Workshop on Intersection Types and Related Systems ITRS 2004
Turku, Finland, July 12, 2004.

Intersection and union types in the $\bar{\lambda}\mu\tilde{\mu}$-calculus

Daniel J. Dougherty [1]

Worcester Polytechnic Institute
Worcester MA 01609 USA

Silvia Ghilezan [2]

Faculty of Engineering, University of Novi Sad
Novi Sad, Serbia

Pierre Lescanne [3]

LIP, École Normale Supérieure de Lyon
Lyon, France

Abstract

The original $\bar{\lambda}\mu\tilde{\mu}$ of Curien and Herbelin has a system of simple types, based on sequent calculus, embodying a Curry-Howard correspondence with classical logic. We introduce and discuss three type assignment systems that are extensions of $\bar{\lambda}\mu\tilde{\mu}$ with intersection and union types. The intrinsic symmetry in the $\bar{\lambda}\mu\tilde{\mu}$ calculus leads to an essential use of both intersection and union types.

Key words: Intersection types, union types, $\bar{\lambda}\mu\tilde{\mu}$-calculus, classical logic, Curry-Howard correspondence.

1 Introduction

Intersection types were introduced into the lambda calculus in the late 1970s by Coppo and Dezani [6,7], Pottinger [20] and Sallé [23]. Intersection type assignment systems were devised in order to type more lambda terms than the basic functional, or simply typed, system; indeed, these intersection types systems can

[1] Email: dd@cs.wpi.edu
[2] Email: gsilvia@uns.ns.ac.yu
[3] Email: Pierre.Lescanne@ens-lyon.fr

characterize exactly all strongly normalizing lambda terms. In addition, these systems are suitable for analyzing λ-models and various normalisation properties of λ-terms. A summary of the early research was given by van Bakel [1].

Barbanera et al. [2] added union types to intersection type systems. This work was motivated by the observation that union types arise naturally in denotational semantics and that they can generate more informative types for some terms. However one cannot type with union types more terms than with intersection types only. That is, the system with intersection and union types exactly characterize all strongly normalizing terms as well.

In the 1980's and early 1990's Reynolds explored the role that intersection types can play in a practical programming language (see for example the report [21] on the language Forsythe). Pierce [19] explored the use of union types in programming, for example as a generalization of variant records. More recently Buneman and Pierce [4], have shown how union types can play a key role in the design of query languages for semistructured data union types.

Under the Curry-Howard correspondence formulae provable in intuitionistic logic coincide with types inhabited in simply typed lambda calculus. Griffin extended the Curry-Howard correspondence to classical logic in his seminal 1990 POPL paper [14], by observing that classical tautologies suggest typings for certain control operators. This initiated an active line of research; in particular the $\lambda\mu$ calculus of Parigot [18] embodies a Curry-Howard correspondence for classical logic based on natural deduction.

Meanwhile Curien and Herbelin [8], building on earlier work in [15], defined the system $\overline{\lambda}\mu\tilde{\mu}$. In contrast to Parigot's $\lambda\mu$-calculus, which bases its type system on a natural deduction system for classical logic, terms in $\overline{\lambda}\mu\tilde{\mu}$ represent derivations in a *sequent calculus* proof system and reduction reflects the process of cut-elimination.

In this paper we recount our experience in deriving a type system for $\overline{\lambda}\mu\tilde{\mu}$ which characterizes the strongly normalizing terms. We are naturally led to enrich the Curien-Herbelin system of simple types by introducing intersection types. But it turns out [10] that union types are also necessary in order to completely characterize all strongly normalizing untyped $\overline{\lambda}\mu\tilde{\mu}$-terms (in contrast to the situation in standard λ-calculus).

Remarkably, Laurent [16] has recently and independently set out to analyze the denotational semantics of the $\lambda\mu$ calculus by defining a type system quite similar to ours: in particular his system involves both intersection and union types.

The system we present also enjoys the Subject Reduction property, which typically fails in the presence of union types. Pierce [19] highlighted the failure of Subject Reduction in the presence of union types and [2] showed how to recover this property by suitably restricting the notion of reduction. Wells et. al. [24] explore a λ-calculus which serves as the foundation for a typed intermediate language for optimizing compilers; they use a novel formulation of intersection and union types and *flow types* to encode control information. This system obeys Subject Reduction, and it will be interesting to understand better the relationship between our

system and theirs.

In two recent papers [11,12] Dunfield and Pfenning investigate a type system incorporating–among others–union types. Their language is specifically a call-by-value language, and their type system and type assignment algorithms exploit this aspect in interesting ways. In particular their system satisfies a version of Subject Reduction: they isolate a notion of "definite substitution" which seems to be related to our "definite" types below. The precise relationship between these systems is an area for future investigation. (We are indebted to one of the referees for bringing this work to our attention.)

The paper is organised as follows. Section 2 deals with the untyped syntax of $\overline{\lambda}\mu\widetilde{\mu}$. In Section 3 we discuss sequent calculi which correspond to types of $\overline{\lambda}\mu\widetilde{\mu}$. In Section 4 three type assignment systems are introduced which are extensions of $\overline{\lambda}\mu\widetilde{\mu}$ with intersection and union types. Their properties are discussed in Section 5.

2 The syntax of GEMINI

In this section we present the language GEMINI which is the untyped version of Curien and Herbelin's $\overline{\lambda}\mu\widetilde{\mu}$ calculus introduced in [8]. We identify three syntactic categories: *callers*, *callees*, and *capsules* (in [8] these are referred to respectively as *terms*, *contexts* and *commands*). Letting r, e, and c ranging over *callers*, *callees*, and *capsules*, respectively, we have

$$r ::= x \mid \lambda x.r \mid \mu\alpha.c$$
$$e ::= \alpha \mid r \bullet e \mid \widetilde{\mu}x.c$$
$$c ::= \langle r \parallel e \rangle$$

Callers, *callees*, and *capsules* are together referred to as G-terms. There are two kinds of variables in this system:

(i) the set Var_r of *caller variables* denoted by Latin variables x, y,... which represent inputs, in particular they are bound by λ-abstractions or $\widetilde{\mu}$-abstractions,

(ii) the set Var_e of *callee variables* denoted by Greek variables α,β,... which represent continuations and which can be bound by μ-abstractions. In $\lambda\mu$, they are called μ-variables.

The core of GEMINI is made of capsules $\langle caller \parallel callee \rangle$ where *caller* and *callee* are two components. The *caller* performs basically one of two actions, either it gets data from the other entity, the *callee*, or it asks the callee to take the place of one of its internal callee variables A *callee* can ask a caller to take the place of one of its specified internal caller variables.

GEMINI has three reduction rules which make this interpretation more precise.

$$(\lambda)\ \langle \lambda x.r \parallel r' \bullet e\rangle \longrightarrow \langle r' \parallel \widetilde{\mu} x.\langle r \parallel e\rangle\rangle$$
$$(\mu)\ \quad \langle \mu\alpha.c \parallel e\rangle \longrightarrow c[\alpha \leftarrow e]$$
$$(\widetilde{\mu})\ \quad \langle r \parallel \widetilde{\mu} x.c\rangle \longrightarrow c[x \leftarrow r]$$

Note that on a term of the form $\langle \mu\alpha.c \parallel \widetilde{\mu} x.c\rangle$ rules (μ) and $(\widetilde{\mu})$ can be applied ambiguously. This will be discussed in Subsection 2.1. If one gives priority to (μ) over $(\widetilde{\mu})$ then it is *call-by-value*, otherwise it is *call-by-name*.

Also note that the usual operation of β-reduction is readily effected in the calculus, with a λ-step followed immediately by a $\widetilde{\mu}$-step. This will be discussed in Subsection 2.2.

Of course the substitutions above are defined so as to avoid variable-capture. In this paper, we use the "Barendregt convention" on variables [3]. It says that in a statement or an expression, there is no subexpression in which a variable is both free and bound. The symbols λ, μ, and $\widetilde{\mu}$ all bind variables in the obvious way. The formal definitions of free and bound variables are as expected. For every G-term r, e and c we define two sets of *free variables*, namely $\mathtt{Fv_r}(r)$, $\mathtt{Fv_e}(r)$, $\mathtt{Fv_r}(e)$, $\mathtt{Fv_e}(e)$, $\mathtt{Fv_r}(c)$ and $\mathtt{Fv_e}(c)$. For instance

$$\mathtt{Fv_r}(\langle x \parallel \mu\alpha.\langle y \parallel \alpha\rangle \bullet \beta\rangle) = \{x, y\}$$
$$\mathtt{Fv_e}(\langle x \parallel \mu\alpha.\langle y \parallel \alpha\rangle \bullet \beta\rangle) = \{\beta\}$$

From the reduction rules, one deduces easily that the *normal forms* of G-terms are generated by the following abstract syntax.

$$r_{nf} ::= x \mid \lambda x.r_{nf} \mid \mu\alpha.c_{nf}$$
$$e_{nf} ::= \alpha \mid r_{nf} \bullet e_{nf} \mid \widetilde{\mu} x.c_{nf}$$
$$c_{nf} ::= \langle x \parallel \alpha\rangle \mid \langle x \parallel r_{nf} \bullet e_{nf}\rangle \mid \langle \lambda x.r_{nf} \parallel \alpha\rangle$$

2.1 *Failure of confluence*

As a rewriting calculus GEMINI (even in its typed version $\bar{\lambda}\mu\widetilde{\mu}$) has an essential critical pair, between the μ and the $\widetilde{\mu}$ redexes. Indeed the calculus is inherently non confluent. This reflects the non confluence of cut elimination in classical sequent calculus. As a simple example observe that the capsule

$$\langle \mu\alpha.\langle z_1 \parallel \beta_1\rangle \parallel \widetilde{\mu} x.\langle z_2 \parallel \beta_2\rangle\rangle$$

reduces to each of $\langle z_1 \parallel \beta_1\rangle$ and $\langle z_2 \parallel \beta_2\rangle$.

This is more than simply a reflection of the well-known fact that the equational theories of call-by-name and call-by-value differ. It is a reflection of the great expressive power of the language: a single term containing several capsules can encompass several complete computational processes, and the μ and $\widetilde{\mu}$ reductions allow free transfer of control between them.

So the combinatorics of pure reduction is very complex. In this light it is perhaps slightly surprising that the strongly normalizing computations can so readily be characterized, via the type system we present later.

When reduction in GEMINI is constrained to commit to the call-by-name discipline or to the call-by-value, the system is confluent.

2.2 Encoding of λ-calculus

It is not hard to see that GEMINI is Turing-complete as a programming language, since the untyped λ-calculus can be coded easily into it. Rather than give a formal development, we simply show how to encode some familiar lambda terms in the calculus.

Notation. If m and n are callers, let $m * n$ denote the caller term $\mu\alpha.\langle m \parallel n \bullet \alpha\rangle$. (Of course α is not free in m or n here.)

Example 2.1 (Classical beta-reduction)

$$\begin{array}{rcl}(\lambda x.r) * s & \equiv & \mu\alpha.\langle \lambda x.r \parallel s \bullet \alpha\rangle \\ & \twoheadrightarrow & \mu\alpha.\langle r[x \leftarrow s] \parallel \alpha\rangle\end{array}$$

Example 2.2 (Eta reduction) Notice that

$$\begin{array}{rcl}\mu\alpha.\langle m * n \parallel \alpha\rangle & \equiv & \mu\alpha.\langle \mu\beta.\langle m \parallel n \bullet \beta\rangle \parallel \alpha\rangle \\ & \xrightarrow[\mu]{} & \mu\alpha.\langle m \parallel n \bullet \alpha\rangle \\ & \equiv & m * n\end{array}$$

According to Curien-Herbelin translation from $\bar{\lambda}\mu\tilde{\mu}$ to $\lambda\mu$ this corresponds to the rule (μ-η) of $\lambda\mu$. Note that this is a rule in $\lambda\mu$ and not in GEMINI. However, we can consider GEMINI_η as GEMINI enriched with

$$\begin{array}{ll}(\eta_\mu)\ \ \mu\alpha.\langle r \parallel \alpha\rangle \to r & \text{if } \alpha \text{ is not free in } r \\ (\eta_{\tilde{\mu}})\ \ \tilde{\mu}x.\langle x \parallel e\rangle \to e & \text{if } x \text{ is not free in } e\end{array}$$

Example 2.3 (A self reproducing term) Let w be $\lambda x.x * x$. Then

$$\begin{array}{rcl}w * w & \equiv & \mu\alpha.\langle \lambda x.x * x \parallel w \bullet \alpha\rangle \\ & \twoheadrightarrow & \mu\alpha.\langle w * w \parallel \alpha\rangle \\ & \twoheadrightarrow & \mu\alpha.\langle \mu\alpha_1.\langle w * w \parallel \alpha_1\rangle \parallel \alpha\rangle \\ & \twoheadrightarrow & \ldots\end{array}$$

So that $w * w$ corresponds to the term $(\lambda x.xx)(\lambda x.xx)$ in λ-calculus. Note that we also have

$$\begin{array}{rcl}w * w & \equiv & \mu\alpha.\langle \lambda x.x * x \parallel w \bullet \alpha\rangle \\ & \twoheadrightarrow & \mu\alpha.\langle w * w \parallel \alpha\rangle \\ & \xrightarrow[\mu]{} & w * w\end{array}$$

as we saw in the Example 2.2.

Example 2.4 (Fixed point) If f is a caller variable, let u_f be $\lambda x.\, f * (x * x)$. Then

$$\begin{array}{rcl} u_f * u_f & \equiv & \mu\alpha.\langle u_f \parallel u_f \bullet \alpha\rangle \\ & \equiv & \mu\alpha.\langle \lambda x.\, f * (x * x) \parallel u_f \bullet \alpha\rangle \\ & \longrightarrow\!\!\!\!\!\rightarrow & \mu\alpha.\langle f * (u_f * u_f) \parallel \alpha\rangle \\ & \xrightarrow[\eta]{} & f * (u_f * u_f) \end{array}$$

So that the term $\lambda f.u_f * u_f$ corresponds to the fixed-point combinator $\mathsf{Y} \equiv \lambda \mathsf{f}.(\lambda \mathsf{x}.\mathsf{f}(\mathsf{xx}))(\lambda \mathsf{x}.\mathsf{f}(\mathsf{xx}))$ in lambda calculus.

3 Classical sequent calculi

3.1 A sequent calculus

Before presenting the Curry-Howard correspondence between classical sequent proofs and $\bar{\lambda}\mu\tilde{\mu}$ programs, let us introduce the deduction system, which is a specific sequent calculus for implications.

First, let us consider a basic sequent calculus for the implicational fragment of classical propositional logic. There propositions are generated as follows:

$$A, B ::= p \mid A \to B$$

p denotes propositional variables and A and B are any proposition. Sequents are expressions of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of propositions. Propositions in Γ are assumptions, whereas propositions in Δ are conclusions. The standard sequent calculus is generated from axioms

$$\frac{}{\Gamma, A \vdash \Delta, A}\,(ax)$$

by left and right introduction rules

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\,(\to L) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}\,(\to R)$$

together with the cut rule

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\,(cut)$$

In this setting there is an easy proof of Peirce's law which is known to be classically valid, but not provable intuitionistically:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash B, A}\,(ax)}{\vdash \boxed{A \to B}, A}\,(\to R) \qquad \dfrac{}{A \vdash A}\,(ax)}{\boxed{(A \to B) \to A} \vdash A}\,(\to L)}{\vdash \boxed{((A \to B) \to A) \to A}}\,(\to R)$$

In this proof we boxed the active propositions, i.e., the proposition in the consequent of a rule which is *split* by the rule (bottom-up), or *created* by the rule (top-down).

3.2 Sequent calculus with active proposition

It is convenient to make the active proposition explicit in the rules. This gives a *sequent classical implicative calculus with active proposition*. The rules of this sequent calculus are given in Figure 1. In each rule, the active proposition (the proposition in the stoup in Girard's sense [13]) is boxed. When applying a rule, one has to take into account where the active proposition is.

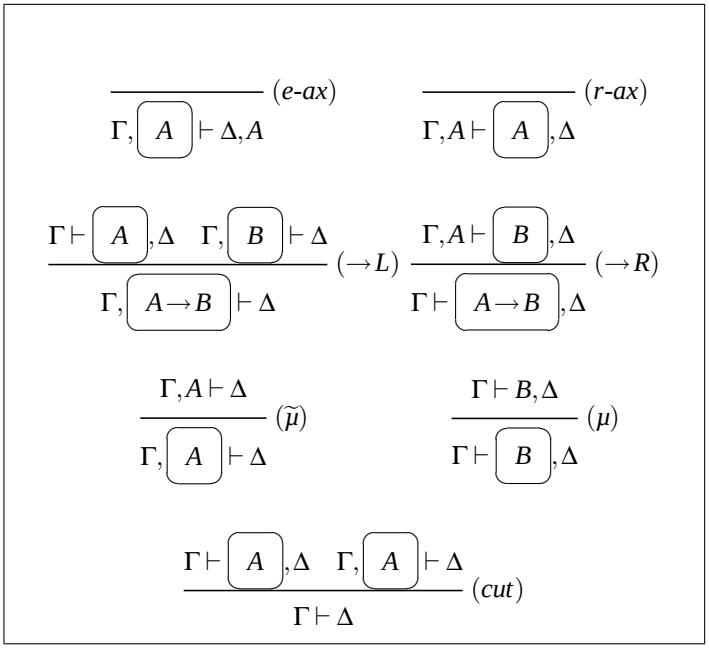

Fig. 1. Sequent calculus with active proposition.

Rather naturally, the active proposition in the cut-rule is the one eliminated by cut, i.e., the cut-formula, in other words the cut-rule makes the newly introduced proposition (bottom-up) the active proposition. This means that there is no active proposition in the consequent of (cut). Therefore, this calculus has three kinds of sequents: sequents with one active proposition on the right, sequents with one active proposition on the left, sequents with no active proposition. Two rules (μ) and $(\widetilde{\mu})$ transform (bottom up) a sequent with an active proposition into a sequent with no active proposition. Notice that the above proof for Peirce's law is not (cannot be made) a proof in this new calculus. Its use of rule $(\rightarrow R)$ does not fulfill the requirement on the active proposition. One has to hack a little (Figure 2). This calculus will be the basis of the Curry Howard correspondence for $\overline{\lambda}\mu\widetilde{\mu}$.

4 Type assignment systems

4.1 Intersection and union types

The form of classical sequent calculus provides the framework for the definition of a type-assignment system for GEMINI using simple types. This is precisely the

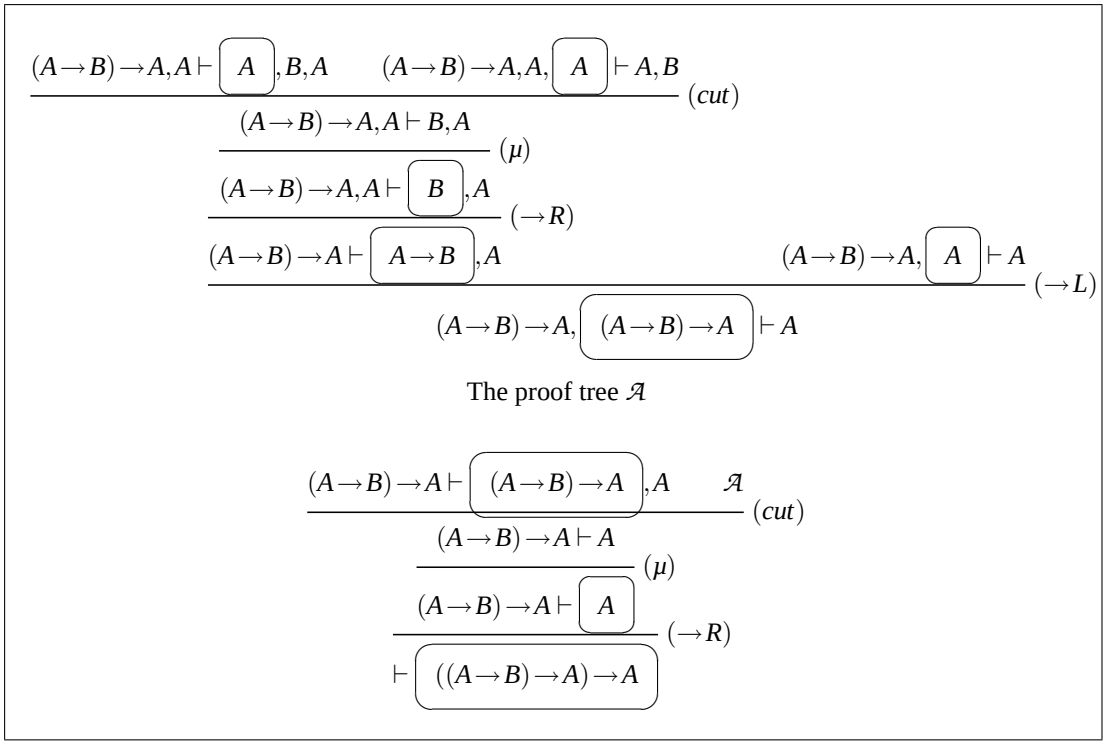

Fig. 2. Peirce's law with active formula

type system $\bar{\lambda}\mu\tilde{\mu}$ of Curien and Herbelin [8], which will be the foundation upon which we build our intersection types.

The informal interpretation of a caller typing judgment $r : A$ is that r denotes a *value* in type A; correspondingly the informal interpretation of a callee typing judgment $e : A$ is that e denotes a *continuation* which expects a value of type A and returns an answer. Capsules return just answers. Under this reading a judgment such as $\mu\alpha.c : A$ says that $\mu\alpha.c$ takes as parameter an A-continuation and returns an answer. If we were to informally denote the space of A-continuations as the set $(A \Rightarrow \bot)$ then this $\mu\alpha.c$ inhabits the set $((A \Rightarrow \bot) \Rightarrow \bot)$, and the fact that such terms are assigned the *type* A is exactly the embodiment of the equivalence of a proposition with its double-negation, hence we are in the world of classical logic.

General consideration of symmetry should lead us to consider union types together with intersection types in our system. If a caller r can have type $A \cap B$, meaning that it denotes values which inhabit both A and B then it can interact with any callee that can receive an A-value *or* a B-value: such a callee will naturally be expected to have the type $A \cup B$. Thus far we have only argued that having intersection types for values suggests having union types for callees, which is in itself not a real extension of the intersection-types paradigm. But any type that can be the type of a caller-variable can be the type of a callee term (via the $\tilde{\mu}$-construction) and any type that can be the type of a callee-variable can be the type of a caller term (via the μ-construction). So we are committed to having intersections *and* unions for callers *and* callees.

Definition 4.1 The set of *types* is generated from a set of *type variables* by the grammar

$$A,B ::= p \mid A \to B \mid A \cap B \mid A \cup B$$

where p ranges over type variables.

A *caller basis* is a set of statements of the form $x : A$, a *callee basis* is a set of statements of the form $\alpha : A$; in each case we stipulate that all variables are distinct.

There are three types of *typing judgments*:

$$\Gamma \vdash \boxed{r : A}, \Delta$$

$$\Gamma, \boxed{e : A} \vdash \Delta$$

$$c \ : \ (\Gamma \vdash \Delta)$$

where Γ is a caller basis and where Δ is a callee basis. □

We will always consider types to be defined modulo commutativity and associativity for $\cap$ and $\cup$.

4.2 First attempt: a naive type system

Consider a standard intersection type system for λ-calculus. At the level of the (natural deduction) *logic*, the rules for $\cap$ are just the rules for $\wedge$ (that is, if one erases the terms and looks just at the formulas). As we know, the difference between $A \cap B$ and $A \wedge B$ is that we require the same term to witness A, B and $A \cap B$.

We can imagine a type system for GEMINI derived using the same principle, applied to sequent calculus deduction. So the $\cap$ rules would be based on the shape of the logic rules for $\wedge$:

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}$$

A key point is that, since introducing an intersection on the left or the right is *not* a logical inference, these type inferences are completely orthogonal to the notion of the stoup, i.e. of the active formula. This means that to write down the typing judgments corresponding to these rules in the context of $\bar{\lambda}\mu\tilde{\mu}$, we should write down several different $\bar{\lambda}\mu\tilde{\mu}$ sequents whose "erasure" looks like a given logic inference. For example, consider

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$$

The formula A on the left-hand side can be the type of an active formula (i.e. the type of a callee), or it can be the type of a caller-variable, and in the latter case the judgment can be typing a caller or a callee. Thus, in terms of $\bar{\lambda}\mu\tilde{\mu}$, this one logic

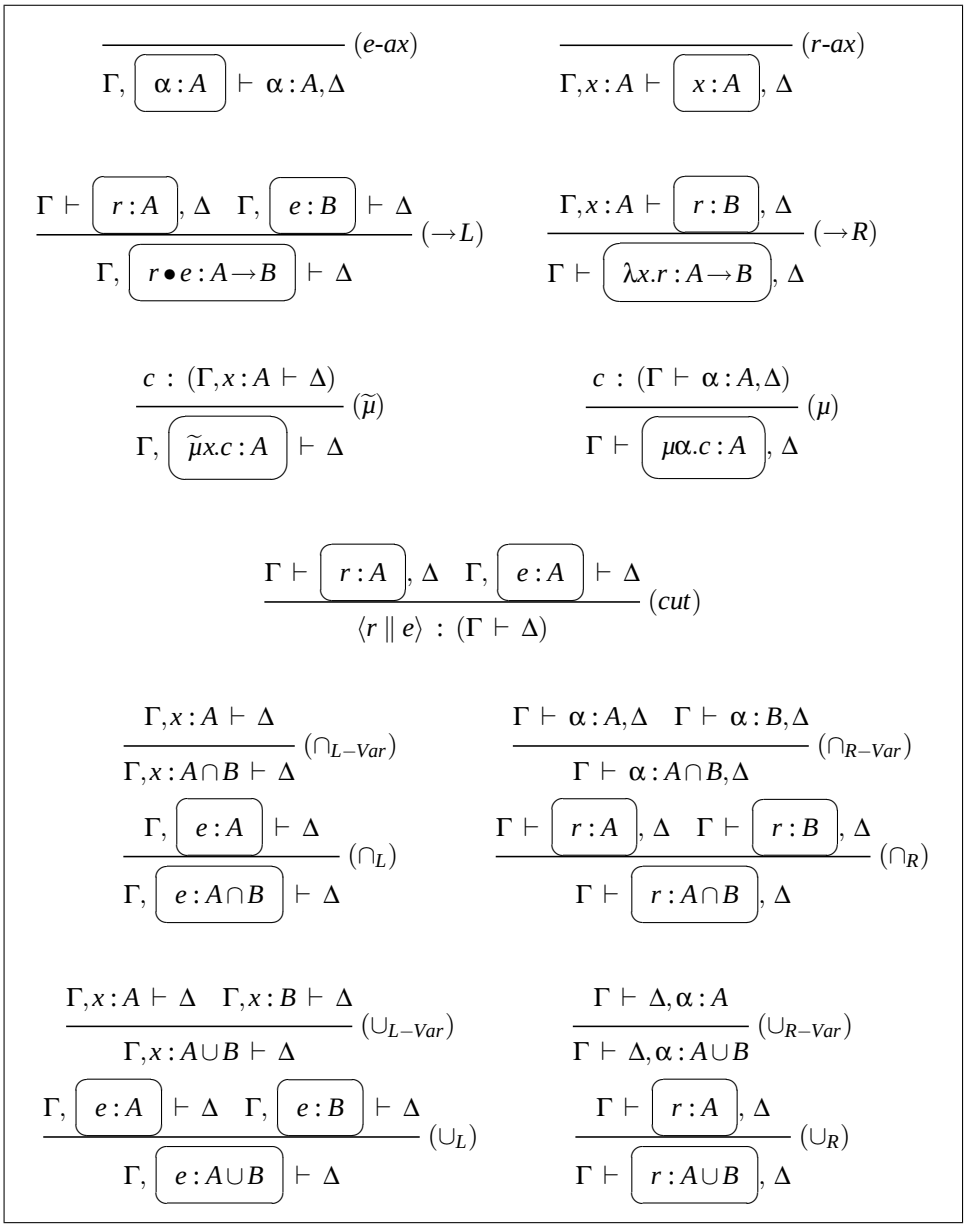

Fig. 3. A naive type system.

inference would yield three different typing rules, as follows.

$$\frac{\Gamma, \boxed{e : A} \vdash \Delta}{\Gamma, \boxed{e : A \cap B} \vdash \Delta} \qquad \frac{\Gamma, x : A \vdash \boxed{r : T}, \Delta}{\Gamma, x : A \cap B \vdash \boxed{r : T}, \Delta} \qquad \frac{\Gamma, x : A, \boxed{e : T} \vdash \Delta}{\Gamma, x : A \cap B, \boxed{e : T} \vdash \Delta}$$

Definition 4.2 [A naive system]
The axioms and rules of this type system are given in Figure 3

Let us mention here that the type system from Figure 3 restricted to $\rightarrow$ types only is exactly the $\bar{\lambda}\mu\tilde{\mu}$ of Curien and Herbelin [8]. It is well known that Peirce's

law is intuitionistically not provable and therefore not inhabited in simply typed λ-calculus. Peirce's law is inhabited in $\bar{\lambda}\mu\tilde{\mu}$ by the term $\lambda x.\mu\alpha.\langle x \parallel (\lambda y.\mu\beta.\langle y \parallel \alpha\rangle) \bullet \alpha\rangle$, since one can prove the following in $\bar{\lambda}\mu\tilde{\mu}$

$$\vdash \boxed{\lambda x.\mu\alpha.\langle x \parallel (\lambda y.\mu\beta.\langle y \parallel \alpha\rangle) \bullet \alpha\rangle : ((A \to B) \to A) \to A}$$

Ong and Stewart [17] give the $\lambda\mu$ term $\lambda x.\mu\alpha.[\alpha](x(\lambda y.\mu\beta.[\alpha]y))$ as the "simplest witness" of Peirce's law. One gets the G-term $\lambda x.\mu\alpha.\langle\mu\gamma.\langle x \parallel \lambda y.\mu\beta.\langle y \parallel \alpha\rangle \bullet \gamma\rangle \parallel \alpha\rangle$ using the Curien Herbelin translation [8]. It reduces in one (μ) step to our witness $\lambda x.\mu\alpha.\langle x \parallel (\lambda y.\mu\beta.\langle y \parallel \alpha\rangle) \bullet \alpha\rangle$.

This shows that $\bar{\lambda}\mu\tilde{\mu}$ gives a more compact witness for the type $((A \to B) \to A) \to A$. Actually the image of $\lambda\mu$-terms in $\bar{\lambda}\mu\tilde{\mu}$ by the translation are not all the callers, but only the G-terms given by the grammar:

$$r_{\lambda\mu} \quad ::= \quad x \mid \lambda x.r_{\lambda\mu} \mid \mu\alpha.\langle r_{\lambda\mu} \parallel r_{\lambda\mu} \bullet \alpha\rangle \mid \mu\alpha.\langle r_{\lambda\mu} \parallel \alpha\rangle$$

Still in $\bar{\lambda}\mu\tilde{\mu}$ one cannot type all normal forms, e.g., the G-term $\lambda x.\mu\alpha.\langle x \parallel x \bullet \alpha\rangle$ (seen in Section 2) which is a normal form and corresponds to the lambda term $\lambda x.xx$ is not typeable in $\bar{\lambda}\mu\tilde{\mu}$. This is one of the reason to introduce new type assignment rules.

As noted above, the schemas $\cap_{L-Var}$, $\cap_{R-Var}$, $\cup_{L-Var}$, and $\cup_{R-Var}$ are really describing two rules each, one for when the associated judgment types a caller, one for when the associated judgment types a callee. That is, one should read these knowing that the active formula may lie within the Γ or within the Δ (here we are temporarily abusing our notational convention that Γ and Δ are just variable-type bindings).

This is a perfectly sensible typing system. But as Barbanera et al. [2] have noted, union types are technically difficult, for example Subject Reduction tends to fail. And indeed the system above leads to difficulties in Subject Reduction (more specifically, it seems difficult to prove the Substitution Lemma).

In particular, if we want to prove:

$$\text{If} \quad \Gamma, x : A \vdash \boxed{r : T}, \Delta \quad \text{and} \quad \Gamma \vdash \boxed{s : A}, \Delta \quad \text{then} \quad \Gamma \vdash \boxed{r[x \leftarrow s] : T}, \Delta$$

we have a problem when the assumed typing was derived using $\cup_{L-Var}$:

$$\frac{\Gamma, x : A_1 \vdash \boxed{r : T}, \Delta \quad \Gamma, x : A_2 \vdash \boxed{r : T}, \Delta}{\Gamma, x : A_1 \cup A_2 \vdash \boxed{r : T}, \Delta}\ (\cup_{L-Var})$$

since knowing

$$\Gamma \vdash \boxed{s : A_1 \cup A_2}, \Delta$$

doesn't allow us to use the induction hypothesis on the given $\Gamma, x : A_i \vdash \boxed{r : T}, \Delta$.

4.3 Second attempt: the "definite" type system

The problem with the naive system seems to be with variable-typings of the form $x : A \cup B$ and $\alpha : A \cap B$. Since, in typing normal forms we only require union types for callees and intersection types for callers, a first idea for fixing this problem is to forbid union types for callers and forbid intersection types for callees.

This is immediately a failure since in the presence of μ and $\widetilde{\mu}$, any type which can be the type of a callee variable can arise as the type of a caller term, and any type which can be the type of a caller variable can arise as the type of a callee term.

But it turns out that we get a successful system if we simply forbid typing judgments whose bases contain variable-typings of the form $x : A \cup B$ and $\alpha : A \cap B$.

We need a slightly more general notion, to ensure that we can type caller-variables by intersections of types which themselves forbid $\cup$, and dually for callee-variables.

Definition 4.3

(i) A type A is $\cap$-*definite* if it is a type variable, an arrow-type or it is $A_1 \cap A_2$, with each A_i a $\cap$-definite type. A type A is $\cup$-*definite* if it is a type variable, an arrow-type or it is $A_1 \cup A_2$, with each A_i a $\cup$-definite type.

(ii) A basis Γ is $\cap$-*definite*, if in each binding $x : A$ in Γ, A is a $\cap$-definite type. A typing judgment Δ is $\cup$-*definite* if in each binding $\alpha : A$ in Δ, A is a $\cup$-definite type.

(iii) A typing judgment is *definite* if its typing bases Γ and Δ are $\cap$- and $\cup$-definite, respectively.

Note that in a definite typing judgment we do not insist that the type of the active formula be definite.

The system of Definition 4.4 is the system obtained from the natural generalization of the sequent calculus by forbidding rules $\cap_{R-Var}$ and $\cup_{L-Var}$ and insisting that typing judgments be definite.

As noted earlier, the schemas $\cap_{L-Var}$ and $\cup_{R-Var}$ are really describing two rules each, one for when the associated judgment types a caller, one for when the associated judgment types a callee. That is, one should read these knowing that the active formula may lie within the Γ or within the Δ (here we are temporarily abusing our notational convention that Γ and Δ are just variable-type bindings).

Definition 4.4 [A "definite" system]
The axioms and rules of this type system are given in Figure 4. In each rule below we assume that the type bases are definite.

4.4 The type system $\mathcal{M}^{\cap\cup}$

It turns out that the presence of rules $\cap_{L-Var}$ and $\cup_{R-Var}$ complicates reasoning about this system. But it is not hard to see that application of these rules can always be pushed towards the leaves of a typing tree. In fact an equivalent formulation of

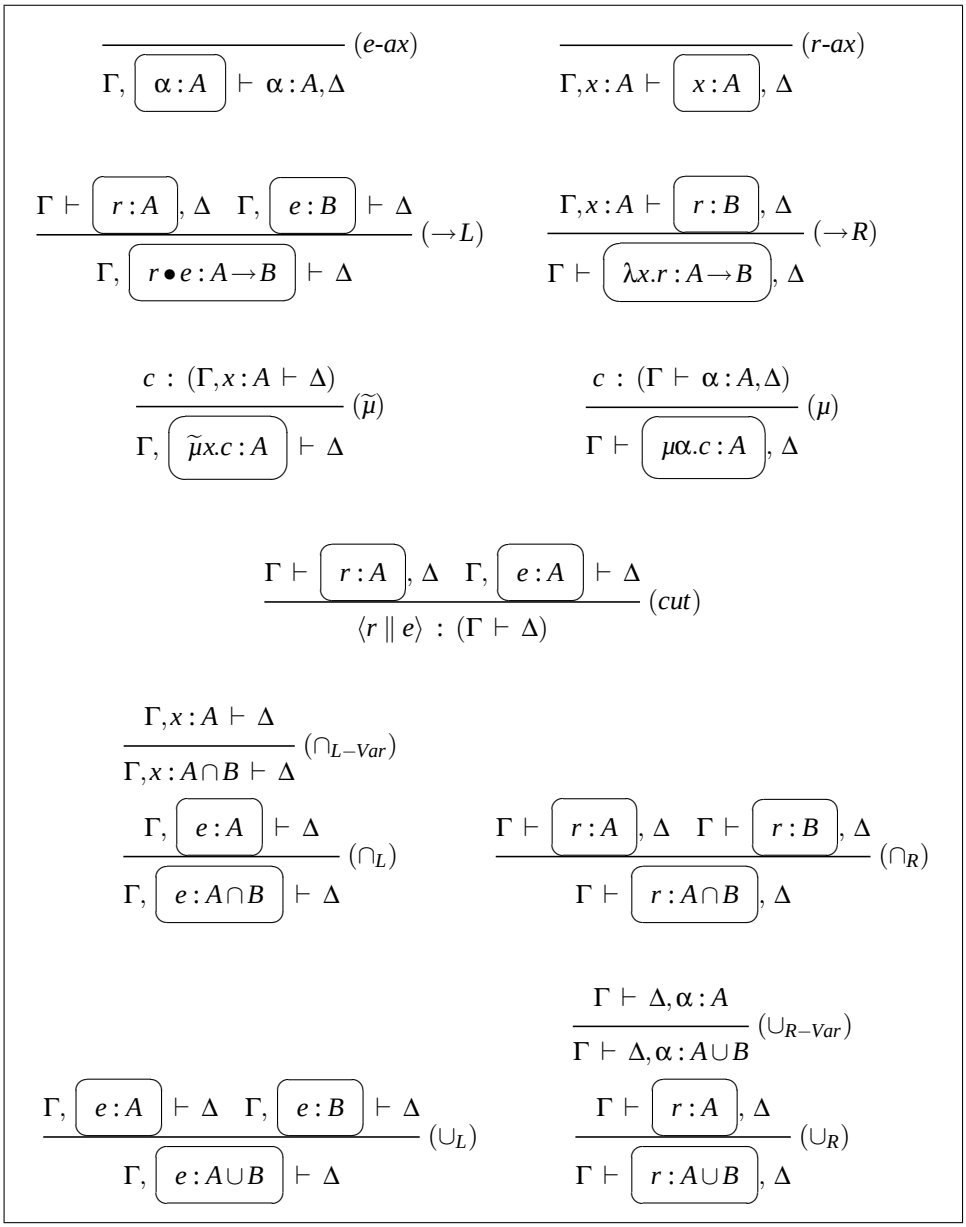

Fig. 4. A "definite" type system.

the system removes these rules completely and replaces them with more flexible axiom schemas.

The system in Definition 4.5 is the system obtained from the basic system by replacing the rules $\cap_{L-Var}$ and $\cup_{R-Var}$ by the more flexible axioms e^+-*ax* and r^+-*ax*.

This system fits well with the sequent calculus based on active formulas, since all the rules concern the terms associated with the active formula, as opposed to the earlier two systems, whose var-rules changed the type basis for an active term while keeping the type of the term the same.

Finally, note that although we have forbidden caller variables to have union types, we **do** have caller terms with union types, due to their typing rule μ. This

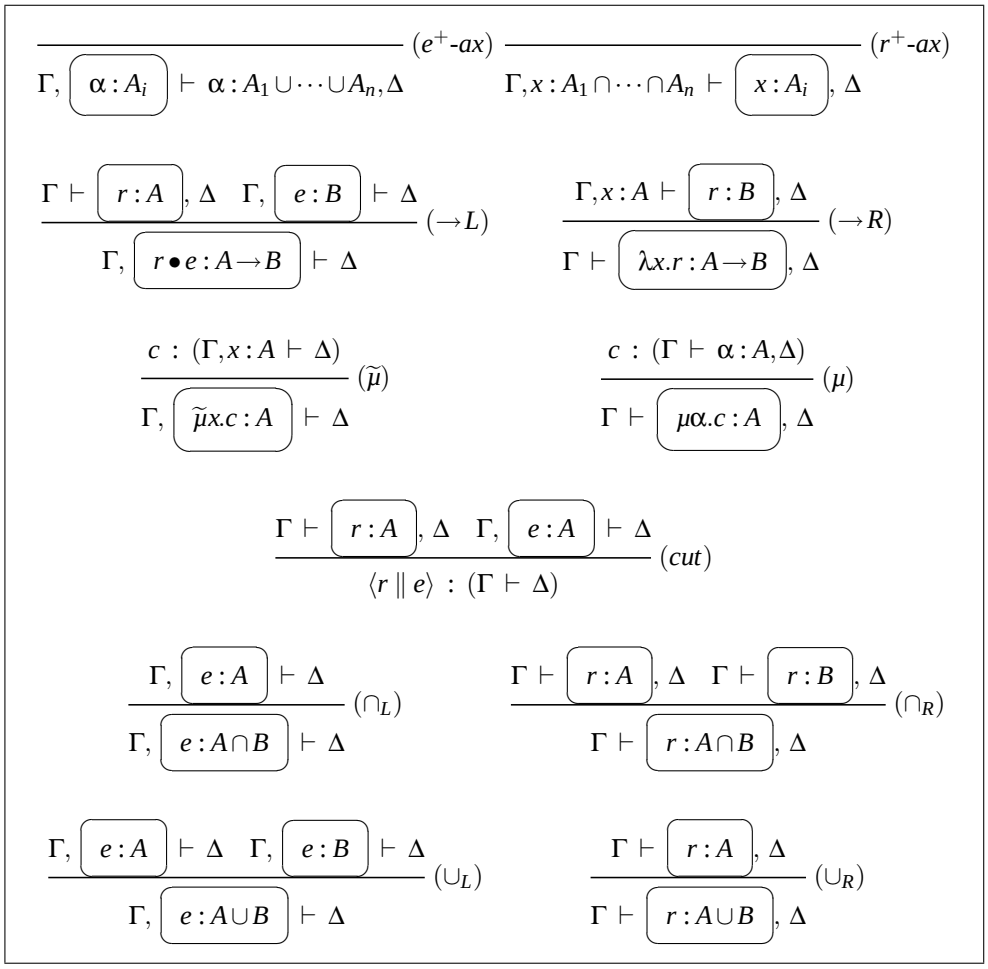

Fig. 5. The system $\mathcal{M}^{\cap\cup}$

(constrained) version of union types is interesting because it enjoys the Subject Reduction property.

Definition 4.5 [The type system $\mathcal{M}^{\cap\cup}$]
The axioms and rules of this type system are given in Figure 5. In each rule below we assume that the type bases are definite.

Lemma 4.6 *The rules in Definitions 4.4 and 4.5 generate the same typing judgments.*

Proof. First we observe that the rules e^+-*ax* and r^+-*ax* are admissible with respect to the basic system. This is clear by considering the rules $\cap_{L-Var}$ and $\cup_{R-Var}$.

Next we show that rules $\cap_{L-Var}$ and $\cup_{R-Var}$ can be eliminated in the presence of e^+-*ax* and r^+-*ax*. This follows from the fact that any use of $\cap_{L-Var}$ or $\cup_{R-Var}$ can be pushed upwards in a typing tree until it is applied just after an axiom. So replacing the old axioms by e^+-*ax* and r^+-*ax* is enough. □

5 Properties of $\mathcal{M}^{\cap\cup}$

For the rest of this paper, we work only with the type system $\mathcal{M}^{\cap\cup}$.

The first property we will need is that the intersection and union rules with two premises can be "inverted" in the sense that if the judgment in the conclusion of the rule is derivable then each of the judgments in the hypotheses are derivable.

It is precisely here that we reap the benefit of our restriction to definite bases. The lemma is false without this restriction.

Lemma 5.1 (Elimination)

(i) *If* $\Gamma \vdash \boxed{r : A_1 \cap A_2}, \Delta$, *then for* $i = 1,2$, $\Gamma \vdash \boxed{r : A_i}, \Delta$.

(ii) *If* $\Gamma, \boxed{e : A_1 \cup A_2} \vdash \Delta$, *then for* $i = 1,2$, $\Gamma, \boxed{e : A_i} \vdash \Delta$.

Proof. For part 1, we just observe that the only rules that could be used to derive $\Gamma \vdash \boxed{r : A_1 \cap A_2}, \Delta$ are r^+-*ax* and $\cap_R$. In the latter case the result is immediate; and in the former case the result is a consequence of the fact that we are considering types modulo associativity and commutativity of $\cap$.

The fact that the last inference cannot be a μ is a direct consequence of our assumption that bases are definite. That is, since bases cannot have assumptions of the form $\alpha : A_1 \cap A_2$ the assumed derivation must look like

$$\frac{\Gamma \vdash \boxed{\mu\alpha.\langle r \parallel e\rangle : A_1}, \Delta \quad \Gamma \vdash \boxed{\mu\alpha.\langle r \parallel e\rangle : A_2}, \Delta}{\Gamma \vdash \boxed{\mu\alpha.\langle r \parallel e\rangle : A_1 \cap A_2}, \Delta}$$

that is, an instance of $\cap_R$.

This completes the proof of part 1 of the lemma. The proof of part 2 is similar.□

The following technical properties are in support of the Subject Reduction (Theorem 5.6).

Lemma 5.2 (Context expansion lemma) *Let* $\Gamma \subseteq \Gamma'$ *and* $\Delta \subseteq \Delta'$.

(i) *If* $\Gamma \vdash \boxed{r : A}, \Delta$, *then* $\Gamma' \vdash \boxed{r : A}, \Delta'$.

(ii) *If* $\Gamma, \boxed{e : A} \vdash \Delta$, *then* $\Gamma', \boxed{e : A} \vdash \Delta'$.

(iii) *If* $c : (\Gamma \vdash \Delta)$, *then* $c : (\Gamma' \vdash \Delta')$.

Lemma 5.3 (Context restriction lemma)

(i) *If* $\Gamma \vdash \boxed{r : A}, \Delta$, *then* $\Gamma \restriction \mathtt{Fv_r}(r) \vdash \boxed{r : A}, \Delta \restriction \mathtt{Fv_e}(r)$.

(ii) *If* $\Gamma, \boxed{e : A} \vdash \Delta$, *then* $\Gamma \restriction \mathtt{Fv_r}(e), \boxed{e : A} \vdash \Delta \restriction \mathtt{Fv_e}(e)$.

(iii) *If* $c : (\Gamma \vdash \Delta)$, *then* $c : (\Gamma \restriction \mathtt{Fv_r}(c) \vdash \Delta \restriction \mathtt{Fv_e}(c))$.

Lemma 5.4 (Generation lemma)

(i) *If* $\Gamma \vdash \boxed{\lambda x.r : \bigcap_{i\in I} A_i \to B_i}, \Delta$, *then* $\Gamma, x : A_i \vdash \boxed{r : B_i}, \Delta$.

(ii) *If* $\Gamma, \boxed{r \bullet e : \bigcup_{i\in I} A_i \to B_i} \vdash \Delta$, *then* $\Gamma \vdash \boxed{r : A_i}, \Delta$ *and* $\Gamma, \boxed{e : B_i} \vdash \Delta$.

(iii) *If* $\Gamma \vdash \boxed{\mu\alpha.c : \bigcap_{i\in I} A_i}, \Delta$, *then* $c : (\Gamma \vdash \alpha : A_i, \Delta)$.

(iv) *If* $\Gamma, \boxed{\widetilde{\mu}x.c : \bigcup_{i\in I} A_i} \vdash \Delta$, *then* $c : (\Gamma, x : A_i \vdash \Delta)$.

Lemma 5.5 (Substitution) *Let us suppose that all judgments are definite.*

(i) *If* $\Gamma, x : S \vdash \boxed{t : T}, \Delta$ *and* $\Gamma \vdash \boxed{s : S}, \Delta$, *then* $\Gamma \vdash \boxed{t[x \leftarrow s] : T}, \Delta$.

(ii) *If* $\Gamma \vdash \boxed{t : T}, \Delta, \alpha : S$ *and* $\Gamma, \boxed{f : S} \vdash \Delta$, *then* $\Gamma \vdash \boxed{t[\alpha \leftarrow f] : T}, \Delta$.

(iii) *If* $\Gamma, x : S, \boxed{g : T} \vdash \Delta$ *and* $\Gamma \vdash \boxed{s : S}, \Delta$, *then* $\Gamma, \boxed{g[x \leftarrow s] : T} \vdash \Delta$.

(iv) *If* $\Gamma, \boxed{g : T} \vdash \Delta, \alpha : S$ *and* $\Gamma, \boxed{f : S} \vdash \Delta$, *then* $\Gamma, \boxed{g[\alpha \leftarrow f] : T} \vdash \Delta$.

Our type system enjoys the Subject Reduction property, for the calculus with unrestricted reduction, that is, even in the presence of the $(\mu, \widetilde{\mu})$ critical pair. As mentioned in the introduction this has been shown in [2] to be a difficult property to achieve in a system with union types.

Theorem 5.6 (Subject Reduction [10])

If $c : (\Gamma \vdash \Delta)$ *and* $c \to c'$ *then* $c' : (\Gamma \vdash \Delta)$.

The typeability of all strongly normalizing terms, a unique property of some traditional λ-calculus systems with intersection types, holds in the system $\mathcal{M}^{\cap\cup}$.

Theorem 5.7 (Strong Normalization [10])

All strongly normalizing terms of GEMINI *are typeable in* $\mathcal{M}^{\cap\cup}$.

6 Conclusion

We defined three systems of intersection and union types, defined in terms of sequents and discussed some properties. The final system, $\mathcal{M}^{\cap\cup}$ has the properties that the typable terms are precisely the strongly normalizing terms [10]. In contrast to systems with union types investigated previously, the type assignment system $\mathcal{M}^{\cap\cup}$enjoys the Subject Reduction property for unrestricted reduction.

Some directions for future research.

Intersection types have proven to be an invaluable tool for studying reduction properties in the traditional λ-calculus, and in future work we expect to use suitable

variants on the system presented here to characterize weak normalization and head-normalization in Gemini.

It is well known that traditional λ-calculus with intersection types does not fit into the Curry-Howard (proofs-as-terms) correspondence. This makes the intersection a proof-theoretical and not a truth-functional connective. There have been several attempts to develop a typed system (à la Church) with intersection types by Dezani et al. [9], Ronchi and Roversi [22], Capitani et al. [5] and recently by Wells and Haack [25]. This direction of research in the framework of $\lambda\mu$-calculus merits attention.

It will be interesting to investigate the relationship between the system presented here and that of Laurent [16].

It is important to better understand the role of union types in a classical calculus. An obvious question is whether the price we pay for having Subject Reduction in our system is a decrease in expressive power relative to the systems in [19] and [4], and if so, we should try to understand the trade-offs.

References

[1] S. van Bakel. Intersection type assignment systems. *Theoretical Computer Science*, 38(2):246–269, 1997.

[2] F. Barbanera, M. Dezani-Ciancaglini, and U. de' Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[3] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[4] P. Buneman and B. Pierce. Union types for semistructured data. In *Internet Programming Languages*, volume 1949 of *Lecture Notes in Computer Science*, pages 184–207. Springer-Verlag, September 2000. Proceedings of the International Database Programming Languages Workshop.

[5] B. Capitani, M. Loreti, and B. Venneri. Hyperformulae, parallel deductions and intersection types. In *Electronic Notes in Theoretical Computer Science*, volume 50. Elsevier, 2001.

[6] M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for lambda terms. *Archiv für Mathematische Logik*, 19:139–156, 1978.

[7] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[8] P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of the 5th ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montreal, Canada, 2000. ACM Press.

[9] M. Dezani-Ciancaglini, S. Ghilezan, and B. Venneri. The "relevance" of intersection and union types. *Notre Dame Journal of Formal Logic*, 38(2):246–269, 1997.

[10] D. Dougherty, S. Ghilezan, and P. Lescanne. Characterizing strong normalization in a language with control operators. In *The 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2004)*, Verona, Italy, 2004.

[11] J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In A.D. Gordon, editor, *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'03)*, volume 2620 of *Lecture Notes in Computer Science*, pages 250–266, Warsaw, Poland, April 2003. Springer-Verlag.

[12] J. Dunfield and F. Pfenning. Tridirectional typechecking. In X.Leroy, editor, *Conference Record of the 31st Annual Symposium on Principles of Programming Languages (POPL'04)*, pages 281–292, Venice, Italy, January 2004. ACM Press.

[13] J.-Y. Girard. A new constructive logic: classical logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991.

[14] T. Griffin. A formulae-as-types notion of control. In *Proceedings of the 17th Annual ACM Symposium on Principles Of Programming Languages (POPL'90), Orlando (Fla., USA)*, pages 47–58, 1990.

[15] H. Herbelin. *Séquents qu'on calcule : de l'interprétation du calcul des séquents comme calcul de λ-termes et comme calcul de stratégies gagnantes*. Thèse d'université, Université Paris 7, Janvier 1995.

[16] O. Laurent. On the denotational semantics of the pure lambda-mu calculus. *Manuscript*, 2004.

[17] C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceedings of the 24th Annual ACM Symposium on Principles Of Programming Languages (POPL'97), Paris (France)*, pages 215–227, 1997.

[18] M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proc. of Int. Conf. on Logic Programming and Automated Reasoning, LPAR'92*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 1992.

[19] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[20] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, London, 1980.

[21] J. C. Reynolds. Design of the programming language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 28, 1996.

[22] S. Ronchi and L. Roversi. Intersection logic. In *Computer Science Logic, CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 421–428. Springer-Verlag, 2001.

[23] P. Sallé. Une extension de la théorie des types en lambda-calcul. In G. Ausiello and C. Böhm, editors, *Fifth International Conference on Automata, Languages and Programming*, volume 62 of *Lecture Notes in Computer Science*, pages 398–410. Springer-Verlag, 1978.

[24] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A calculus with polymorphic and polyvariant flow types. *Journal of Functional Programming*, 12(3):183–227, May 2002.

[25] J.B. Wells and C. Haack. Branching types. *Information and Computation*, 200X.