Electronic Notes in Theoretical Computer Science 70 No. 1 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 16 pages

Polar Type Inference with Intersection Types and ω

Sébastien Carlier [1,2]

EPITA Research and Development Laboratory
14-16 rue Voltaire F-94276 Le Kremlin-Bicetre France
phone +33 1 53 14 59 16 - fax +33 1 53 14 59 22

Abstract

We present a type system featuring intersection types and ω, a type constant which is assigned to unused terms. We exploit and extend the technology of expansion variables from the recently developed System **I**, with which we believe our system shares many interesting properties, such as strong normalization, principal typings, and compositional analysis. Our presentation emphasizes a polarity discipline and shows its benefits. We syntactically distinguish positive and negative types, and give them different interpretations. We take the point of view that the interpretation of a type is intrinsic to it, and should not change implicitly when it appears at the opposite polarity. Our system is the result of a process which started with an extension of Trevor Jim's Polar Type System.

1 Introduction

1.1 Background and Motivation

Designing a programming languages involves making many choices, and a crucial part is finding a type system which achieves good balance between complexity and expressivity. Complex analyses usually lead to long compilation times, so the ability to perform type inference in a compositional way — allowing for incremental compilation — is also highly desirable. A system with principal typings can support this feature in an elegant way [5].

1.1.1 Algorithm $\mathcal{W}$

Languages such as ML [3] and Haskell [7] use the popular algorithm $\mathcal{W}$ by Damas and Milner [1], which relies on restricted universal quantification to

[1] This work was partly supported by NATO grant CRG 971607.
[2] Email: sebc@lrde.epita.fr.

obtain polymorphism. Sometimes these languages require the programmer to add type annotations or extraneous data types to bypass their limitations, possibly hindering readability and runtime performance. Also, this algorithm works in a top-down fashion and does not have the principal typings property, thus does not permit incremental compilation.

*1.1.2 System **P***

The search for a simple and expressive type system allowing for total type inference and compositional analysis led us to consider type systems combining universal quantification and intersection types, which is another way to obtain polymorphism.

In [6], Trevor Jim introduces System **P**, which is designed around the duality between positive types (which express capabilities) and negative types (which express requirements).

System **P** is a restriction of the combined system of intersection types and universal type quantification. Intersections are restricted to appear only at negative positions, and quantifiers only at positive positions.

It attempts to minimize requirements, and maximize capabilities, by placing the weakest possible types at negative occurrences and the strongest possible types at positive occurrences.

In order to preserve the polarity discipline, System **P** requires variables to be instantiated to simple types (types without $\forall$ and $\wedge$). Although this system types more terms than algorithm $\mathcal{W}$ (see table 1 for some examples), this restriction is annoying in practice, because it means, for example, that for any typable term M, the term $(\lambda x.x)M$ has to be given a simple type.

*1.1.3 System **P'***

We first developed an extension of System **P** (we call it **P'**, for lack of a better name) which can instantiate type variables to polymorphic types, exploiting the property of the system that in principal typings, type variables have exactly one positive occurence and one negative occurence. The resulting system is able to type more terms than System **P**, but is undecidable (in its unrestricted version), and is still unable to type some code that arises in practice. For instance, this system is unable to type the last example shown in table 1:

$$(\lambda f.(\lambda x.xx)(\lambda y.fy))(\lambda z.z)$$

The principal typing $\{f : \alpha \to \beta\} \vdash (\lambda y.fy) : \alpha \to \beta$ would be derived, and later refined to $\{f : (\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2)\} \vdash (\lambda y.fy) : (\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2)$. This typing is too weak to type the above term, we would like to be able to infer $\{f : (\alpha_1 \to \beta_1) \wedge (\alpha_2 \to \beta_2)\} \vdash (\lambda y.fy) : (\alpha_1 \to \beta_1) \wedge (\alpha_2 \to \beta_2)$.

$\lambda f.f(\lambda x.x)$	$\mathcal{W}$	$\forall\alpha,\beta.((\alpha\to\alpha)\to\beta)\to\beta$
	P	$\forall\beta.((\forall\alpha.\alpha\to\alpha)\to\beta)\to\beta$
	P'	$\forall\beta.((\forall\alpha.\alpha\to\alpha)\to\beta)\to\beta$
	I	$(F(\alpha\to\alpha)\to\beta)\to\beta$
$\lambda y.yy$	$\mathcal{W}$	not typable
	P	$\forall\alpha,\beta.((\beta\to\alpha)\land\beta)\to\alpha$
	P'	$\forall\alpha.(\exists\beta.(\beta\to\alpha)\land\beta)\to\alpha$
	I	$((F\beta\to\alpha)\land F\beta)\to\alpha$
$(\lambda f.f(\lambda x.x))(\lambda y.yy)$	$\mathcal{W}$	not typable
	P	$\forall\alpha.\alpha\to\alpha$
	P'	$\forall\alpha.\alpha\to\alpha$
	I	$\alpha\to\alpha$
$(\lambda x.x)(\lambda y.yy)$	$\mathcal{W}$	not typable
	P	not typable
	P'	$\forall\alpha.(\exists\beta.(\beta\to\alpha)\land\beta)\to\alpha$
	I	$((F\alpha\to\beta)\land F\alpha)\to\beta$
$\lambda x.(\lambda y.yy)x$	$\mathcal{W}$	not typable
	P	not typable
	P'	$\forall\alpha.(\exists\beta.(\beta\to\alpha)\land\beta)\to\alpha$
	I	$((F\alpha\to\beta)\land F\alpha)\to\beta$
$(\lambda f.(\lambda x.xx)(\lambda y.fy))(\lambda z.z)$	$\mathcal{W}$	not typable
	P	not typable
	P'	not typable
	I	$\alpha\to\alpha$

Table 1
Some terms and their infered types with algorithm $\mathcal{W}$, Systems **P**, **P'**, and **I**

*1.1.4 System **I***

The term $(\lambda f.(\lambda x.xx)(\lambda y.fy))(\lambda z.z)$ is typable in Kfoury and Wells' recent System **I** [9], which does not have quantifiers, but uses a new technology, called *expansion variables*, to enable terms to be analyzed several times at different types.

After incorporating expansion variables to our system, it appeared that they completely eliminated the need for (positive) universal and (negative) existential quantifiers, when doing full inference. The essential difference between quantifiers and expansion variables is that an expansion variable may appear in several locations, and expansions operate on whole derivations (instantiation operates on a subset of the variables occuring in a derivation).

Also, the presence of quantifiers hides parts of the analysis: the types are more concise, but the fact that some term is used at several types does not clearly appear in derivations. This information can be used in compilers to generate better code by program specialisation [4,10].

Since expansion variables are clearly superior, we decided to remove quantifiers from the system.

1.1.5 System $\boldsymbol{I}_\omega$

The system we present here is a slight variant of System **I**, which we will call System $\mathbf{I}_\omega$. Its technical contribution is relatively small (we assign the type constant ω to unused terms), and we do not prove any of its properties. Our intent is rather to give a simple presentation of a type system and an algorithm exploiting expansion variables, in order to make this technology more accessible.

Our presentation keeps explicit the distinction between positive and negative types, which we found very useful while developping System **P'**. Principal typings in System $\mathbf{I}_\omega$ also have the property that type variables have two occurences of opposite polarities, which guarantees that we respect the polarity discipline; for instance, we never put negative types in positive positions. When implementing the system, this property enables bindings for type variables to be removed from substitutions after every single use, since the other occurrence of the variable generated the binding. We found that checking that the substitution was empty after typing a term was an efficient way to catch bugs in the implementation. Also, we found it very helpful to use distinct data types for positives and negative types, so that the compiler could statically check that the code respected the polarity discipline.

Note that this discipline may appear to be somewhat arbitrary, because the system can be presented without so much emphasis on polarities. However, we think that assigning intrinsic interpretations to positive and negative types also sheds some light on the inference process. The core of our algorithm uses a slightly modified version of the β-unification developped in [8], named for its ability to characterize β-strong normalization of terms. With the polarity discipline, it is easier to see that the unification algorithm indeed performs β-reduction at the level of types: positive types (capabilities, the type of values manipulated by the program) are propagated and refined using expansion variables to match the requirements expressed by negative types. The inference algorithm thus performs some form of exact flow analysis.

1.2 Connection with Linear logic

A connection can be made with Girard's Linear Logic [2], which also emphasizes on the duality between capabilities and requirements. The identity rule in linear sequent calculus is:

$$\frac{}{\vdash A^{\perp}, A}\ (\texttt{identity})$$

The linear negation, $(\cdot)^{\perp}$, expresses the duality between an action A (input, requirements) and reaction $A^{\perp}$ (output, capabilities).

The linear implication is a defined connective: $(A \multimap B) = (A^{\perp} \parr B)$; it "consumes" a proof of A to produce a proof of B. The consumer for an implication is $(A \multimap B)^{\perp} = (A \otimes B^{\perp})$; it contains a proof of A and a consumer for a proof of B — programmaticaly, $B^{\perp}$ is a continuation.

The type system we present makes use of a similar negation to express that some (positive or negative) type is "consumed", without violating the polarity discipline (i.e. without confusing the interpretations of types).

Note that in our system, an arrow appearing in a positive (respectively negative) position intuitively corresponds to Linear Logic's tensor sum $⅋$ (respectively tensor product $\otimes$).

1.3 Contributions

By experimenting with a system mixing intersection types and universal quantifiers, we noticed that when performing total inference, intersection types with expansion variables seem to completely remove the need for universal quantifiers. They express the same form of polymorphism, but expansion variables are more flexible and powerful.

We underline the importance of the polarity discipline, by explicitly assigning different interpretations to positive and negative types. Checking that this discipline is not violated has proven to be very useful while designing and implementing the system.

We present an intersection type system that respects this discipline, and we are confident that it enjoys the same properties as System **I**'s, though we do not have any proofs yet.

We extend the technology of expansion variables by including ω in expansions, leading to a slightly simpler presentation and typings which better reflect the behaviour of terms.

We provide a simple and relatively easy to implement inference algorithm. Our practical experience is that even a naive implementation (explicitly applying substitution instead of using mutable variables) is reasonably efficient. We have also implemented it using a union-find structure.

1.4 Future Work

Giving different interpretations to positive and negative types, along with the ability to express the fact that some type is "consumed", helped greatly while designing the system. This discipline guides the intuition, and will certainly be useful when extending the system, for example with union types and recursion.

Quantifiers have been left out of the system for the sake of simplicity. However, it may be interesting to re-introduce them in some cases, for example when the types have to be shown to the programmer. While making the analysis cruder, the typings become smaller and more readable.

We have an implementation which uses a different strategy of solving the constraints, and which is able to type some weakly normalizing terms, thanks to the ω expansion which discards constraints which can have no impact on the result. Whether this strategy allows to type all normalizing terms is left for future work.

2 Intersection Types with Expansion Variables and ω

In order to ease the comparison with System **I**, we adopt and adapt most of the definitions and conventions of [9].

2.1 The Type System

Definition 2.1 [λ-Terms]

Let x and y range over λ-`Var`, the set of λ-term variables. We use the set of terms

$$M, N \in \Lambda ::= x \mid \lambda x.M \mid MN \ ,$$

quotiented by α-conversion, and the usual notion of reduction

$$(\lambda x.M)N \to_\beta M[x := N].$$

Definition 2.2 [Variables]

We use binary strings in $\{0,1\}^*$, called *offset labels*, to name (and later rename) variables. Let ϵ denote the empty string. Let s and t range over offsets. If $s, t \in \{0,1\}^*$, we note their concatenation $s \cdot t$.

We define the set `TVar` of *type variables*, the set `EVar` of *expansion variables*, and the set `Var` of *variables*, as well as metavariables over these sets:

$$\begin{aligned} \alpha \in \mathtt{TVar} &= \{\mathtt{a}_i^s \mid i \in \mathbb{N}, s \in \{0,1\}^*\} \\ F \in \mathtt{EVar} &= \{\mathtt{F}_i^s \mid i \in \mathbb{N}, s \in \{0,1\}^*\} \\ v \in \mathtt{Var} &= \mathtt{EVar} \cup \mathtt{TVar} \end{aligned}$$

We also define sets of polarized type variables by annotating variables with a $-$ or $+$ in prefix superscript:

$$\begin{aligned} \mathtt{TVar}^+ &= \{{}^+\alpha \mid \alpha \in \mathtt{TVar}\} \\ \mathtt{TVar}^- &= \{{}^-\alpha \mid \alpha \in \mathtt{TVar}\} \end{aligned}$$

Definition 2.3 [Types] We define the set $\mathcal{P}$ of *positive types*, the set $\mathcal{N}$ of *negative types*, and the set $\mathcal{T}$ of *types*, as well as metavariables over these sets:

$$\begin{aligned} \pi \in \mathcal{P} &::= \omega \mid \bar{\pi} \mid (\pi_0 \wedge \pi_1) \mid (F\, \pi) \\ \bar{\pi} \in \bar{\mathcal{P}} &::= {}^+\alpha \mid (\nu \to \bar{\pi}) \\ \nu \in \mathcal{N} &::= \omega \mid \bar{\nu} \mid (\nu_0 \wedge \nu_1) \mid (F\, \nu) \\ \bar{\nu} \in \bar{\mathcal{N}} &::= {}^-\alpha \mid (\pi \to \bar{\nu}) \\ \tau \in \mathcal{T} &::= \nu \mid \pi \end{aligned}$$

We use the set of types $\mathcal{T}$ only when a definition is uniform on $\mathcal{P}$ and $\mathcal{N}$.

Note that polarity annotations on variables are redundant, because they can always be recovered if the global polarity of a type is known. Their purpose is to introduce a syntactic difference between positive types and negative types, to support our intuition that the interpretation of a type is intrinsic to it.

We use identical symbols for connectives which have dual purposes. Positive types $\mathcal{P}$ are interpreted in terms of capabilities of an output, and negative types $\mathcal{N}$ in terms of requirements on an input.

A positive arrow type $\nu \to \bar{\pi}$ is the type of a functional value, as introduced by a λ-abstraction $(\lambda x.M)$. A negative arrow type $\pi \to \bar{\nu}$ expresses the fact that a functional value is expected/consumed, and is of course introduced by an application (xM). It is interesting to note that each positive (resp. negative) arrow in the typing of a term M corresponds to an abstraction (resp. application) in the normal form of M.

Positive intersections permit analyzing a term several times, in order to give it a stronger type (i.e., with greater capabilities). Negative intersections combine requirements into stronger requirements, and are mainly introduced by an application MN when some variable x occurs free in both M and N.

In positive positions, the type constant ω stands for the greatest capabilities, while in negative positions ω stands for the least requirements. For example, it is syntactically obvious that $\omega \to \pi$ is the type of a function which does not use its argument (either directly or indirectly). The presence of this type constant will allow environments to be defined as total functions, assigning type ω to all variables which do not occur free in a term. This leads to simpler definitions.

The type constant ω can be thought of as the intersection of zero types, so we establish the following equalities on types:

$$\omega \wedge \tau = \tau = \tau \wedge \omega$$

Definition 2.4 [The $\perp$ Relation] We define an asymmetric relation $\perp$ (read "perp") of $\mathcal{N}$ and $\mathcal{P}$ as follows:

$$\omega \perp \omega \qquad\qquad {}^{-}\alpha \perp {}^{+}\alpha$$

$$\frac{\nu_1 \perp \pi_1 \quad \nu_2 \perp \pi_2}{(\nu_1 \wedge \nu_2) \perp (\pi_1 \wedge \pi_2)} \qquad \frac{\nu \perp \pi \quad \bar{\nu} \perp \bar{\pi}}{(\pi \to \bar{\nu}) \perp (\nu \to \bar{\pi})}$$

$$\frac{\nu \perp \pi}{(F\nu) \perp (F\pi)}$$

When we write $\nu \perp \pi$, it means that the requirements ν are fulfilled by the capabilities π.

We define $(\cdot)^{\perp}$ (read "perp") as an operation from $\mathcal{T}$ to $\mathcal{T}$ such that $\tau^{\perp}$ is τ with the polarity of every variable flipped. Note that $\nu^{\perp} \in \mathcal{P}$ and $\pi^{\perp} \in \mathcal{N}$, and that $(\cdot)^{\perp}$ is involutive (i.e. $\tau^{\perp^{\perp}} = \tau$).

$$\frac{\nu \perp \pi}{\{x : \nu\} \vdash x : \pi}\ \texttt{VAR} \qquad \frac{A \vdash M : \pi}{FA \vdash M : F\pi}\ F$$

$$\frac{A \vdash M : \pi}{A \setminus x \vdash \lambda x.M : A(x) \to \pi}\ \texttt{ABS} \qquad \frac{A \vdash M : \pi_0 \quad B \vdash M : \pi_1}{A \wedge B \vdash M : \pi_0 \wedge \pi_1}\ \wedge$$

$$\frac{A \vdash M : \pi_1 \quad B \vdash N : \pi_2 \quad (\pi_2 \to \pi_3^\perp) \perp \pi_1}{A \wedge B \vdash MN : \pi_3}\ \texttt{APP} \qquad \frac{A \vdash M : \pi}{\{\} \vdash M : \omega}\ \omega$$

Table 2
Type inference rules of System $\mathbf{I}_\omega$

The intuitive meaning of $(\cdot)^\perp$ depends on the polarity at which it occurs. In negative position, $\pi^\perp$ indicates that a value of type π is consumed. In positive position, $\nu^\perp$ indicates that the requirements ν are used to produce a value. For example, one of the types of the identity function is $({}^-\alpha \to {}^+\alpha)^\perp \to ({}^-\alpha \to {}^+\alpha)$, meaning that a value of type ${}^-\alpha \to {}^+\alpha$ is consumed, and a value of the same type is produced.

Essentially, the $\perp$ relation and the $(\cdot)^\perp$ operation express the duality between requirements/inputs and capabilities/outputs.

Definition 2.5 [Environment] An environment A is a total function from λ-`Var` to $\mathcal{N}$. We use the following notation:

$$(\{x_i : \nu_i\}_{i\in I})(x_j) = \begin{cases} \nu_j & \text{if } j \in I, \\ \omega & \text{if } j \notin I. \end{cases}$$

We write $A \setminus x$ for the environment that :

$$(A \setminus x)(y) = \begin{cases} \omega & \text{if } x = y, \\ A(y) & \text{if } x \neq y. \end{cases}$$

We write $A_1 \wedge A_2$ for the environment such that:

$$(A_1 \wedge A_2)(x) = A_1(x) \wedge A_2(x)$$

We write FA for the environment such that:

$$(FA)(x) = \begin{cases} \omega & \text{if } A(x) = \omega, \\ F\nu & \text{if } A(x) = \nu \text{ (and } \nu \neq \omega). \end{cases}$$

2.2 Typing Rules

Type inference rules are given in table 2.

Rule `VAR` is very close to the identity rule of Linear Logic. It introduces a value of type π, interpreted as capabilities (the variable is viewed as an output), and a "consumer" of type ν in the environment, which is interpreted as requirements (the variable is viewed as an input); the two types are tied by the $\perp$ relation.

Rule `ABS` introduces a positive arrow, which represents a functional value. Note that since the environment is a total function, no special rule is necessary to handle the case where x is not free in M. This is unlike System $\mathbf{I}$, which uses a fresh type variable instead of ω. Type variables introduced in this fashion have a single occurrence in a principal typing, while System $\mathbf{I}_\omega$ has the property that each type variable corresponds to a flow that can actually occur in the program, from the negative occurrence of the variable to its positive occurrence.

In rule `APP`, the negative arrow on the left of the $(\pi_2 \to \pi_3^\perp) \perp \pi_1$ condition is a "consumer" for a functional value. The type $(\pi_3^\perp)$ to the right of the arrow can be thought of as a continuation which will consume the result of the function, which has type π_3. There are several equivalent ways to write the `APP` rule. We choose the one in table 2 because it makes the negative application explicit, and it is closer to the inference algorithm. A more succinct but equivalent version is:

$$\frac{A \vdash M : \pi_1^\perp \to \pi_2 \quad B \vdash N : \pi_1}{A \wedge B \vdash MN : \pi_2}\ \texttt{APP}$$

Here capabilities π_1 are consumed by the function M: this is made explicit by the occurrence of $(\cdot)^\perp$ in $\pi_1^\perp$.

The rule F opens up the possibility for a term to be analyzed any number of times. It is not needed to derive a type for a term, but it allows the inference algorithm to give stronger types. For example, $(\lambda f.\lambda x.fx) : (F^{+}\alpha \to {}^{-}\beta) \to F^{-}\alpha \to {}^{+}\beta$ is stronger than $(\lambda f.\lambda x.fx) : ({}^{+}\alpha \to \beta) \to {}^{-}\alpha \to {}^{+}\beta$, because the first one can be applied to a function which uses its argument at different types, while the second one cannot.

The rule $\wedge$ combines different analysis of the same term, yielding a stronger type for it. The inference algorithm transforms occurences of the F rules (which express a potentiality) into $\wedge$, as analysis proceeds. This transformation is called an expansion.

The rule ω allows a typable term to be assigned type ω. This rule needs to be used when applying a function $(\lambda x.M)$ whose argument has type ω (for example, when x does not occur free in M). In fact, running the inference algorithm on some term performs some kind of dead code analysis: all occurrences of the ω rule indicate that the term is never used. We do not allow type ω for any term because it would break the correspondence with the inference algorithm.

2.3 Substitution

We now introduce the necessary definitions used by the inference algorithm.

Definition 2.6 [Expansion] We define the set $\mathcal{E}$ of *expansions*, as well as a metavariable over this set:

$$e \in \mathcal{E} ::= \omega \mid \Box \mid e_0 \wedge e_1 \mid F\, e$$

Expansions are the actualization of the potentiality expressed by expansion variables. It indicates how many times a term is analyzed: 0 (ω), 1 ($\Box$), $m+n$ ($e_0 \wedge e_1$, where e_0 indicates that the term is analyzed m times, and e_1 indicates that the term is analyzed n times), or that it still has the potentiality to further be refined ($F\, e$).

Definition 2.7 [Substitution] A *substitution* is a total function $\mathbf{S} : (\mathtt{TVar} \cup \mathtt{EVar}) \to (\mathcal{T} \cup \mathcal{E})$ which respects "sorts", i.e., $\mathbf{S}({}^-\alpha) \in \mathcal{N}$ for every ${}^-\alpha \in \mathtt{TVar}^-$, $\mathbf{S}({}^+\alpha) \in \mathcal{P}$ for every ${}^+\alpha \in \mathtt{TVar}^+$, and $\mathbf{S}(F) \in \mathcal{E}$ for every $F \in \mathtt{EVar}$. The notation $\mathbf{S} = \{\!\!\{\alpha_i \mapsto \langle \nu_i, \pi_i \rangle, F_j \mapsto e_j\}\!\!\}_{i \in I, j \in J}$ denotes the substitution such that:

$$\begin{aligned}
\mathbf{S}({}^-\alpha_k) &= \nu_k && \text{if } k \in I,\\
\mathbf{S}({}^+\alpha_k) &= \pi_k && \text{if } k \in I,\\
\mathbf{S}({}^-\alpha_k) &= {}^-\alpha_k && \text{if } k \notin I,\\
\mathbf{S}({}^+\alpha_k) &= {}^+\alpha_k && \text{if } k \notin I,\\
\mathbf{S}(F_k) &= e_k && \text{if } k \in J,\\
\mathbf{S}(F_k) &= F_k \Box && \text{if } k \notin J.
\end{aligned}$$

Definition 2.8 [Variable Renaming] For every $t \in \{0,1\}^*$, we define a variable renaming $\langle \cdot \rangle^t$ from $\mathcal{T} \cup \mathcal{E}$ to $\mathcal{T} \cup \mathcal{E}$, by induction:

$$\begin{array}{rclcrcl}
\langle \omega \rangle^t &=& \omega & & & & \\
\langle {}^-\mathtt{a}_i^s \rangle^t &=& {}^-\mathtt{a}_i^{s \cdot t} & & \langle \Box \rangle^t &=& \Box \\
\langle {}^+\mathtt{a}_i^s \rangle^t &=& {}^+\mathtt{a}_i^{s \cdot t} & & & & \\
\langle \tau \to \tau' \rangle^t &=& \langle \tau \rangle^t \to \langle \tau' \rangle^t & & & & \\
\langle \tau_0 \wedge \tau_1 \rangle^t &=& \langle \tau_0 \rangle^t \wedge \langle \tau_1 \rangle^t & & \langle e_0 \wedge e_1 \rangle^t &=& \langle e_0 \rangle^t \wedge \langle e_1 \rangle^t \\
\langle \mathtt{F}_i^s\, \tau \rangle^t &=& \mathtt{F}_i^{s \cdot t}\, \langle \tau \rangle^t & & \langle \mathtt{F}_i^s\, e \rangle^t &=& \mathtt{F}_i^{s \cdot t}\, \langle e \rangle^t
\end{array}$$

In words, $\langle \tau \rangle^t$ is obtained from τ by appending t to the offset label of every variable in τ. Note that renaming respects sorts and polarities, i.e. if $\pi \in \mathcal{P}$ then $\langle \pi \rangle^t \in \mathcal{P}$, if $\nu \in \mathcal{N}$ then $\langle \nu \rangle^t \in \mathcal{N}$, and if $e \in \mathcal{E}$ then $\langle e \rangle^t \in \mathcal{E}$.

Definition 2.9 [Substitution on Types and Expansions] A substitution $\mathbf{S}$ is extended to a function $\bar{\mathbf{S}}$ from $\mathcal{E} \times (\mathcal{T} \cup \mathcal{E})$ to $(\mathcal{T} \cup \mathcal{E})$ which respects sorts, in

$$\dfrac{\dfrac{\dfrac{\pi^{\perp} \perp \pi}{\{x : \pi^{\perp}\} \vdash x : \pi}\,\mathtt{VAR}}{\{\} \vdash \lambda x.x : \pi^{\perp} \to \pi}\,\mathtt{ABS} \qquad \dfrac{\dfrac{\dfrac{F^{+}\alpha \to {}^{-}\beta \perp F^{-}\alpha \to {}^{+}\beta}{\{y : F^{+}\alpha \to {}^{-}\beta\} \vdash y : F^{-}\alpha \to {}^{+}\beta}\,\mathtt{VAR} \quad \dfrac{\dfrac{{}^{-}\alpha \perp {}^{+}\alpha}{\{y : {}^{-}\alpha\} \vdash y : {}^{+}\alpha}\,\mathtt{VAR}}{\{y : F^{-}\alpha\} \vdash y : F^{+}\alpha}\,\mathtt{F}}{\{y : (F^{+}\alpha \to {}^{-}\beta) \wedge F^{-}\alpha\} \vdash yy : {}^{+}\beta}\,\mathtt{APP}}{\{\} \vdash \lambda y.yy : \pi}\,\mathtt{ABS}}{\{\} \vdash (\lambda x.x)(\lambda y.yy) : \pi}\,\mathtt{APP}$$

where $\pi = ((F^{+}\alpha \to {}^{-}\beta) \wedge F^{-}\alpha) \to {}^{+}\beta$

Table 3
Principal typing of $(\lambda x.x)(\lambda y.yy)$

the following way:

$$\begin{aligned}
\bar{\mathbf{S}}(\square, \omega) &= \omega & & \\
\bar{\mathbf{S}}(\square, {}^{-}\alpha) &= \mathbf{S}({}^{-}\alpha) & \bar{\mathbf{S}}(\square, \square) &= \square \\
\bar{\mathbf{S}}(\square, {}^{+}\alpha) &= \mathbf{S}({}^{+}\alpha) & & \\
\bar{\mathbf{S}}(\square, F\tau) &= \bar{\mathbf{S}}(\mathbf{S}(F), \tau) & \bar{\mathbf{S}}(\square, Fe) &= \bar{\mathbf{S}}(\mathbf{S}(F), e) \\
\bar{\mathbf{S}}(\square, \tau \to \tau') &= \bar{\mathbf{S}}(\square, \tau) \to \bar{\mathbf{S}}(\square, \tau') & & \\
\bar{\mathbf{S}}(\square, \tau_0 \wedge \tau_1) &= \bar{\mathbf{S}}(\square, \tau_0) \wedge \bar{\mathbf{S}}(\square, \tau_1) & \bar{\mathbf{S}}(\square, e_0 \wedge e_1) &= \bar{\mathbf{S}}(\square, e_0) \wedge \bar{\mathbf{S}}(\square, e_1) \\
\bar{\mathbf{S}}(\omega, \tau) &= \omega & \bar{\mathbf{S}}(\omega, e) &= \omega \\
\bar{\mathbf{S}}(e_0 \wedge e_1, \tau) &= \bar{\mathbf{S}}(e_0, \langle \tau \rangle^0) \wedge \bar{\mathbf{S}}(e_1, \langle \tau \rangle^1) & \bar{\mathbf{S}}(e_0 \wedge e_1, e') &= \bar{\mathbf{S}}(e_0, \langle e' \rangle^0) \wedge \bar{\mathbf{S}}(e_1, \langle e' \rangle^1) \\
\bar{\mathbf{S}}(F\,e, \tau) &= F\,\bar{\mathbf{S}}(e, \tau) & \bar{\mathbf{S}}(F\,e, e') &= F\,\bar{\mathbf{S}}(e, e')
\end{aligned}$$

Applying expansions and substitutions can be defined as distinct operations, but it is clearer and more convenient to apply them simultaneously.

Definition 2.10 [Substitution on Environments] A substitution is lifted to operate on environments, in the following way:

$$(\bar{\mathbf{S}}(e, A))(x) = \bar{\mathbf{S}}(e, A(x)).$$

2.4 Examples

Depicted in tables 3, 4 and 5 are examples of derivations obtained by the inference algorithm exposed in section 3 for some interesting terms.

Table 3 presents the derivation for the fourth example in table 1.

Table 4 exposes the dynamics of the system. The substitution computed for the last use of the `APP` rule is

$$\{\!\![F_1 \mapsto \square, F_2 \mapsto \square \wedge \square, \alpha \mapsto \langle \pi_2^{\perp}, \pi_2 \rangle, \beta \mapsto \langle \pi_2^{\perp}, \pi_2 \rangle]\!\!\}$$

The expansion variable F_2 has been instantiated to $\square \wedge \square$ to allow analyzing the type of $(\lambda x.x)$ twice, as $(\lambda y.yy)(\lambda x.x)$ reduces to $(\lambda x.x)(\lambda x.x)$. The type π_2 of the right subterm (which takes the place of α) is still allowed to be refined any number of times thanks to F_1. The left subterm $(\lambda x.x)$ uses its argument once, hence $F_1 \mapsto \square$.

Table 5 shows the result of an ω-expansion. For comparison, System **I** infers the typing $z : \alpha_0 \wedge \alpha_6^1 \vdash (\lambda x.xx)(\lambda y.z) : \alpha_0$.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{F_1\,{}^{+}\alpha \to {}^{-}\beta \perp F_1\,{}^{-}\alpha \to {}^{+}\beta}{\{y : F_1\,{}^{+}\alpha \to {}^{-}\beta\} \vdash y : F_1\,{}^{-}\alpha \to {}^{+}\beta}\,\texttt{VAR}
\qquad
\dfrac{\dfrac{{}^{-}\alpha \perp {}^{+}\alpha}{\{y : {}^{-}\alpha\} \vdash y : {}^{+}\alpha}\,\texttt{VAR}}{\{y : F_1\,{}^{-}\alpha\} \vdash y : F_1\,{}^{+}\alpha}\,\texttt{F}
}{\{y : (F_1\,{}^{+}\alpha \to {}^{-}\beta) \wedge F_1\,{}^{-}\alpha\} \vdash yy : {}^{+}\beta}\,\texttt{APP}
}{\{\} \vdash \lambda y.yy : ((F_1\,{}^{+}\alpha \to {}^{-}\beta) \wedge F_1\,{}^{-}\alpha) \to {}^{+}\beta}\,\texttt{ABS}
\qquad
\dfrac{\dfrac{\dfrac{{}^{-}\gamma \perp {}^{+}\gamma}{\{x : {}^{-}\gamma\} \vdash x : {}^{+}\gamma}\,\texttt{VAR}}{\{\} \vdash \lambda x.x : \pi_2}\,\texttt{ABS}}{\{\} \vdash \lambda x.x : F_2({}^{-}\gamma \to {}^{+}\gamma)}\,\texttt{F}
}{?}
$$

$$\Downarrow$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{(\pi_2^{\perp} \to \pi_2)^{\perp} \perp \pi_2^{\perp} \to \pi_2}{\{y : (\pi_2^{\perp} \to \pi_2)^{\perp}\} \vdash y : \pi_2^{\perp} \to \pi_2}\,\texttt{VAR}
\qquad
\dfrac{\pi_2^{\perp} \perp \pi_2}{\{y : \pi_2^{\perp}\} \vdash y : \pi_2}\,\texttt{VAR}
}{\{y : (\pi_2^{\perp} \to \pi_2)^{\perp} \wedge \pi_2^{\perp}\} \vdash yy : \pi_2}\,\texttt{APP}
}{\{\} \vdash \lambda y.yy : (\pi_2^{\perp} \to \pi_2)^{\perp} \wedge \pi_2^{\perp} \to \pi_2}\,\texttt{ABS}
\qquad
\dfrac{
\dfrac{\dfrac{\pi_2^{\perp} \perp \pi_2}{\{x : \pi_2^{\perp}\} \vdash x : \pi_2}\,\texttt{VAR}}{\{\} \vdash \lambda x.x : \pi_2^{\perp} \to \pi_2}\,\texttt{ABS}
\qquad
\dfrac{\dfrac{{}^{-}\gamma^1 \perp {}^{+}\gamma^1}{\{x : {}^{-}\gamma^1\} \vdash x : {}^{+}\gamma^1}\,\texttt{VAR}}{\{\} \vdash \lambda x.x : \pi_2}\,\texttt{ABS}
}{\{\} \vdash \lambda x.x : (\pi_2^{\perp} \to \pi_2) \wedge \pi_2}\,\wedge
}{\{\} \vdash (\lambda y.yy)(\lambda x.x) : \pi_2}\,\texttt{APP}
$$

where $\pi_2 = {}^{-}\gamma^1 \to {}^{+}\gamma^1$.

Table 4
Principal typing of $(\lambda y.yy)(\lambda x.x)$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{(\omega \to {}^{-}\alpha_1) \perp (\omega \to {}^{+}\alpha_1)}{\{x : \omega \to {}^{-}\alpha_1\} \vdash x : \omega \to {}^{+}\alpha_1}\,\texttt{VAR}
\qquad
\dfrac{\dfrac{{}^{-}\alpha_2 \perp {}^{+}\alpha_2}{\{x : {}^{-}\alpha_2\} \vdash x : {}^{+}\alpha_2}\,\texttt{VAR}}{\{\} \vdash x : \omega}\,\omega
}{\{x : \omega \to {}^{-}\alpha_1\} \vdash xx : {}^{+}\alpha_1}\,\texttt{APP}
}{\{\} \vdash \lambda x.xx : (\omega \to {}^{-}\alpha_1) \to {}^{+}\alpha_1}\,\texttt{ABS}
\qquad
\dfrac{
\dfrac{\dfrac{{}^{-}\alpha_1 \perp {}^{+}\alpha_1}{\{z : {}^{-}\alpha_1\} \vdash z : {}^{+}\alpha_1}\,\texttt{VAR}}{\{z : {}^{-}\alpha_1\} \vdash \lambda y.z : \omega \to {}^{+}\alpha_1}\,\texttt{ABS}
\qquad
\dfrac{\vdots}{\{\} \vdash \lambda y.z : \omega}\,\omega
}{\{z : {}^{-}\alpha_1\} \vdash \lambda y.z : \omega \to {}^{+}\alpha_1}\,\wedge
}{\{z : {}^{-}\alpha_1\} \vdash (\lambda x.xx)(\lambda y.z) : {}^{+}\alpha_1}\,\texttt{APP}
$$

Table 5
Principal typing of $(\lambda x.xx)(\lambda y.z)$

3 The Inference Algorithm

The inference algorithm is given in table 6. It takes a term M as input, and gives a pair $\langle A, \pi\rangle$ of an environment A and a positive type π such that $A \vdash M : \pi$ is a typing of M, which we believe is principal (though we do not have a proof yet).

Definition 3.1 [Ground Composition of Substitutions] Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be two substitutions with disjoint domains. We write their ground composition $\mathbf{S}_2 \diamond \mathbf{S}_1$ defined as follows:

$$\mathbf{S}_2 \diamond \mathbf{S}_1 = \{\!\!\{ v \mapsto \bar{\mathbf{S}}_2(\Box, \mathbf{S}_1(v)) \mid v \in \texttt{Var} \}\!\!\}$$

Definition 3.2 [Most General Solution] We write $\texttt{MGS}(\nu \perp \pi)$ for a substitution $\mathbf{S}$ such that $\bar{\mathbf{S}}(\nu) \perp \bar{\mathbf{S}}(\pi)$. In order to make the algorithm more readable, we define an operator ";" as follows:

$$\mathbf{S}\ ;\ (\nu \perp \pi) = \texttt{MGS}(\bar{\mathbf{S}}(\Box, \nu) \perp \bar{\mathbf{S}}(\Box, \pi)) \diamond \mathbf{S}$$

The substitution $\texttt{MGS}(\nu \perp \pi)$ is computed as shown in table 7, choosing rules by first matched.

$$
\begin{aligned}
&\texttt{PP}(x) = \\
&\quad \text{let } \alpha \text{ be fresh} \\
&\quad \text{in } \langle \{x : {}^-\alpha\}, {}^+\alpha \rangle \\
&\texttt{PP}(\lambda x.M) = \\
&\quad \text{let } \langle A, \pi \rangle = \texttt{PP}(M) \\
&\quad \text{in } \langle A \setminus x, A(x) \to \pi \rangle \\
&\texttt{PP}(MN) = \\
&\quad \text{let } \langle A, \pi_1 \rangle = \texttt{PP}(M) \\
&\qquad\;\; \langle B, \pi_2 \rangle = \texttt{PP}(N) \\
&\qquad\;\; \alpha \text{ and } F \text{ be fresh} \\
&\qquad\;\; \mathbf{S} = \texttt{MGS}((F\pi_2 \to {}^-\alpha) \perp \pi_1) \\
&\quad \text{in } \langle \bar{\mathbf{S}}(\Box, A \wedge FB), \bar{\mathbf{S}}(\Box, {}^+\alpha) \rangle
\end{aligned}
$$

Table 6
The inference algorithm computing Principal Pairs

$$
\begin{aligned}
\texttt{MGS}(\bar{\nu} \perp {}^+\alpha) &= \{\!\![\alpha \mapsto \langle \bar{\nu}, \bar{\nu}^{\perp} \rangle]\!\!\} \\
\texttt{MGS}({}^-\alpha \perp \bar{\pi}) &= \{\!\![\alpha \mapsto \langle \bar{\pi}^{\perp}, \bar{\pi} \rangle]\!\!\} \\
\texttt{MGS}(\omega \perp F\pi) &= \{\!\![F \mapsto \omega]\!\!\} \\
\texttt{MGS}(\bar{\nu} \perp F\pi) &= \{\!\![F \mapsto \Box]\!\!\} \;;\; (\bar{\nu} \perp \pi) \\
\texttt{MGS}((\nu_0 \wedge \nu_1) \perp F\pi) &= \{\!\![F \mapsto F^0\Box \wedge F^1\Box]\!\!\} \;; \\
&\qquad (\nu_0 \perp \langle F\pi \rangle^0) \;;\; (\nu_1 \perp \langle F\pi \rangle^1) \\
\texttt{MGS}(F\nu \perp F\pi) &= \texttt{MGS}(\nu \perp \pi) \\
\texttt{MGS}(G\nu \perp F\pi) &= \{\!\![F \mapsto GH\Box]\!\!\} \;;\; (\nu \perp H\pi) \quad \text{with } H \text{ fresh} \\
\texttt{MGS}((\pi_2 \to \nu_2) \perp (\nu_1 \to \pi_1)) &= \texttt{MGS}(\nu_1 \perp \pi_2) \;;\; (\nu_2 \perp \pi_1)
\end{aligned}
$$

Table 7
Computing the most general solution to a $\perp$-problem

This algorithm is a variant of β-unification, developped in [8].

Proposition 3.3 *In any problem $\nu \perp \pi$ generated in $\texttt{PP}(MN)$, any expansion variable has exactly one positive occurrence, and may have an arbitrary number of negative occurrences.*

The intuition behind this remark is that a term of the λ-calculus has an arbitrary number of inputs (its free and abstracted variables), but a single output. During β-unification, the positive occurence of an expansion variable may be matched against either an intersection, a simple type $\bar{\nu}$, or ω. Applying the resulting expansion will effectively duplicate the term $(e_0 \wedge e_1)$, actualize the fact that it is used only once ($\Box$), or discard the term (ω).

As a side note, it may be worth noting the similarity with the rules of Linear Logic exponentials, contraction (duplicate), dereliction (use only once), weakening (discard), and "of course" (use any number of times). An essential difference is that these rules manipulate formulas, while expansions manipulate proofs.

Proposition 3.4 *Principal typings returned by algorithm* `PP` *have the property that every type variable which occurs in the typing has exactly two occurences, of opposite polarities.*

The case for x obviously has this property. The case for $(\lambda x.M)$ does not affect the number of occurences of variables, and does not affect polarities (it does not mention $(\cdot)^{\perp}$). In the case for (MN), by the induction hypothesis, type variables which have a single occurence in π_1 (resp. π_2) have their only other occurence in A (resp. B). Thus, every type variable occuring in the problem $((F\pi_2 \to \alpha^-) \perp \pi_2)$ has its other occurence in $\langle A \wedge FB, \alpha^+\rangle$. Applying the most general solution $\mathbf{S}$ of the $\perp$ problem preserves the invariant (we do not prove it here), hence the stated result.

The fresh type variable α plays the role of an identity continuation for the application: its negative occurence ${}^-\alpha$ can be seen as a sink which will receive the type of the result of the function, and its positive occurence ${}^+\alpha$ as a source which will output it unchanged as the result of the application.

Note that in an implementation interested only in principal typings of terms, the $(\cdot)^{\perp}$ operation is never needed. It expresses the fact that a some capabilities or requirements have been used up, but the inference algorithm only manipulates actual capabilities and requirements. This can be seen by looking at the first two rules in table 7, and considering that variables have only two occurrences of opposite polarities: each type variable occurring once in the $\perp$ problem may generate a binding in the most general solution to this problem, and its other occurrence is then either the fresh variable α taken in the `APP` rule, or some type variable occurring in the environment $A \wedge FB$. Consequently, the part in the range of the substitution which is under a $(\cdot)^{\perp}$ is only useful when applying substitutions to the original problems — which the algorithm never does. Types under a $(\cdot)^{\perp}$ are seen when displaying whole derivations.

4 Comparison with System I

The system we present is very close to System **I**, and we expect it to share most of its properties. However, we believe that our presentation is clearer, and has many small improvements over System **I**:

- Defining environments as total functions allows for a single `ABS` rule instead of two as in System **I**.
- Our principal typings [11] have the property that every type variable has exactly one negative occurrence and one positive occurrence, and corresponds

to a flow that can actually occur in the program.

- Including ω in expansions is new. Thus, if the analysis determines that a particular value will not be used, this information can be propagated to the type environment information for the free variables referenced by the value. E.g., in the term

$$(\lambda x.y)(\lambda z.abcde)$$

we can infer the type ω in the type environments for the variables a, b, c, d, and e.

- Our definition of lifting substitutions is much simpler than the one in [9].

5 Acknowledgments

Thanks to Yohann Fabre for arising my interest for type theory. Thanks also to Akim Demaille, for useful discussions and introducing me to Linear Logic, and to Julien Dufour, for his constant interest and support in my work. I am indebted to Assaf Kfoury and Joe Wells, whose contributions were invaluable, and without whom this paper would never have been written.

References

[1] Damas, L. and R. Milner, *Principle Type Inference for Functional Programs (extended abstract)*, in: *9th ACM Symposium on Principle of Programming Langugages*, 1982, pp. 207–212.

[2] Girard, J.-Y., *Linear logic: its syntax and semantics*, in: J.-Y. Girard, Y. Lafont and L. Regnier, editors, *Advances in Linear Logic* (1995), pp. 1–42.
URL `http://citeseer.nj.nec.com/girard95linear.html`

[3] Harper, R., R. Milner and M. Tofte, "The Definition of Standard ML," MIT-press, 1990.

[4] Hughes, J., *An Introduction to Program Specialisation by Type Inference*, in: *Functional Programming* (1996), published electronically.
URL
`http://www.dcs.gla.ac.uk/fp/workshops/fpw96/Proceedings96.html`

[5] Jim, T., *What are principal typings and what are they good for?*, Tech. memo. MIT/LCS/TM-532, MIT (1995).
URL `http://www.research.att.com/~trevor/papers.html`

[6] Jim, T., *A polar type system*, Workshop on Intersection Types and Related Systems (2000).
URL `http://www.cee.hw.ac.uk/~jbw/itrs/itrs00/program.html`

[7] Jones, S. L. P., C. V. Hall, K. Hammond, W. Partain and P. Wadler, *The glasgow haskell compiler: a technical overview*, in: *Proc. UK Joint Framework*

for Information Technology (JFIT) Technical Conference, 93.
URL `citeseer.nj.nec.com/jones92glasgow.html`

[8] Kfoury, A. J., *Beta-reduction as unification*, in: D. Niwinski, editor, *Logic, Algebra, and Computer Science (H. Rasiowa Memorial Conference, December 1996), Banach Center Publication, Volume 46*, Springer-Verlag, 1999 pp. 137–158.
URL `http://types.bu.edu/reports/Kfoury:Rasiowa-memorial.html`

[9] Kfoury, A. J. and J. B. Wells, *Principality and decidable type inference for finite-rank intersection types*, in: *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, 1999, pp. 161–174.
URL `http://types.bu.edu/reports/Kfo+Wel:POPL-1999.html`

[10] Thiemann, P., *Enforcing safety properties by program specialization (extended abstract)*.
URL `citeseer.nj.nec.com/thiemann00enforcing.html`

[11] Wells, J. B., *The essence of principal typings*, in: *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, LNCS (2002).
URL `http://types.bu.edu/reports/Wells:ICALP-2002.html`