over its usefulness (i.e. the class of PER whom depends the interpretation of the type).

This work suggests to push further investigations to see if it is possible to represent more complex abstract domains in order to perform more complex analyses with type-based systems (one can for instance imagine a set $\mathcal{S}_I$ together with an order relation on its elements). Also adding sub-typing to typing rules comes to mind. Indeed diagonal PERs are included into trivial PERs, so the same semantics would also give a meaning to a sub-typing rule based on sort ($*_\top \subseteq *_\bot$).

## Références


[ABHR99] M. Abadi, N. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, pages 147–160, January 1999.

[ALL96] M. Abadi, J.-J. Lévy, and B. Lampson. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM International Conference on Functional Programming*, pages 83–91, May 1996.

[BB95] S. Berardi and L. Boerio. Using subtyping in program optimization. In *Proceedings of TLCA'95, LNCS 902*. Springer-Verlag, 1995.

[Ber93] S. Berardi. Pruning simply typed $\lambda$–terms. Technical report, Turin University, 1993.

[BM84] K. B. Bruce and A. R. Meyer. The semantics of second-order polymorphic lambda calculus. In Kahn, MacQueen, and Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1984.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (*POPL '77*), pages 238–252, New York, 1977. ACM Press.

[CDG98] M. Coppo, F. Damiani, and P. Giannini. A type inference approach to program analysis. In *Proceedings of the international workshop : "Theories of Types and Proofs"*. Mathematical Society of Japan, 1998.

[DP98] F. Damiani and F. Prost. Detecting and removing dead code using rank-2 intersection. In *International Workshop :"TYPES'96", selected papers*, volume 1512 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[Gir72] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[Hei95] N. Heintze. Control-flow analysis and type systems. In Alan Mycroft, editor, *Proceeding of SAS 1995, LNCS 983*, pages 189–206. Springer-Verlag, 1995.

[HM94] C. Hankin and D. Le Métayer. A type-based framework for program analysis. In *Proceedings of the Static Analysis Symposium, LNCS 864*, pages 380–394. Springer-Verlag, 1994.

[HS91] S. Hunt and D. Sands. Binding time analysis : A new PERspective. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, June 1991.

[MM85] J.C. Mitchell and A. R. Meyer. Second-order logical relations. In R. Parikh, editor, *Logics of Programs, LNCS 193*, pages 225–236. Springer-Verlag, 1985.

[Mon95] B. Monsuez. System F and abstract interpretation. In A. Mycroft, editor, *2nd Static Analysis Symposium, SAS'95*, volume 983 of *Lecture Notes in Computer Science*, pages 279–295. Springer-Verlag, September 1995.

[NSN94] H.R. Nielson, K.L. Solberg, and F. Nielson. Strictness and totality analysis. In *Static Analysis, LNCS 864*, pages 408–422. Springer-Verlag, 1994.

[Pro99a] F. Prost. A formalization of static analyses in system F. In H. Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, Trento, Italy, July 7–10, 1999. Springer-Verlag.

[Pro99b] F. Prost. *Interprétation de l'analyse statique en théorie des types.* PhD thesis, École Normale Supérieure de Lyon, december 1999.

[Pro00] F. Prost. A static calculus of dependencies for the λ-cube. In *Proc. of IEEE 15th Ann. Symp. on Logic in Computer Science (LICS'2000).* IEEE Computer Society Press, 2000.

[Sol95] K. Lackner Solberg. *Annotated Type Systems for Program Analysis.* PhD thesis, Odense University, July 1995.

[TJ92] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programing*, 2(2), 1992.

[Wad89] P. Wadler. Theorems for free! In *Proc. of the ACM Conf. on Func. Progr. and Comp. Arch.*, pages 347–359, 1989.