HAL
open science

Towards an Intersection Typed System à la Church

Luigi Liquori, Simona Ronchi Della Rocca

To cite this version:

Luigi Liquori, Simona Ronchi Della Rocca. Towards an Intersection Typed System à la Church. Proceedings of the Third International Workshop on Intersection Types and Related Systems (ITRS 2004) Intersection Types and Related Systems 2004, Jul 2004, Turku, Finland. pp.43-56, 10.1016/j.entcs.2005.06.015 . hal-01149611

HAL Id: hal-01149611

https://inria.hal.science/hal-01149611v1

Submitted on 7 May 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards an Intersection Typed System *à la* Church

Luigi Liquori

INRIA Sophia Antipolis, France

Simona Ronchi della Rocca

CS, University of Turin, Italy

Abstract

In this paper, we presents a comfortable fully typed lambda calculus based on the well-known intersection type system discipline where proof are not only feasible but easy; the present system is the counterpart *à la* Church of the type assignment system as invented by Coppo and Dezani.

1 Introduction

The *Intersection Type Assignment System* ($\Lambda^{\mu}\wedge$) is a set of inference rules for assigning *Intersection types* to terms of the untyped λ-calculus. Intersection types are formulæ of the implicational and conjunctive fragment of the predicate logic. The syntax and the typing rules are presented in Figure 1. Intersection types were introduced by Coppo and Dezani at the end of '70s, to increase the typability power of Curry's type assignment system for the λ-calculus [5]. Since then, intersection types have been fruitfully used for designing static semantics of programming languages (*e.g.* Algol-like [12], and object-oriented [11]), for characterizing interesting classes of λ-terms (*e.g.* the strongly normalizing ones [10]), and for studying denotational semantics of various untyped λ-calculi [1, 2].

There are many versions in the literature of intersection type assignment systems. Here we choses that one presented as "System D" by Krivine [9], characterized by the presence of explicit rules for dealing with the introduction and elimination of the intersection. Note that, differently from the big part of the systems presented in the literature, as for example [6], in this system the connective $\wedge$ is neither commutative nor associative nor idempotent, This last choice has been done since we are looking for a typed version of the calculus, and in this setting it is natural to consider types as syntactical entities. In

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Syntax of $\Lambda^{\mathsf{u}}\wedge$

$$M ::= x \mid \lambda x.M \mid M\,M$$

$$\sigma ::= \alpha \mid \sigma\to\sigma \mid \sigma\wedge\sigma$$

Let α range over a denumerable set $\mathcal{V}$ of constants, and $\Gamma \equiv \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$, where $i \neq j$ implies $x_i \not\equiv x_j$. The operator "$\wedge$" takes precedence over "$\to$".

Typing Rules

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_\wedge x : \sigma}\,(Var) \qquad \frac{\Gamma, x{:}\sigma_1 \vdash_\wedge M : \sigma_2}{\Gamma \vdash_\wedge \lambda x.M : \sigma_1\to\sigma_2}\,(\to I)$$

$$\frac{\Gamma \vdash_\wedge M : \sigma_1\to\sigma_2 \quad \Gamma \vdash_\wedge N : \sigma_1}{\Gamma \vdash_\wedge M\,N : \sigma_2}\,(\to E) \qquad \frac{\Gamma \vdash_\wedge M : \sigma_1 \quad \Gamma \vdash_\wedge M : \sigma_2}{\Gamma \vdash_\wedge M : \sigma_1\wedge\sigma_2}\,(\wedge I)$$

$$\frac{\Gamma \vdash_\wedge M : \sigma_1\wedge\sigma_2}{\Gamma \vdash_\wedge M : \sigma_1}\,(\wedge E_L) \qquad \frac{\Gamma \vdash_\wedge M : \sigma_1\wedge\sigma_2}{\Gamma \vdash_\wedge M : \sigma_2}\,(\wedge E_R)$$

Figure 1. The Intersection Type Assignment System $\Lambda^{\mathsf{u}}\wedge$

any case this choice does not have any consequence on the typability power of the system, which is well known to characterize all and only the strongly normalizing terms [9].

Following a standard terminology, let us call *à la* Curry a system assigning types to untyped terms, and *à la* Church a system assigning types to typed terms, *i.e.* where types are part of the syntax of terms. Differently from other type assignment systems *à la* Curry, $\Lambda^{\mathsf{u}}\wedge$ has no natural counterpart *à la* Church. The classical example is the polymorphic identity in $\Lambda^{\mathsf{u}}\wedge$ that has the following type-derivation:

$$\frac{\dfrac{x{:}\sigma_1 \vdash_\wedge x : \sigma_1}{\vdash_\wedge \lambda x.x : \sigma_1\to\sigma_1}\,(\to I) \qquad \dfrac{x{:}\sigma_2 \vdash_\wedge x : \sigma_2}{\vdash_\wedge \lambda x.x : \sigma_2\to\sigma_2}\,(\to I)}{\vdash_\wedge \lambda x.x : (\sigma_1\to\sigma_1)\wedge(\sigma_2\to\sigma_2)}\,(\wedge I)$$

but is untypable using a naïve corresponding rule *à la* Church for the introduction of intersection types [8].

$$\frac{\dfrac{x{:}\sigma_1 \vdash_\wedge x : \sigma_1}{\vdash_\wedge \lambda x{:}\sigma_1.x : \sigma_1\to\sigma_1}\,(\to I) \qquad \dfrac{x{:}\sigma_2 \vdash_\wedge x : \sigma_2}{\vdash_\wedge \lambda x{:}\sigma_2.x : \sigma_2\to\sigma_2}\,(\to I)}{\vdash_\wedge \lambda x{:}\boxed{\;?\;}.x : (\sigma_1\to\sigma_1)\wedge(\sigma_2\to\sigma_2)}\,(\wedge I)$$

By the Curry-Howard isomorphism, the λ-term must record the shape of its type-derivation. Naïve solutions of λ-calculi *à la* Church lead to incompleteness, in the sense that the resulting typed system has less typability power of the type assignment one (see for example [12]). The problem is, as the skilled reader can understand, the presence of non syntax-directed rules that disconnect the λ-term with its type-derivation (hence losing Curry-Howard correspondence).

It is important to point out that this problem does not depend on the chosen intersection type assignment system; indeed no one of the type assignment systems presented in the literature is completely syntax directed (and not-even it cannot be!) .

Our goal is to build a typed λ-calculus *à la* Church with related typing system, such that the underlying untyped calculus is exactly the λ-calculus *à la* Curry, and its relation with the type assignment system follows the standard path designed in [3]. Namely the following requirements must be satisfied:

(i) there exists an erasing function $\mathcal{J}$, erasing type informations from terms, such that, if M is a typed term, then $\mathcal{J}(M) \in \Lambda$;

(ii) typed and type assignment proofs are *isomorphic*, *i.e.* both the application of the *erasing function* $\mathcal{J}$ on all terms in a typed proofs produces a correct type assignment proof, and every type assignment proof is obtained from a typed one by erasure.

Moreover, we want that the intersection calculus *à la* Church inherits all the properties of intersection calculus *à la* Curry, namely:

(iii) subject reduction;

(iv) strong normalization of typable terms;

plus the following ones:

(v) unicity of typing;

(vi) decidability of type reconstruction and of type checking.

No one of the proposals present in the literature satisfies all the given requirements. The typed languages proposed in [12] and [11] are not complete with respect to the type assignment, the ones in [4], [13] and [16] do not satisfy requirement 1, while the language in [17] does not satisfy requirement 2.

In order to build such a calculus, our attempt has been to design a calculus, where typing depends not only on a new "imperative" formulation of context, now assigning types to term variables *at a given location*, but also on a new notion of *store*, that remembers the associations between locations and types. The further task of a store is to record the shape of a proof. Hence a store is a truly typed *proof calculus* that can be *executed* by means of suitable reduction rules; reduction in this calculus models cut-elimination. The store-calculus can be defined *per se*, as decoration of the implicative and conjunctive fragment of intuitionistic logic. So it codifies a set of proofs that is strictly bigger than

these corresponding to intersection type-derivations (see [15]).

Finally the desired intersection typed calculus can be built, where stores can be seen as *modalities* for terms. Very roughly speaking, a modality is a unary operator that can be used for decorating a logical formula, saying additional informations about its provability. In particular the information carried out by a modality can be a meta-theoretical information about the structure of the proof, as for example the modality ! in Linear Logic [7], that denotes a possible not linear use of a premise. In [14] a typed language has been showed, where a modality on terms is the counterpart of the modality on types, and it is used for characterizing the variables that can occur not linearly. Here the modality is used for describing the structure of the type-derivation. In fact, packing together λ-terms and stores as modalities restrict the stores to describe just the intersection type-derivations. So the typed identity with type $(\sigma_1 \to \sigma_1) \wedge (\sigma_2 \to \sigma_2)$ now is the term: $\lambda x : 0.x$, where 0 is a location, and the modality of this term is: $(\lambda 0{:}\sigma_1.0) \wedge (\lambda 0{:}\sigma_2.0)$. The typed λ-calculus so obtained satisfies all the above requirements. In particular the type reconstruction and type checking are decidable, and the algorithms are pretty easy.

The paper is organized as follows: In Section 2 the calculus of stores is presented, in Section 3 the whole intersection typed λ-calculus is showed, Section 4 lists its properties, and in particular it contain the type checking and the type inference algorithm. The conclusion presents some final considerations.

2 The Typed Proof-calculus

The syntax of intersection types is that of the formulas of the implicative and conjunctive fragment of the intuitionistic logic $(\mathcal{L}\wedge_{\to,\wedge})$, if we think to the connective $\to$ as the implication and to the connective $\wedge$ as the conjunction. However the intersection type assignment system is not corresponding, in the Curry-Howard sense, to this logic, because of the "anomalous" decoration of the rules dealing with the conjunction.

In this section we define a typed calculus, decorating the proof of such a logic. The calculus is a typed λ-calculus, but it is built, instead on variables, on *locations*. In fact it will be used as a (particular notion of) store, for remembering both the association between locations and types and the structure of the type-derivation.

Syntax of $\Lambda\mathcal{P}$.

Definition 2.1

(i) Type-locations (or type-addresses) range over Nat.

(ii) Intersection types are defined as follows:

$$\sigma ::= \alpha \mid \sigma \to \sigma \mid \sigma \wedge \sigma$$

$$\frac{\iota{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathcal{P}} \iota : \sigma}\,(Var) \qquad \frac{\Gamma, \iota{:}\sigma_1 \vdash_{\mathcal{P}} \Delta : \sigma_2}{\Gamma \vdash_{\mathcal{P}} \lambda\iota{:}\sigma_1.\Delta : \sigma_1 \to \sigma_2}\,(\to I)$$

$$\frac{\Gamma \vdash_{\mathcal{P}} \Delta_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash_{\mathcal{P}} \Delta_2 : \sigma_1}{\Gamma \vdash_{\mathcal{P}} \Delta_1\,\Delta_2 : \sigma_2}\,(\to E) \qquad \frac{\Gamma \vdash_{\mathcal{P}} \Delta_1 : \sigma_1 \quad \Gamma \vdash_{\mathcal{P}} \Delta_2 : \sigma_2}{\Gamma \vdash_{\mathcal{P}} \Delta_1 \wedge \Delta_2 : \sigma_1 \wedge \sigma_2}\,(\wedge I)$$

$$\frac{\Gamma \vdash_{\mathcal{P}} \Delta : \sigma_1 \wedge \sigma_2}{\Gamma \vdash_{\mathcal{P}} \swarrow\!\Delta : \sigma_1}\,(\wedge E_L) \qquad \frac{\Gamma \vdash_{\mathcal{P}} \Delta : \sigma_1 \wedge \sigma_2}{\Gamma \vdash_{\mathcal{P}} \Delta\!\searrow : \sigma_2}\,(\wedge E_R)$$

Figure 2. The Proof-calculus $\Lambda\mathcal{P}$.

where α range over a denumerable set $\mathcal{V}$ of constants.

(iii) Contexts are finite associations between locations and types, and are defined by the following grammar:

$$\Gamma ::= \epsilon \mid \Gamma, \iota{:}\sigma$$

(iv) Tree-stores are labeled unary/binary trees defined as follows:

$$\Delta ::= \iota \mid \lambda\iota{:}\sigma.\Delta \mid \Delta\,\Delta \qquad \text{(for syntax directed rules)}$$

$$\Delta \wedge \Delta \mid \swarrow\!\Delta \mid \Delta\!\searrow \qquad \text{(for intersection rules)}$$

(v) The set $\mathsf{Fl}(\Delta)$ of the free locations in a tree-store Δ is an easy adaptation to free-stores of the notion of the set of free variables in a λ-term.

In what follows, the symbol $\equiv$ denotes the syntactic equality for terms, types, contexts, type-locations and tree-stores, respectively.

Type System of $\Lambda\mathcal{P}$.

The system proves judgments of the shape:

$$\Gamma \vdash_{\mathcal{P}} \Delta : \sigma$$

Intuitively: in the judgment, the type-context Γ assigns intersection types to store-locations of Δ; the tree-store keeps track of the type of the used location together with a written trace of the *skeleton* of the typing proof, where the skeleton of a proof is the tree obtained from it by erasing all the informations but the name of the applied rules. So the tree-store Δ plays the role of the "road map" to backtrack (*i.e.* roll back) the derivation tree. The typing rules are presented in Figure 2. Some comments are in order:

- (Var) gives types to free-locations;
- $(\to I)$ the tree-store Δ evolves in a new tree-store enriched with the binding for the location ι;

- $(\to E)$ observes that the two type-stores of the premises become sub tree-stores in the conclusion (the hidden application operator being the root);
- $(\wedge I)$ $(\wedge E_L)$, and $(\wedge E_R)$ are the three standard rules that introduce/eliminate intersection-types; note that in order to keep track of which branch is chosen in the intersection-elimination (and keep unicity of typing), tree-stores are marked with two place-holders $\swarrow$ and $\searrow$ indicating the correct branch to select.

Reduction Semantics of $\Lambda\mathcal{P}$.

Being every term in $\Lambda\mathcal{P}$ a decorations of a proof of $\mathcal{L}\wedge_{\to,\wedge}$, the reduction rules of the languages correspond to the cut-elimination steps of the logic. So the $\to$-cut elimination gives rise to the following reduction rule (similar to the standard β-rule) :

$$(\lambda\iota{:}\sigma.\Delta_1)\,\Delta_2 \to_\iota \Delta_1[\Delta_2/\iota]$$

and the $\wedge$-cut eliminations give rise to the following two reduction rules:

$$\swarrow(\Delta_1 \wedge \Delta_2) \to_{\pi_1} \Delta_1$$

$$(\Delta_1 \wedge \Delta_2)\searrow \to_{\pi_2} \Delta_2$$

The reduction relation $\to_{\iota\pi}$ is the contextual closure of the reduction rules $\to_\iota$, $\to_{\pi_1}$, $\to_{\pi_2}$. In particular, we denote by $\mapsto\!\!\!\!\to_\pi$ the contextual, reflexive and transitive closure of the two rules $\to_{\pi_1}$ and $\to_{\pi_2}$.

We also work modulo α-conversion, which can be rephrased in our $\Lambda\mathcal{P}$ as the symmetric, reflexive, transitive and contextual closure of the following rules:

$$\lambda\iota_1{:}\sigma.\Delta \to_{\alpha_\iota} \lambda\iota_2{:}\sigma.\Delta[\iota_2/\iota_1] \qquad \text{if } \iota_2 \notin \mathsf{Fl}(\Delta)$$

Every type-derivation in $\Lambda^{\mathsf{u}}\wedge$, when terms are erased, corresponds to a proof in $\mathcal{L}\wedge_{\to,\wedge}$, but the vice-versa is not true, since the rule $(\to I)$ corresponds to a meta-theoretical condition on the proofs of the two premises. For a complete analysis of the relation between $\Lambda^{\mathsf{u}}\wedge$ and $\mathcal{L}\wedge_{\to,\wedge}$ see [15]. Here just note that, for example, the skeleton of the proof encoded by the tree-store $(\lambda 0{:}\sigma.0) \wedge 1$ is the following:

$$\dfrac{\dfrac{}{}(Var)}{}(\to I) \qquad \dfrac{}{}(Var)$$
$$\dfrac{\qquad\qquad}{}(\wedge I)$$

which cannot be the skeleton of any intersection type assignment derivation.

Example 2.2 We show a type-derivation for the tree-store $(\lambda 0{:}\sigma_1.0)\wedge(\lambda 0{:}\sigma_2.0)$.

$$\dfrac{\dfrac{\dfrac{}{0{:}\sigma_1 \vdash_{\mathcal{P}} 0 : \sigma_1}(Var)}{\vdash_{\mathcal{P}} \lambda 0{:}\sigma_1.0 : \sigma_1\to\sigma_1}(\to I) \qquad \dfrac{\dfrac{}{0{:}\sigma_2 \vdash_{\mathcal{P}} 0 : \sigma_2}(Var)}{\vdash_{\mathcal{P}} \lambda 0{:}\sigma_2.0 : \sigma_2\to\sigma_2}(\to I)}{\vdash_{\mathcal{P}} (\lambda 0{:}\sigma_1.0) \wedge (\lambda 0{:}\sigma_2.0) : (\sigma_1\to\sigma_1) \wedge (\sigma_2\to\sigma_2)}(\wedge I)$$

3 The Intersection Typed System $\Lambda^{\mathsf{t}}\wedge$

The Intersection Typed System $\Lambda^{\mathsf{t}}\wedge$ is built starting from a particular notion of context. In fact, now a context associates a type to a variable *at a given location*, *i.e.*, it both associates the type to the variable and takes a free location for storing the variable itself. This trick allows for remembering, in the rule $(\rightarrow I)$, just the location, being the corresponding type stored in the tree-store, built by the system in parallel with the typed term. In this way the underlying term is a term of the classical untyped λ-calculus. Moreover, since the tree-store describes the structure of the type-derivation, we obtain the decidability property of typing.

Syntax of $\Lambda^{\mathsf{t}}\wedge$.

Definition 3.1 (i) Type-locations, intersection types and tree-stores are defined as in Definition 2.1.

(ii) Contexts are finite associations between variables and types at a given location, and they are defined as follows:

$$\Gamma ::= \epsilon \mid \Gamma, x@\iota{:}\sigma$$

The set $\Lambda^{\mathsf{t}}\wedge$ of intersection typed terms is defined as follows:

$$M ::= x \mid \lambda x{:}\iota.M \mid M\,M$$

Type System of $\Lambda^{\mathsf{t}}\wedge$.

The intersection typed calculus $\Lambda^{\mathsf{t}}\wedge$ is in some sense a modal calculus, where terms of $\Lambda\mathcal{P}$ represent the modality. The typed system proves statements of the shape:

$$\Gamma \vdash M@\Delta : \sigma$$

where Γ is a context, $M \in \Lambda^{\mathsf{t}}\wedge$, and $\Delta \in \Lambda\mathcal{P}$. Intuitively: in the judgment, the type-context Γ assigns intersection types to the free-variables of M allocated in the free store-locations; the tree-store keeps track of the type of the used location together with a written trace of the *skeleton* of the derivation tree. The tree-store Δ plays the role of the road map to backtrack (*i.e.* roll back) the derivation tree. The typing rules are presented in Figure 3. Some comments are in order:

- (Var) gives types to free-variables at a given location;
- $(\rightarrow I)$ is a quasi-classical abstraction rule, but it records in the term only the type-location associated to the abstracted variable; the tree-store Δ evolves in a new tree-store enriched with the binding for the location ι;
- $(\rightarrow E)$ is a quasi-classical application rule; observe that the two type-stores of the premises become sub tree-stores in the conclusion (the hidden application operator being the root.)

$$\frac{x@\iota{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\,(Var)$$

$$\frac{\Gamma, x@\iota{:}\sigma_1 \vdash M@\Delta : \sigma_2}{\Gamma \vdash (\lambda x{:}\iota.M)@(\lambda\iota{:}\sigma_1.\Delta) : \sigma_1 \to \sigma_2}\,(\to I)$$

$$\frac{\Gamma \vdash M@\Delta_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash N@\Delta_2 : \sigma_1}{\Gamma \vdash (M\,N)@(\Delta_1\,\Delta_2) : \sigma_2}\,(\to E)$$

$$\frac{\Gamma \vdash M@\Delta_1 : \sigma_1 \quad \Gamma \vdash M@\Delta_2 : \sigma_2}{\Gamma \vdash M@(\Delta_1 \wedge \Delta_2) : \sigma_1 \wedge \sigma_2}\,(\wedge I)$$

$$\frac{\Gamma \vdash M@\Delta : \sigma_1 \wedge \sigma_2}{\Gamma \vdash M@(\swarrow\Delta) : \sigma_1}\,(\wedge E_L)$$

$$\frac{\Gamma \vdash M@\Delta : \sigma_1 \wedge \sigma_2}{\Gamma \vdash M@(\Delta\searrow) : \sigma_2}\,(\wedge E_R)$$

Figure 3. The Intersection Typed System $\Lambda^{\mathsf{t}}\wedge$.

- $(\wedge I)$ is the most important rule; given two judgments for M proving type σ_1 and type σ_2, (both proved in the same context Γ but with different tree-stores Δ_1, and Δ_2), we can assign the intersection type $\sigma_1 \wedge \sigma_2$ to M in the context Γ but in the new tree-store $\Delta_1 \wedge \Delta_2$. At this point the λ-term M *loses the one-to-one correspondence with its proof*. Luckily, the new tree-store keeps track of the derivation and guarantees unicity of typing;
- $(\wedge E_L)$, and $(\wedge E_R)$ are the two standard rules that eliminate intersection types. Also in this case the λ-term M *loses the one-to-one correspondence with its (logical) proof*, But the proof is memorized by the tree-store, thanks to the two place-holders $\swarrow$ and $\searrow$, indicating the applied rule.

Reduction Semantics of $\Lambda^{\mathsf{t}}\wedge$.

In order to maintain the correct relation between a term and its modality, the reduction works in parallel both on terms and on tree-stores. Formally, the reduction relation $\to_{\beta\iota}$ of $\Lambda^{\mathsf{t}}\wedge$ is the contextual closure of the following reduction rules:

$$((\lambda x{:}\iota.M)N)@((\lambda\iota{:}\sigma.\Delta_1)\,\Delta_2) \;\to_\beta\; M[N/x]@\Delta_1[\Delta_2/\iota]$$

$$M@\swarrow(\Delta_1 \wedge \Delta_2) \to_{\pi_1} M@\Delta_1$$

$$M@(\Delta_1 \wedge \Delta_2)\searrow \to_{\pi_2} M@\Delta_2$$

We also work modulo α-conversion, which is the symmetric, reflexive, transitive and contextual closure of the following rules:

$$(\lambda x@\iota.M)@\Delta \;\to_\alpha\; (\lambda y{:}\iota.M[y/x])@\Delta \qquad \text{if } \; y \notin \mathsf{Fv}(M)$$

$$M@(\lambda\iota_1{:}\sigma.\Delta) \to_{\alpha_\iota} M[\iota_2/\iota_1]@(\lambda\iota_2{:}\sigma.\Delta[\iota_2/\iota_1]) \qquad \text{if } \iota_2 \notin \mathsf{Fv}(\Delta)$$

Example 3.2 [Classical polymorphic identity] We show a polymorphic type-derivation for the classical polymorphic identity $\lambda x{:}0.x$ in the tree-store

$$(\lambda 0{:}\sigma_1.0) \wedge (\lambda 0{:}\sigma_2.0)$$

$$\dfrac{\dfrac{\dfrac{}{x@0{:}\sigma_1 \vdash x@0 : \sigma_1}(Var)}{\vdash (\lambda x{:}0.x)@(\lambda 0{:}\sigma_1.0) : \sigma_1{\to}\sigma_1}(\to I) \quad \dfrac{\dfrac{}{x@0{:}\sigma_2 \vdash x@0 : \sigma_2}(Var)}{\vdash (\lambda x{:}0.x)@(\lambda 0{:}\sigma_2.0) : \sigma_2{\to}\sigma_2}(\to I)}{\vdash (\lambda x{:}0.x)@((\lambda 0{:}\sigma_1.0) \wedge (\lambda 0{:}\sigma_2.0)) : (\sigma_1{\to}\sigma_1) \wedge (\sigma_2{\to}\sigma_2)}(\wedge I)$$

Example 3.3 [The polymorphic self-application] Let $\Gamma \equiv x{:}0{:}\sigma_2$, with $\sigma_2 \equiv (\sigma_1{\to}\sigma_1) \wedge \sigma_1$. We show a polymorphic type-derivation for the classical self-application $\lambda x{:}0.x\,x$ in the tree-store $\lambda 0{:}\sigma_2.(\swarrow 0)\,(0\searrow)$.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x@0 : \sigma_2}{\Gamma \vdash x{:}(\swarrow 0) : \sigma_1{\to}\sigma_1}(\wedge E_L) \quad \dfrac{}{\Gamma \vdash x@(0\searrow) : \sigma_1}(\wedge E_R)}{\Gamma \vdash (x\,x)@(\swarrow 0)\,(0\searrow) : \sigma_1}(\to E)}{\vdash (\lambda x{:}0.x\,x)@(\lambda 0{:}\sigma_2.(\swarrow 0)\,(0\searrow)) : \sigma_2 \to \sigma_1}(\to I)$$

Note how the tree-store memorizes exactly the skeleton of the type-derivation.

4 Sketch of the Proof-theoretical Development

The system $\Lambda^{\mathsf{t}}\wedge$ enjoys all the properties we asked for. In order to list them, we need to define an erasure function, connecting $\Lambda^{\mathsf{t}}\wedge$ with $\Lambda^{\mathsf{u}}\wedge$.

Definition 4.1 [Erasure]
Let $\mathcal{J} : \Lambda^{\mathsf{t}}\wedge \to \Lambda$ be inductively defined on terms and point-wise extended to contexts and tree-stores as follows:

$$\begin{aligned}
\mathcal{J}(x@_) &\triangleq x \\
\mathcal{J}((\lambda x@\iota.M)@_) &\triangleq \lambda x.\mathcal{J}(M@_) \\
\mathcal{J}((M\,N)@_) &\triangleq \mathcal{J}(M@_)\,\mathcal{J}(N@_) \\
\mathcal{J}(\epsilon) &\triangleq \epsilon \\
\mathcal{J}(\Gamma, x@\iota{:}\sigma) &\triangleq \mathcal{J}(\Gamma), x{:}\sigma
\end{aligned}$$

The notion of skeleton of a proof, defined for the system $\Lambda\mathcal{T}$, can be naturally extended to the system $\Lambda^{\mathsf{t}}\wedge$.

Theorem 4.2 (Galleria) (i) *(Isomorphism)*

(a) *If $\mathcal{D} : \Gamma \vdash M@\Delta : \sigma$, then there exists $\mathcal{D}'$, such that $\mathcal{D}' : \mathcal{J}(\Gamma) \vdash_\wedge \mathcal{J}(M@\Delta) : \sigma$, with $\mathcal{D}$ and $\mathcal{D}'$ having the same skeleton;*

(b) *If $\mathcal{D} : \Gamma \vdash_\wedge M : \sigma$, then there exists $\mathcal{D}'$, M' and Δ, such that $\mathcal{D}' : \Gamma' \vdash M'@\Delta : \sigma$, and $\mathcal{J}(M'@\Delta) = M$, and $\mathcal{J}(\Gamma') = \Gamma$, with $\mathcal{D}$ and $\mathcal{D}'$ having the same skeleton.*

(ii) *(Subject Reduction) If $\Gamma \vdash P@\Delta : \sigma$, and $P@\Delta \to_\beta Q@\Delta'$, then $\Gamma \vdash Q@\Delta'' : \sigma$ with $\Delta' \mapsto\!\!\!\!\rightarrow_\pi \Delta''$.*

(iii) *(Strong Normalization) $\Gamma \vdash M@\Delta : \sigma$ if and only if $M@\Delta$ is strongly normalizing.*

(iv) *(Unicity of Typing) If $\Gamma \vdash P@\Delta : \sigma$, and $\Gamma \vdash P@\Delta : \sigma'$, then $\sigma \equiv \sigma'$.*

(v) *(Type Reconstruction) Given a context Γ, a type-store Δ, and a term M, there is a type σ such that $\Gamma \vdash M@\Delta : \sigma$ if and only if $\mathsf{Type}_\wedge(\Gamma, M@\Delta) = \sigma$.*

(vi) *(Type Checking) Given a context Γ, a tree-store Δ, a term M and a type σ, $\Gamma \vdash M@\Delta : \sigma$ if and only if $\mathsf{Typecheck}_\wedge(\Gamma, M@\Delta, \sigma) = \mathsf{true}$.*

Properties 4, 5 and 6 of this galleria are reached thanks to the notion of tree-store. Figure 4 shows, in ML-style, the type reconstruction and type-checking algorithms.

As far as the properties listed at the points 1,2,3 of the galleria are concerned, the key one is the isomorphism property between the system $\Lambda^{\mathsf{t}}\wedge$ and the system $\Lambda^{\mathsf{u}}\wedge$, which can be easily proved by induction on type-derivations. Both the subject reduction and the strong normalization property are consequences of it.

5 Conclusions

We studied in this paper the problem of designing a Church version of the intersection type assignment system. In particular we asked for a typed language such that its relationship with the intersection type assignment system enjoys all the standard requirements we posed in [3]. Examples of such "good" correspondences are respectively the Church and Curry version of the simple typed λ-calculus, the typed and type assignment version of the second order λ-calculus. We succeed in designing a language satisfying the given requirements, which is based essentially on two tools: an imperative notion of typing, when types are assigned to variables "at a given location", and a typed language, describing intersection type-derivations, whose terms are used as modalities for the terms of the target language.

A reader interested in particular in programming applications could object that the used language is far for being "usable", since the user needs to specify

$$
\begin{array}{llll}
\mathsf{Type}_\wedge(\Gamma, M@\Delta) & \triangleq & \mathsf{match}\ (M@\Delta)\ \mathsf{with} & \\
(_@(\swarrow\Delta_1)) & \Rightarrow & \sigma_1 & \mathsf{if}\ \mathsf{Type}_\wedge(\Gamma, M@\Delta_1) = \sigma_1 \wedge \sigma_2 \\
(_@(\Delta_1\searrow)) & \Rightarrow & \sigma_2 & \mathsf{if}\ \mathsf{Type}_\wedge(\Gamma, M@\Delta_1) = \sigma_1 \wedge \sigma_2 \\
(_@(\Delta_1 \wedge \Delta_2)) & \Rightarrow & \sigma_1 \wedge \sigma_2 & \mathsf{if}\ \mathsf{Type}_\wedge(\Gamma, M@\Delta_1) = \sigma_1\ \mathsf{and} \\
 & & & \quad \mathsf{Type}_\wedge(\Gamma, M@\Delta_2) = \sigma_2 \\
(x@_) & \Rightarrow & \sigma & \mathsf{if}\ x@\iota{:}\sigma \in \Gamma \\
((\lambda x@\iota.M_1)@(\lambda\iota{:}\sigma_1.\Delta_1)) & \Rightarrow & \sigma_1 \to \sigma_2 & \mathsf{if}\ \mathsf{Type}_\wedge((\Gamma, x@\iota{:}\sigma_1), M_1@\Delta_1) = \sigma_2 \\
((M_1\ M_2)@(\Delta_1\ \Delta_2)) & \Rightarrow & \sigma_2 & \mathsf{if}\ \mathsf{Type}_\wedge(\Gamma, M_1@\Delta_1) = \sigma_1 \to \sigma_2\ \mathsf{and} \\
 & & & \quad \mathsf{Type}_\wedge(\Gamma, M_2@\Delta_1) = \sigma_1 \\
(_@_) & \Rightarrow & \mathsf{false} & \mathsf{otherwise} \\
\mathsf{Typecheck}_\wedge(\Gamma, M@\Delta, \sigma) & \triangleq & \mathsf{Type}_\wedge(\Gamma, M@\Delta) = \sigma &
\end{array}
$$

Figure 4. The Type Reconstruction and Type Checking Algorithms for $\Lambda^{\mathsf{t}}\wedge$.

not only the typed terms, but also their modalities, which are codings of type-derivations. The answer can be twofold. From a programming languages point of view, in every typed language the user, in order to write explicitly the type of a term, in some sense needs to "guess" the correct type-derivation assigning that type to the term itself. Here obviously the type-derivations are more difficult than in the simple typed case. But if we think, for example, to the second order typed λ-calculus, in order to write the term

$$\Lambda\beta.\Lambda\gamma.\lambda x{:}\forall\alpha.\alpha.x(\beta \to \gamma) \ \text{ of type } \ \forall\beta.\forall\gamma.(\forall\alpha.\alpha) \to (\beta \to \gamma)$$

it is necessary to know exactly how the rules for introducing and eliminating the universal quantifier work. However, we think that the production of an usable language is not the only justification for the problem we studied. The relationship between typed and type assignment systems is an important theoretical issue, that is interesting in itself.

Acknowledgment. Simona was kindly supported by *QSL: Qualité et Sûreté du Logiciel, CPER, Région Lorraine*, Nancy, and by INRIA; Luigi was visiting

the Department of Informatics, University of Sussex, Brighton; he would like to thank his hosts Matthew Hennessy and Vladimiro Sassone, and the whole Department of Informatics for the ideal working conditions they provided. The two anonymous referees give also useful comments.

References

[1] Barendregt H., Coppo M., and Dezani-Ciancaglini M. "A Filter Lambda Model and the Completeness of Type Assignment", *Journal of Symbolic Logic*, 48(4):931-940, 1983.

[2] Coppo M., Dezani-Ciancaglini M., Honsell F., and Longo G. "Extended Type Structures and Filter Lambda Models", *Logic Colloquium '82*, pp.241-262, 1983.

[3] van Bakel S., Liquori L., Ronchi Della Rocca S., and Urzyczyn P. "Comparing Cubes of Typed and Type Assignment systems", *Annals of Pure and Applied Logic*, 86(3):267–303, 1997.

[4] Capitani B., Venneri B. "Hyperformulæ, Parallel Deductions and Intersection Types", Proc. BOTH 2001, ENTCS, 50(2):180-198, 2001.

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

[6] Dezani-Ciancaglini M., Giovannetti E., de' Liguoro U. "Intersection types, lambda-models and Böhm trees". In MSJ-Memoir Vol. 2 "Theories of Types and Proofs", volume 2, pages 45-97. Mathematical Society of Japan, 1998.

[7] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1-102, 1987.

[8] Hindley J. R. "Coppo Dezani types do not correspond to propositional logic", *Theoretical Computer Science*, 28(1-2):235-236, 1984.

[9] Krivine J.L. "Lambda-calcul, types et modèles", *Masson*, 1990.

[10] Pottinger G. "A type assignment for the strongly normalizable λ- terms", in: *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*, pp.561-577, Academic Press, London, 1980.

[11] Pierce, B., C. and Turner, D., N. "Simple Type-theoretic foundations for object-oriented programming", *Journal of Functional Programming*, 4(2):207–247, 1994.

[12] Reynolds J.C. "Design of the programming language Forsythe", in:*Algol-like Languages*, O'Hearn and Tennent ed.s, Birkhauser, 1996.

[13] Ronchi Della Rocca S. "Intersection Typed Lambda-Calculus", In *Proc of ICTRS*, ENTCS, 70(1), 2002.

[14] S. Ronchi della Rocca and L. Roversi. " Lambda calculus and intuitionistic linear logic". Studia Logica, 59(3), 1997.

[15] Ronchi Della Rocca S. and Roversi L. "Intersection logic", *Proc. of CSL*, LNCS 2142, pp.414-428, Springer-Verlag, 2001.

[16] Wells J.B., Dimock A., Muller R., and Turbak F., "A Calculus with Polymorphic and Polyvariant Flow Types", *Journal of Functional Programming*, 12(3), pp. 183-227, 2002.

[17] Wells J.B., and Haack C. "Branching Types", *Information and Computation*, 200X.