Electronic Notes in Theoretical Computer Science 136 (2005) 153–172 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Intersection and Union Types in the

*λµµ*˜-calculus

# Daniel J. Dougherty [1](#_bookmark0)

*Worcester Polytechnic Institute Worcester MA 01609 USA*

# Silvia Ghilezan[2](#_bookmark0)

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

# Pierre Lescanne[3](#_bookmark0)

*LIP, E´cole Normale Sup´erieure de Lyon Lyon, France*

Abstract

The original *λµµ*e 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 *λµµ*e with intersection and union types. The intrinsic

symmetry in the *λµµ*e calculus leads to an essential use of both intersection and union types.

*Keywords:* Intersection types, union types, *λµµ*e-calculus, classical logic, Curry-Howard correspondence.

1 Email: [dd@cs.wpi.edu](mailto:dd@cs.wpi.edu)

2 Email: [gsilvia@uns.ns.ac.yu](mailto:gsilvia@uns.ns.ac.yu)

3 Email: [Pierre.Lescanne@ens-lyon.fr](mailto:Pierre.Lescanne@ens-lyon.fr)

1571-0661 © 2005 Published by Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

doi:10.1016/j.entcs.2005.06.010

# Introduction

Intersection types were introduced into the lambda calculus in the late 1970s by Coppo and Dezani [[5](#_bookmark19),[6](#_bookmark20)], Pottinger [[19](#_bookmark33)] and Sall´e [[23](#_bookmark37)]. 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 characterize exactly all strongly normalizing lambda terms. In addition, these systems are suitable for analyzing *λ*-models and various nor- malization properties of *λ*-terms. A summary of the early research was given by van Bakel [[24](#_bookmark38)].

Barbanera et al. [[1](#_bookmark15)] 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 [[20](#_bookmark34)] on the language Forsythe). Pierce [[18](#_bookmark32)] explored the use of union types in programming, for example as a generalization of variant records. More recently Buneman and Pierce [[3](#_bookmark17)], 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 intuitionis- tic 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 [[13](#_bookmark26)], by observing that classical tautologies suggest typings for certain control operators. This initiated an active line of research; in partic- ular the *λµ* calculus of Parigot [[17](#_bookmark31)] embodies a Curry-Howard correspondence for classical logic based on natural deduction.

Meanwhile Curien and Herbelin [[7](#_bookmark21)], building on earlier work in [[14](#_bookmark28)], de- fined the system *λµµ*. In contrast to Parigot’s *λµ*-calculus, which bases its type system on a natural deduction system for classical logic, terms in *λµµ*

˜

˜

represent derivations in a *sequent calculus* proof system and reduction reflects

the process of cut-elimination.

˜

In this paper we recount our experience in extending the language *λµµ* in an untyped language which we propose to call Gemini (see [[21](#_bookmark35)] for similar attempt) an then deriving a type system for this extended language 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 [[9](#_bookmark22)] that union types are also necessary in order to completely

characterize all strongly normalizing untyped *λµµ*-terms (in contrast to the situation in standard *λ*-calculus).

Remarkably, Laurent [[15](#_bookmark29)] has recently and independently set out to analyze the denotational semantics of the *λµ* 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 [[18](#_bookmark32)] highlighted the failure of Subject Reduction in the presence of union types and [[1](#_bookmark15)] showed how to recover this property by suitably restricting the notion of reduction. Wells et. al. [[25](#_bookmark39)] 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 [[10](#_bookmark23),[11](#_bookmark24)] Dunfield and Pfenning investigate a type sys- tem 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 rela- tionship 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 organized as follows. Section [2](#_bookmark1) deals with the untyped syntax of *λµµ*, which we call Gemini. In Section [3](#_bookmark5) we discuss sequent calculi which correspond to types of *λµµ*. In Section [4](#_bookmark7) three type assignment systems are introduced which are extensions of *λµµ* with intersection and union types. Their properties are discussed in Section [5](#_bookmark13).

˜

˜

˜

# The syntax of Gemini

In this section we present the language Gemini which is the untyped version of Curien and Herbelin’s *λµµ* calculus introduced in [[7](#_bookmark21)]. We identify three syntactic categories: *callers*, *callees*, and *capsules* (in [[7](#_bookmark21)] 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* | *λx.r* | *µα.c e* ::= *α* | *r* • *e* | *µ*˜*x.c c* ::= ⟨*r * *e*⟩

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

1. the set *V arr* of *caller variables* denoted by Latin variables *x*, *y*,... which represent inputs, in particular they are bound by *λ*-abstractions or *µ*- abstractions,

˜

1. the set *V are* of *callee variables* denoted by Greek variables *α*,*β*,... which represent continuations and which can be bound by *µ*-abstractions. In *λµ*, they are called *µ*-variables.

The core of Gemini is made of capsules ⟨*caller * *callee*⟩ 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

precise.

has three reduction rules which make this interpretation more

(*λ*) ⟨*λx.r * *r*' • *e*⟩ z⟨,*r*'  *µ*˜*x.*⟨*r * *e*⟩⟩

(*µ*) ⟨*µα.c * *e*⟩ z*c*,[*α* ← *e*]

(*µ*˜) ⟨*r*

 *µ*˜*x.c*⟩ z*c*,[*x* ← *r*]

Note that on a term of the form ⟨*µα.c * *µ*˜*x.c*⟩ rules (*µ*) and (*µ*˜) can be applied

ambiguously. This will be discussed in Subsection [2.1](#_bookmark2). If one gives priority to

(*µ*) over (*µ*˜) 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 *µ*-step. This will be discussed in Subsection [2.2](#_bookmark3).

˜

Of course the substitutions above are defined so as to avoid variable- capture. In this paper, we use the “Barendregt convention” on variables [[2](#_bookmark16)]. 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 *µ* 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 Fvr(*r*), Fve(*r*), Fvr(*e*), Fve(*e*), Fvr(*c*) and Fve(*c*). For instance

Fvr(⟨*x * *µα.*⟨*y * *α*⟩• *β*⟩)= {*x, y*}

Fve(⟨*x * *µα.*⟨*y * *α*⟩• *β*⟩)= {*β*}

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

*rnf* ::= *x* | *λx.rnf* | *µα.cnf enf* ::= *α* | *rnf* • *enf* | *µ*˜*x.cnf*

*cnf* ::= ⟨*x α*⟩ | ⟨*x rnf* • *enf* ⟩ | ⟨*λx.rnf α*⟩

* 1. *Failure of confluence*

˜

˜

As a rewriting calculus Gemini (even in its typed version *λµµ*) has an essential critical pair, between the *µ* and the *µ* 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

⟨*µα.*⟨*z*1  *β*1⟩  *µ*˜*x.*⟨*z*2  *β*2⟩⟩

reduces to each of ⟨*z*1  *β*1⟩ and ⟨*z*2  *β*2⟩.

This is more than simply a reflection of the well-known fact that the equa-

tional 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 *µ*˜ 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.

* 1. *Encoding of λ-calculus*

It is not hard to see that Gemini is Turing-complete as a programming lan- guage, 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 *µα.*⟨*m * *n* • *α*⟩. (Of course *α* is not free in *m* and *n* here.)

Example 2.1 (Classical beta-reduction) (*λx.r*) ∗ *s* ≡ *µα.*⟨*λx.r * *s* • *α*⟩

zz*µ*,,*α.*⟨*r*[*x* ← *s*]  *α*⟩

Example 2.2 (Eta reduction) Notice that

*µα.*⟨*m* ∗ *n * *α*⟩ ≡ *µα.*⟨*µβ.*⟨*m * *n* • *β*⟩  *α*⟩

*µ* z*µ*,*α.*⟨*m * *n* • *α*⟩

≡ *m* ∗ *n*

˜

According to Curien-Herbelin translation from *λµµ* to *λµ* this corresponds to the rule (*µ*-*η*) of *λµ*. Note that this is a rule in *λµ* and not in Gemini.

However, we can consider Gemini*η* as Gemini enriched with

(*ηµ*) *µα.*⟨*r * *α*⟩ → *r* if *α* is not free in *r*

(*ηµ*e)

*µ*˜*x.*⟨*x * *e*⟩ → *e* if *x* is not free in *e*

Example 2.3 (A self reproducing term) Let *w* be *λx.x* ∗ *x*. Then

*w* ∗ *w* ≡ *µα.*⟨*λx.x* ∗ *x * *w* • *α*⟩

zz*µ*,,*α.*⟨*w* ∗ *w * *α*⟩

 zz*µ*,,*α.*⟨*µα*1*.*⟨*w* ∗ *w * *α*1⟩  *α*⟩

zz*.*,,*..*

So that *w* ∗ *w* corresponds to the term (*λx.xx*)(*λx.xx*) in *λ*-calculus. Note that we also have

*w* ∗ *w* ≡ *µα.*⟨*λx.x* ∗ *x * *w* • *α*⟩

zz*µ*,,*α.*⟨*w* ∗ *w * *α*⟩

*µ* z*w*,∗ *w*

as we saw in the Example [2.2](#_bookmark4).

Example 2.4 (Fixed point) If *f* is a caller variable, let *uf* be *λx.f* ∗ (*x* ∗ *x*).

Then

*uf* ∗ *uf* ≡ *µα.*⟨*uf * *uf* • *α*⟩

≡ *µα.*⟨*λx.f* ∗ (*x* ∗ *x*)  *uf* • *α*⟩

zz*µ*,,*α.*⟨*f* ∗ (*uf* ∗ *uf* )  *α*⟩

*η* z*f*,∗ (*uf* ∗ *uf* )

So that the term *λf.uf* ∗ *uf* corresponds to the fixed-point combinator

Y ≡ *λ*f*.*(*λ*x*.*f(xx))(*λ*x*.*f(xx)) in lambda calculus.

# Classical sequent calculi

Before presenting the Curry-Howard correspondence between classical sequent proofs and *λµµ* 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* | *A* → *B*

*p* denotes propositional variables and *A* and *B* are any proposition. Sequents are expressions of the form Γ ▶ ∆, where Γ and ∆ are sets of propositions. Propositions in Γ are assumptions, whereas propositions in ∆ are conclusions. The standard sequent calculus is generated from axioms

(*ax*)

Γ*,A* ▶ ∆*,A*

by left and right introduction rules

Γ ▶ *A,* ∆ Γ*,B* ▶ ∆

(→ *L*)

Γ*,A* → *B* ▶ ∆

together with the cut rule

Γ ▶ *A,* ∆ Γ*,A* ▶ ∆ Γ ▶ ∆

Γ*,A* ▶ *B,* ∆

Γ ▶ *A* → *B,* ∆

(*cut*)

(→ *R*)

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

(*ax*)

*A* ▶ *B, A*

*, ,* (→ *R*)

*z ~~,~~ r ,*

▶

*A* → *B* , A

*A* ▶ *A*

(*A* → *B*) → *A* ▶ *A*

(*ax*)

(→ *L*)

*, J*

▶ ((*A* → *B*) → *A*) → *A*

*,*(→ *R*)

*J*

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

*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](#_bookmark6). In each rule, the active proposition

(the proposition in the stoup in Girard’s sense [[12](#_bookmark27)]) is boxed. When applying a rule, one has to take into account where the active proposition is.

Rather naturally, the active proposition in the cut-rule is the one elimi- nated 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

*,*

Γ*, A*

*z*

*, ,*

*,*

▶ ∆*,A*

*r*

*,*

(*e*-*ax*)

*,*

*,*

(*r*-*ax*)

Γ ▶ *A ,* ∆ Γ*, B*

*z ,r z,*

Γ*, A* → *B* ▶ ∆

*,*

▶ ∆

*r*

Γ*,A* ▶ *A ,* ∆

*z r*

*, ,*

Γ*,A* ▶ *B ,* ∆

(→ *L*)

*,*

*z r,*

(→ *R*)

*z*

*r*

Γ*,A* ▶ ∆

Γ ▶ *A* → *B ,* ∆

*z r*

Γ ▶ *B,* ∆

*,*

Γ*, A*

*z*

*,*

▶ ∆

*r*

*,*

(*µ*˜)

*,*

Γ ▶ *B*

*z*

*,*

*,* ∆

*r*

(*µ*)

*,*

*, ,*

*z r z r* (*cut*)

Γ ▶ ∆

Γ ▶ *A ,* ∆ Γ*, A* ▶ ∆

Fig. 1. Sequent calculus with active proposition.

there is no active proposition in the consequent of (*cut*). Therefore, this cal- culus 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 (*µ*) 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 (→ *R*) does not fulfill the requirement on the active proposition. One has to hack a little in order to find the proof of Peirce’s law

in this sequent calculus with active proposition (Figure [2](#_bookmark8)). This calculus will be the basis of the Curry Howard correspondence for *λµµ*˜.

# Type assignment systems

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 type system *λµµ* of Curien and Herbelin [[7](#_bookmark21)], 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 *µα.c* : *A* says that *µα.c* takes as parameter

*, ,*

*, ,*

*z r z r* (cut) (A → B) → A, A ▶ B, A

*, ,* (µ)

(A → B) → A, A ▶ B , A

*, z r,* (→ R)

(A → B) → A ▶ A → B , A

(A → B) → A, A ▶ A , B, A (A → B) → A, A, A ▶ A, B

*, ,*

(A → B) → A, A ▶ A

*z*

*r*

*,*

*z r*

(A → B) → A, (A → B) → A

*z*

The proof tree A

*,*

▶ A

*r*

(→ L)

*,*

*,*

(A → B) → A ▶ (A → B) → A , A A

*z r* (cut) (A → B) → A ▶ A

*, ,*(µ)

(A → B) → A ▶ A

*,*

*z*

▶ ((A → B) → A) → A

*z*

*r,*(→ R)

*r*

Fig. 2. Peirce’s law with active formula

an *A*-continuation and returns an answer. If we were to informally denote the space of *A*-continuations as the set (*A* ⇒ ⊥) then this *µα.c* inhabits the set ((*A* ⇒ ⊥) ⇒ ⊥), 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* ∩ *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* ∪ *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 *µ*-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* | *A* → *B* | *A* ∩ *B* | *A* ∪ *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 *α* : *A*; in each case we stipulate that all variables are distinct.

There are three kinds of *typing judgments*:

*, ,*

Γ ▶ *r* : *A ,* ∆

*J*

*, ,*

Γ*, e* : *A* ▶ ∆

*J*

*c* : (Γ ▶ ∆)

where Γ is a caller basis and where ∆ is a callee basis.

We will always consider types to be defined modulo commutativity and associativity for ∩ and ∪.

* 1. *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 intersection ∩ are just the rules for conjunction ∧ (that is, if one erases the terms and looks just at the formulas).

As we know, the difference between *A* ∩ *B* and *A* ∧ *B* is that we require the same term to witness *A*, *B* and *A* ∩ *B*.

We can imagine a type system for Gemini derived using the same principle,

applied to sequent calculus deduction. So the ∩ rules would be based on the shape of the logic rules for ∧:

Γ*,A* ▶ ∆ Γ ▶ ∆*,A* Γ ▶ ∆*,B*

Γ*,A* ∧ *B* ▶ ∆ Γ ▶ ∆*,A* ∧ *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 *λµµ*, we

should write down several different *λµµ* sequents whose “erasure” looks like a given logic inference. For example, consider

˜

Γ*,A* ▶ ∆

Γ*,A* ∧ *B* ▶ ∆

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 *λµµ*˜,

this one logic inference would yield three different typing rules, as follows.

*, ,*

Γ*, e* : *A* ▶ ∆

*, z r,*

Γ*, e* : *A* ∩ *B* ▶ ∆

*z*

*r*

*, ,*

Γ*,x* : *A* ▶ *r* : *T ,* ∆

*z, r,*

Γ*,x* : *A* ∩ *B* ▶ *r* : *T ,* ∆

*z r*

*, ,*

Γ*,x* : *A, e* : *T* ▶ ∆

*z, r,*

Γ*,x* : *A* ∩ *B, e* : *T* ▶ ∆

*z r*

Definition 4.2 [A naive system]

The axioms and rules of this type system are given in Figure [3](#_bookmark9).

Let us mention here that the type system from Figure [3](#_bookmark9) restricted to →

types only is exactly the *λµµ* of Curien and Herbelin [[7](#_bookmark21)]. 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 *λµµ*˜ by the term *λx.µα.*⟨*x* (*λy.µβ.*⟨*y α*⟩) • *α*⟩, since one can prove the following in *λµµ*˜

*, ,*

▶ *λx.µα.*⟨*x * (*λy.µβ.*⟨*y * *α*⟩) • *α*⟩ : ((*A* → *B*) → *A*) → *A*

*J*

Ong and Stewart [[16](#_bookmark30)] give the *λµ* term *λx.µα.*[*α*](*x*(*λy.µβ.*[*α*]*y*)) as the “sim- plest witness” of Peirce’s law. One gets the G-term *λx.µα.*⟨*µγ.*⟨*x * *λy.µβ.*⟨*y * *α*⟩• *γ*⟩  *α*⟩ using the Curien Herbelin translation [[7](#_bookmark21)]. It reduces in one (*µ*)

step to our witness *λx.µα.*⟨*x * (*λy.µβ.*⟨*y * *α*⟩)•*α*⟩. In [[21](#_bookmark35)], J´erˆome Rocheteau

studies in detail this translation and its reciprocal.

˜

Thus, *λµµ* gives a more compact witness for the type ((*A* → *B*) → *A*) → *A*. Actually the image of *λµ*-terms in *λµµ* by the translation are not all the

callers, but only the G-terms given by the grammar:

˜

*rλµ* ::= *x* | *λx.rλµ* | *µα.*⟨*rλµ * *rλµ* • *α*⟩ | *µα.*⟨*rλµ * *α*⟩

˜

Still in *λµµ* one cannot type all normal forms, e.g., the G-term *λx.µα.*⟨*x * *x*•*α*⟩ (seen in Section [2](#_bookmark1)) which is a normal form and corresponds to the lambda term *λx.xx* is not typeable in *λµµ*. This is one of the reason to introduce new

type assignment rules.

˜

As noted above, the schemas ∩*L*−*V ar*, ∩*R*−*V ar*, ∪*L*−*V ar*, and ∪*R*−*V ar* 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 ∆ (here we are temporarily abusing our notational convention that Γ and ∆ are just variable-type bindings).

This is a perfectly sensible typing system. But as Barbanera et al. [[1](#_bookmark15)] have noted, union types are technically difficult, for example Subject Reduction tends to fail. And indeed the system above leads to difficulties in Subject Re- duction (more specifically, it seems difficult to prove the Substitution Lemma).



*,*

Γ*, α* : *A*

*z*

*,*

▶ *α* : *A,* ∆

*r*

(*e*-*ax*)

*,*

*,* (*r*-*ax*)

Γ*,x* : *A* ▶ *x* : *A ,* ∆

*z r*

*, ,*

*, ,*

Γ*, e* : *B* ▶ ∆

*z , r*

*,*

*,*

Γ ▶ *r* : *A ,* ∆

*z , r*

Γ*,x* : *A* ▶ *r* : *B ,* ∆

(→ *L*)

*,*

*z r,*

(→ *R*)

Γ*, r* • *e* : *A* → *B* ▶ ∆

*z*

*r*

Γ ▶ *λx.r* : *A* → *B ,* ∆

*z r*

*c* : (Γ*,x* : *A* ▶ ∆)

*c* : (Γ ▶ *α* : *A,* ∆)

*,*

Γ*,*  *µ*˜*x.c* : *A*

*,*

▶ ∆

*J*

*,*

(*µ*˜)

*,*

*,* (*µ*)

Γ ▶ *µα.c* : *A ,* ∆

*J*

*,*

*,*

Γ ▶ *r* : *A ,* ∆ Γ*, e* : *A*

*z r z*

*,*

▶ ∆

*r*

(*cut*)

⟨*r e*⟩ : (Γ ▶ ∆)

Γ*,x* : *A* ▶ ∆

Γ ▶ *α* : *A,* ∆ Γ ▶ *α* : *B,* ∆

(∩*L*−*V ar* )

Γ*,x* : *A* ∩ *B* ▶ ∆

*, ,*

Γ*, e* : *A* ▶ ∆

*, z r,*

Γ*, e* : *A* ∩ *B* ▶ ∆

*, ,*

Γ ▶ *α* : *A* ∩ *B,* ∆

(∩*R*−*V ar* )

*, ,*

(∩*L*)

Γ ▶ *r* : *A ,* ∆ Γ ▶ *r* : *B ,* ∆

*z ,r z, r*

Γ ▶ *r* : *A* ∩ *B ,* ∆

(∩*R*)

*z*

*r*

*z*

*r*

Γ*,x* : *A* ▶ ∆ Γ*,x* : *B* ▶ ∆

Γ ▶ ∆*,α* : *A*

(∪*L*−*V ar* )

*,*

▶ ∆

(∪*R*−*V ar* )

*, ,*

Γ*,x* : *A* ∪ *B* ▶ ∆

*,*

Γ ▶ ∆*,α* : *A* ∪ *B*

*, ,*

Γ*, e* : *A* ▶ ∆ Γ*, e* : *B*

*z ,r z,*

Γ*, e* : *A* ∪ *B* ▶ ∆

*r*

(∪*L*)

Γ ▶ *r* : *A ,* ∆

*, z r,*

Γ ▶ *r* : *A* ∪ *B ,* ∆

(∪*R*)

*z*

*r*

*z*

*r*

Fig. 3. A naive type system.

In particular, if we want to prove:

*, , , , , ,*

If Γ*,x* : *A* ▶

*r* : *T ,* ∆ and Γ ▶ *s* : *A ,* ∆ then Γ ▶ *r*[*x* ← *s*] : *T ,* ∆

*J J J*

we have a problem when the assumed typing was derived using ∪*L*−*V ar*:

(∪*L*−*V ar* )

*, ,*

Γ*,x* : *A*1 ▶ *r* : *T ,* ∆

*z r*

*, ,*

Γ*,x* : *A*2 ▶ *r* : *T ,* ∆

*,*

*, z r*

Γ*,x* : *A*1 ∪ *A*2 ▶ *r* : *T ,* ∆

*z r*

since knowing

*, ,*

Γ ▶ *s* : *A*1 ∪ *A*2 *,* ∆

*J*

doesn’t allow us to use the induction hypothesis on the given

*, ,*

Γ*,x* : *Ai* ▶ *r* : *T ,* ∆*.*

*J*

* 1. *Second attempt: the “deﬁnite” type system*

The problem with the naive system seems to be with variable-typings of the form *x* : *A* ∪ *B* and *α* : *A* ∩ *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 *µ*, 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* ∪ *B* and *α* : *A* ∩ *B*.

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

Definition 4.3

* + 1. A type *A* is ∩*-deﬁnite* if it is a type variable, an arrow-type or it is *A*1 ∩ *A*2, with each *Ai* a ∩-definite type. A type *A* is ∪*-deﬁnite* if it is a type variable, an arrow-type or it is *A*1 ∪ *A*2, with each *Ai* a ∪-definite type.
    2. A basis Γ is ∩*-deﬁnite*, if in each binding *x* : *A* in Γ, *A* is a ∩-definite type. A typing judgment ∆ is ∪*-deﬁnite* if in each binding *α* : *A* in ∆, *A* is a ∪-definite type.
    3. A typing judgment is *deﬁnite* if its typing bases Γ and ∆ are ∩- and

∪-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](#_bookmark10) is the system obtained from the natural gen- eralization of the sequent calculus by forbidding rules ∩*R*−*V ar* and ∪*L*−*V ar* and insisting that typing judgments be definite.

As noted earlier, the schemas ∩*L*−*V ar* and ∪*R*−*V ar* 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 ∆ (here we are temporarily abusing our notational convention that Γ and ∆ are just variable-type bindings).

Definition 4.4 [A “definite” system]

The axioms and rules of this type system are given in Figure [4](#_bookmark12). In each rule below we assume that the type bases are definite.

* 1. *The type system* M∩∪

It turns out that the presence of rules ∩*L*−*V ar* and ∪*R*−*V ar* complicates reason- ing 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 the system removes these rules completely and replaces them with more flexible axiom schemas.

The system in Definition [4.5](#_bookmark11) is the system obtained from the basic system by replacing the rules ∩*L*−*V ar* and ∪*R*−*V ar* 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 (constrained) version of union types is interesting because it enjoys the Subject Reduction property.

Definition 4.5 [The type system M∩∪]

The axioms and rules of this type system are given in Figure [5](#_bookmark13). In each rule

below we assume that the type bases are definite.

Lemma 4.6 *The rules in Deﬁnitions* [*4.4*](#_bookmark10) *and* [*4.5*](#_bookmark11) *generate the same typing judgments.*



*,*

Γ*, α* : *A*

*z*

*,*

▶ *α* : *A,* ∆

*r*

(*e*-*ax*)

*,*

*,* (*r*-*ax*)

Γ*,x* : *A* ▶ *x* : *A ,* ∆

*z r*

*,*

*,*

*, ,*

Γ*, e* : *B* ▶ ∆

*z , r*

*,*

*,*

Γ ▶ *r* : *A ,* ∆

*z , r*

Γ*,x* : *A* ▶ *r* : *B ,* ∆

(→ *L*)

*,*

*z r,*

(→ *R*)

Γ*, r* • *e* : *A* → *B* ▶ ∆

*z*

*r*

Γ ▶ *λx.r* : *A* → *B ,* ∆

*z r*

*c* : (Γ*,x* : *A* ▶ ∆)

*c* : (Γ ▶ *α* : *A,* ∆)

*,*

Γ*,*  *µ*˜*x.c* : *A*

*,*

▶ ∆

*J*

*,*

(*µ*˜)

*,*

*,* (*µ*)

Γ ▶ *µα.c* : *A ,* ∆

*J*

*,*

*, ,*

Γ ▶ *r* : *A ,* ∆ Γ*, e* : *A* ▶ ∆

*z r z r*

(*cut*)

⟨*r e*⟩ : (Γ ▶ ∆)

Γ*,x* : *A* ▶ ∆

(∩*L*−*V ar* )

Γ*,x* : *A* ∩ *B* ▶ ∆

*, ,*

Γ*, e* : *A* ▶ ∆

*, z r,*

Γ*, e* : *A* ∩ *B* ▶ ∆

*,*

*,*

*,*

*,*

(∩*L*)

Γ ▶ *r* : *A ,* ∆ Γ ▶ *r* : *B ,* ∆

*z ,r z, r*

(∩*R*)

*z r*

Γ ▶ *r* : *A* ∩ *B ,* ∆

*z r*

Γ ▶ ∆*,α* : *A*

(∪*R*−*V ar* )

*, ,*

*,*

Γ ▶ ∆*,α* : *A* ∪ *B*

Γ*, e* : *A* ▶ ∆ Γ*, e* : *B*

*z ,r z,*

Γ*, e* : *A* ∪ *B* ▶ ∆

*,*

▶ ∆

*r*

*, ,*

(∪*L*)

Γ ▶ *r* : *A ,* ∆

*, z r,*

Γ ▶ *r* : *A* ∪ *B ,* ∆

(∪*R*)

*z r z r*

Fig. 4. A “definite” type system.

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 ∩*L*−*V ar* and

∪*R*−*V ar* .

Next we show that rules ∩*L*−*V ar* and ∪*R*−*V ar* can be eliminated in the presence of *e*+-*ax* and *r*+-*ax*. This follows from the fact that any use of ∩*L*−*V ar* or ∪*R*−*V ar* 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.



*,*

Γ*, α* : *Ai*

*,*

*,*

▶ *α* : *A*1 ∪ ··· ∪ *An,* ∆

*J*

(*e*+-*ax*)

*,*

*,* (*r*+-*ax*)

Γ*,x* : *A*1 ∩ ··· ∩ *An* ▶ *x* : *Ai ,* ∆

*J*

*,*

*, ,*

*,*

*,*

Γ ▶ *r* : *A ,* ∆ Γ*, e* : *B* ▶ ∆

*z , r*

*z , r*

Γ*,x* : *A* ▶ *r* : *B ,* ∆

(→ *L*)

*,*

*z r,*

(→ *R*)

Γ*, r* • *e* : *A* → *B* ▶ ∆

*z*

*r*

Γ ▶ *λx.r* : *A* → *B ,* ∆

*z r*

*c* : (Γ*,x* : *A* ▶ ∆)

*c* : (Γ ▶ *α* : *A,* ∆)

*,*

Γ*,*  *µ*˜*x.c* : *A*

*,*

▶ ∆

*J*

*,*

(*µ*˜)

*,*

*,* (*µ*)

Γ ▶ *µα.c* : *A ,* ∆

*J*

*,*

*, ,*

Γ ▶ *r* : *A ,* ∆ Γ*, e* : *A* ▶ ∆

*z r z r*

(*cut*)

⟨*r e*⟩ : (Γ ▶ ∆)

*,*

Γ*, e* : *A*

*, z*

*,*

▶ ∆

*r,*

*,*

*,*

*,*

*,*

(∩*L*)

Γ ▶ *r* : *A ,* ∆ Γ ▶ *r* : *B ,* ∆

*z ,r z, r*

Γ ▶ *r* : *A* ∩ *B ,* ∆

(∩*R*)

*z*

*, ,*

Γ*, e* : *A* ∩ *B* ▶ ∆

*r*

*,*

*z*

*r*

*,*

▶ ∆

*r*

*,*

*,*

Γ*, e* : *A* ▶ ∆ Γ*, e* : *B*

*z ,r z,*

Γ*, e* : *A* ∪ *B* ▶ ∆

(∪*L*)

Γ ▶ *r* : *A ,* ∆

*, z r,*

Γ ▶ *r* : *A* ∪ *B ,* ∆

(∪*R*)

*z r z r*

Fig. 5. The system M∩∪

# Properties of M∩∪

For the rest of this paper, we work only with the type system M∩∪.

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)

*, , , ,*

1. *If* Γ ▶ *r* : *A*1 ∩ *A*2 *,* ∆*, then for i* = 1*,* 2*,* Γ ▶ *r* : *Ai ,* ∆*.*

*, ,J , ,J*

1. *If* Γ*,*

*e* : *A*1 ∪ *A*2 ▶ ∆*, then for i* = 1*,* 2*,* Γ*, e* : *Ai* ▶ ∆*.*

*J J*

Proof. For *,*p art 1, we jus*,*t observe that the only rules that could be used to

derive Γ ▶ *r* : *A*1 ∩ *A*2 *,* ∆ are *r*+-*ax* and ∩*R*. In the latter case the result

*J*

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 ∩.

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 *α* : *A*1 ∩ *A*2 the assumed derivation must look like

*,*

Γ ▶ *µα.*⟨*r * *e*⟩ : *A*1

*,*

*,* ∆ Γ ▶

*, ,*

*µα.*⟨*r * *e*⟩ : *A*2 *,* ∆

*, J , J*

Γ ▶

that is, an instance of ∩*R*.

*µα.*⟨*r * *e*⟩ : *A*1 ∩ *A*2 *,* ∆

*J*

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](#_bookmark14)).

Lemma 5.2 (Context expansion lemma) *Let* Γ ⊆ Γ' *and* ∆ ⊆ ∆'*.*

*, , , ,*

1. *If* Γ ▶ *r* : *A ,* ∆*, then* Γ' ▶ *r* : *A ,* ∆'*.*

*, ,J , ,J*

1. *If* Γ*,*

*e* : *A* ▶ ∆*, then* Γ'*, e* : *A* ▶ ∆'*.*

*J J*

1. *If c* : (Γ ▶ ∆)*, then c* : (Γ' ▶ ∆')*.*

Lemma 5.3 (Context restriction lemma)

*, , , ,*

1. *If* Γ ▶ *r* : *A ,* ∆*, then* Γ T Fvr(*r*) ▶ *r* : *A ,* ∆ T Fve(*r*)*.*

*, ,J , ,J*

1. *If* Γ*,*

*e* : *A* ▶ ∆*, then* Γ T Fvr(*e*)*, e* : *A* ▶ ∆ T Fve(*e*)*.*

*J J*

1. *If c* : (Γ ▶ ∆)*, then c* : (Γ T Fvr(*c*) ▶ ∆ T Fve(*c*))*.*

Lemma 5.4 (Generation lemma)

*, , , ,*

1. *If* Γ ▶ *λx.r* : *i*∈*I Ai* → *Bi ,* ∆*, then* Γ*,x* : *Ai* ▶ *r* : *Bi ,* ∆*.*

*, ,J J*

*,*

*,*

*,*

*,*

1. *If* Γ*, r* • *e* : *i*∈*I Ai* → *Bi* ▶ ∆*, then* Γ ▶ *r* : *Ai ,* ∆ *and* Γ*, e* : *Bi* ▶

∆*. J J J*

*, ,*

1. *If* Γ ▶

*µα.c* :

*i*∈*I Ai*

*,* ∆*, then c* : (Γ ▶ *α* : *Ai,* ∆)*.*

*J*

*, ,*

1. *If* Γ*,*

*µx.c* :

˜

*i*∈*I Ai* ▶ ∆*, then c* : (Γ*,x* : *Ai* ▶ ∆)*.*

*J*

Lemma 5.5 (Substitution)

*, , , ,*

1. *If* Γ*,x* : *B* ▶ *r*' : *A ,* ∆ *and* Γ ▶ *r* : *B ,* ∆*, then*

*, ,J J*

Γ ▶ *r*'[*x* ← *r*] : *A ,* ∆*.*

*, , J , ,*

1. *If* Γ ▶ *r*' : *A ,* ∆*,α* : *B and* Γ*, e* : *B* ▶ ∆*, then*

*, J , J*

Γ ▶ *r*'[*α* ← *e*] : *A ,* ∆*.*

*, ,J , ,*

1. *If* Γ*,x* : *B, e*' : *A* ▶ ∆ *and* Γ ▶ *r* : *B ,* ∆*, then*

*, ,J J*

Γ*, e*'[*x* ← *r*] : *A* ▶ ∆*.*

*, , J , ,*

1. *If* Γ*, e*' : *A* ▶ ∆*,α* : *B and* Γ*, e* : *B* ▶ ∆*, then*

*, J , J*

Γ*, e*'[*α* ← *e*] : *A* ▶ ∆*.*

*J*

Our type system enjoys the Subject Reduction property, for the calculus with unrestricted reduction, that is, even in the presence of the (*µ, µ*) critical pair. As mentioned in the introduction this has been shown in [[1](#_bookmark15)] to be a difficult property to achieve in a system with union types.

˜

Theorem 5.6 (Subject Reduction [[9](#_bookmark22)])

*If c* : (Γ ▶ ∆) *and c* → *c*' *then c*' : (Γ ▶ ∆)*.*

The typeability of all strongly normalizing terms, a unique property of some traditional *λ*-calculus systems with intersection types, holds in the system M∩∪.

Theorem 5.7 (Strong Normalization [[9](#_bookmark22)])

*All strongly normalizing terms of*

Gemini *are typeable in* M∩∪*.*

# Conclusion

We defined three systems of intersection and union types, defined in terms of sequents and discussed some properties. The final system, M∩∪ has the prop- erties that the typable terms are precisely the strongly normalizing terms [[9](#_bookmark22)].

In contrast to systems with union types investigated previously, the type as- signment system M∩∪enjoys the Subject Reduction property for unrestricted

reduction.

*Some directions for future research.*

Intersection types have proven to be an invaluable tool for studying reduc- tion 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 (a` la Church) with intersection types by Dezani et al. [[8](#_bookmark25)], Ronchi Della Rocca and Roversi [[22](#_bookmark36)], Capitani et al. [[4](#_bookmark18)] and recently by Wells and Haack [[26](#_bookmark40)]. This direction of research in the framework of *λµ*-calculus merits attention.

It will be interesting to investigate the relationship between the system presented here and that of Laurent [[15](#_bookmark29)].

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 [[18](#_bookmark32)] and [[3](#_bookmark17)], and if so, we should try to understand the trade-offs.

# Acknowledgement

We are grateful to the referees for valuable suggestions.

# References

1. Barbanera, F., M. Dezani-Ciancaglini and U. de’ Liguoro, *Intersection and union types: syntax and semantics*, Information and Computation 119 (1995), pp. 202–230.
2. Barendregt, H. P., “The Lambda Calculus: its Syntax and Semantics,” North-Holland, Amsterdam, 1984, revised edition.
3. Buneman, P. and B. Pierce, *Union types for semistructured data*, in: *Research Issues* *in Structured and Semistructured Database Programming, 7th International Workshop on Database Programming Languages*, Lecture Notes in Computer Science 1949 (2000), pp. 184– 207.
4. Capitani, B., M. Loreti and B. Venneri, *Hyperformulae, parallel deductions and intersection types*, Electronic Notes in Theoretical Computer Science 50 (2001).
5. Coppo, M. and M. Dezani-Ciancaglini, *A new type-assignment for lambda terms*, Archiv fu¨r Mathematische Logik 19 (1978), pp. 139–156.
6. Coppo, M. and M. Dezani-Ciancaglini, *An extension of the basic functionality theory for the*

*λ-calculus*, Notre Dame Journal of Formal Logic 21 (1980), pp. 685–693.

1. Curien, P.-L. and H. Herbelin, *The duality of computation*, in: *Proceedings of the 5th ACM- SIGPLAN International Conference on Functional Programming (ICFP’00)* (2000).
2. Dezani-Ciancaglini, M., S. Ghilezan and B. Venneri, *The “relevance” of intersection and union types*, Notre Dame Journal of Formal Logic 38 (1997), pp. 246–269.
3. Dougherty, D., S. Ghilezan and P. Lescanne, *Characterizing strong normalization in a language with control operators*, in: D. Warren, editor, *Proceedings of the 6th ACM-SIGPLAN* *International Conference on Principles and Practice of Declarative Programming (PPDP 2004)*, Verona, Italy, 2004, pp. 155–166.
4. Dunfield, J. 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)*, Lecture Notes in Computer Science 2620 (2003), pp. 250–266.
5. Dunfield, J. and F. Pfenning, *Tridirectional typechecking*, in: X. Leroy, editor, *Conference* *Record of the 31st Annual ACM Symposium on Principles Of Programming Languages (POPL’04)* (2004), pp. 281–292.
6. Girard, J.-Y., *A new constructive logic: classical logic*, Mathematical Structures in Computer Science 1 (1991), pp. 255–296.
7. Griffin, T., *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)*, 1990,

pp. 47–58.

1. Herbelin, H., “S´equents qu’on calcule : de l’interpr´etation du calcul des s´equents comme calcul de *λ*-termes et comme calcul de strat´egies gagnantes,” Th`ese d’universit´e, Universit´e Paris 7 (1995).
2. Laurent, O., *On the denotational semantics of the pure lambda-mu calculus*, Manuscript (2004).
3. Ong, C.-H. L. 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)*, 1997, pp. 215–227.
4. Parigot, M., *An algorithmic interpretation of classical natural deduction*, in: *Proceedings of International Conference on Logic Programming and Automated Reasoning, LPAR’92*, Lecture Notes in Computer Science 624 (1992), pp. 190–201.
5. Pierce, B. C., *Programming with intersection types, union types, and polymorphism*, Technical Report CMU-CS-91-106, Carnegie Mellon University (1991).
6. Pottinger, G., *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*, Academic Press, London, 1980 pp. 561–577.

1. Reynolds, J. C., *Design of the programming language Forsythe*, Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania (1996).
2. Rocheteau, J., “Lambda mu calcul et dualit´e : appel par nom et appel par valeur,” Master’s thesis, University of Paris VII (2002), PPS’s preprint: PPS//02/10//n8.
3. Ronchi Della Rocca, S. and L. Roversi, *Intersection logic*, in: *Computer Science Logic, CSL’01*, Lecture Notes in Computer Science 2142 (2001), pp. 421–428.
4. Sall´e, P., *Une extension de la th´eorie des types en lambda-calcul*, in: G. Ausiello and C. Bo¨hm, editors, *Fifth International Conference on Automata, Languages and Programming*, Lecture Notes in Computer Science 62 (1978), pp. 398–410.
5. van Bakel, S., *Intersection type assignment systems*, Theoretical Computer Science 38 (1997),

pp. 246–269.

1. Wells, J. B., A. Dimock, R. Muller and F. Turbak, *A calculus with polymorphic and polyvariant flow types*, Journal of Functional Programming 12 (2002), pp. 183–227.
2. Wells, J. B. and C. Haack, *Branching types*, forthcoming in Information and Computation (200X).