Second Year Report: Syntactic and Semantic properties of useful λ-calculi: Church-Rosser, reducibility, realisability

Supervisors	:	Professor Fairouz Kamareddine
		Doctor Joe B. Wells
Student	:	Vincent Rahli

August 6, 2008

Contents

1 Introduction

In the nineteenth century, due to the lack of precision of natural languages and the apparition of some controversial results in analysis [37], mathematicians and logicians became interested in a more precise formalisation of Mathematics. Frege [66, 37] was the first to set the solid foundations for logic. He, among other things, presented a formalisation of the concept of function. The development of formal systems by Frege and his contemporaries led to the discovery of some paradoxes. The paradox in the work of Frege, found by Russell [60], was due to the problem of self-reflexiveness. This problem is inherent in the fact that any function can be applied to any function (in particular to itself). In order to solve this problem, Russell [60] defined a theory of types where types are used to restrict the application of functions.

One of the great improvement in the movement aiming at the formalisation of Mathematics has been the design of the λ-calculus by Church [10]. He designed a formal system for logic and functions which turned out to be inconsistent. Nevertheless, the subsystem dealing only with functions appeared to a be a successful model for computation. It led to the actual λ-calculus. In the early 1940s, Church added simple typing to the λ-calculus in a system with logical axioms to deal with logic and functions. In this model for computation, functions are studied as computational rules rather than as sets of pairs.

The introduction of type systems in which proofs are introduced as part of the defined theory was also a great improvement. It led to the discovery that types in a type system can be associated to formulae in a logical system and that the proofs of formulae can be associated to typable terms. This association is known as the Curry-Howard or the Curry-De Bruijn-Howard isomorphism. Brouwer, Heyting and Kolmogorov already suggested this isomorphism in their BHK-semantics [1, 64, 65] which interprets formulae of intuitionistic logic by proofs, which are constructive methods based on functions. Kleene [46] also proposed an interpretation, called *realisability*, which stresses the connection between recursive functions and intuitionism.

Many applications have been found to realisability. Initially, realisability has been designed as a method to interpret formulae, i.e. in the semantic domain. This semantics enables to stress the constructivity of systems. Based on realisability, Tait [62] developed a method called *reducibility* to prove properties (for example normalisation properties) of the λ-calculus. Since then, this method has been improved by Girard [26, 27], Koletsos [49] and Gallier [21, 24, 22, 23] amongst others.

So far we have discussed the λ-calculus, type systems and the proof methods known as realisability/reducibility. A system of λ-calculus with types allows different expressive power depending on the interactions that exist between λ-terms and types. Eight representative λ-calculi with types have been generalised in the well known λ-cube of Barendregt [4]. The simplest of the system in the λ-cube is the type system known as the Simply Typed Lambda Calculus [12, 4] first introduced by Church. This λ-cube enables to express different kinds of abstraction, allowing to express among other things, polymorphism. The most popular way to express polymorphism is to use the quantifier $\forall$ as is the case in the system F of Girard. Coppo and Dezani [14] introduced another way to express polymorphism using intersection types. These intersection types are lists of usages. Because of the ramified structure of these types, Coppo, Dezani and Venneri introduced in [15] an operation called expansion in order to restore the principal typing property in such systems (more precisely, in order to be able to calculate any typing from a principal one). Since then, this operation has been extensively improved [9, 7].

During the first two years of the author of this report, three main topics have been studied:

- The λ-calculus, its variants and their properties (e.g., Church Rosser, Strong Normalisation and Standardisation) [42, 44, 45]
- The semantics of intersection type systems with expansion using realisability semantics establishing soundness and completeness [39, 40, 41].
- Type error slicing [31] generating a set of type constraints, enumerating minimal errors and displaying the corresponding error slices [43].

These three themes are a logical follow up of the well-rounded study of a calculus, its semantics/models and its usages/applications. In short, the untyped λ-calculus is a computational model used for the formalisation of the foundations of Mathematics. Based on the untyped λ-calculus (typed version) some type systems were developed to formalise the foundations of Mathematics. Such type systems are syntactical constructs. Developing a semantics for such systems helps in their understanding (such as the Curry-Howard isomorphism). Following these lines we were interested in providing a realisability semantics for an intersection type system with expansion in order to get some light on the expansion concept. Finally, the development and study of a type error slicer such as the one developed by Haack and Wells [31] provides an example of a "real" application of a type system out of the study of the foundations of Mathematics.

In Section 2, we will introduce a few common and general concepts used throughout this report. In Section 3, we introduce the untyped λ-calculus, some of its extensions and properties. We first recall some common definitions on the λ-calculus in Section 3.1. And in Section 3.2, we concentrate on the first theme and in particular, the Church-Rosser property. We recall some of its proofs since its first statement and briefly present our contribution in the domain. In Section 4, we provide an introduction to the work we have carried out in the past two years on the second theme which is finding a semantics of intersection type systems with expansion. In section 5, we present our third and last theme of interest which is type error slicing. The aim of this project is to accurately identify and report the location of a type error of a piece of code (for a SML-based programming language), by providing a set of minimal and necessary collection of points in the piece of code (a slice). In Section 6, we give a short outline of our proposal of work for the third year of the thesis. Finally, we conclude in section 7.

We would like to draw the attention of the reader to the fact that this report is a short summary of what has been achieved during the second year of the author. Due to accuracy, only articles that have been accepted or officially submitted to international journals/conferences are attached to this report. Other work and results have been carried out (for example in [43]), but are not included. The attachments to this report include:

- Article [44] accepted at ITRS'08.
- Article [41] accepted at ITRS'08.
- Article [45] submitted to fundamenta informaticae (short and long versions).
- Article [39] submitted to fundamenta informaticae (short and long versions).
- Article [42] accepted at LSFA'08.
- Article [40] accepted at ICTAC'08.

It is important to emphasise that this report is only a short survey of the above mentioned articles. It should be read with all the above attached articles. These articles are always available on the web page of the author.

2 General background

We let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers over which the metavariables n, m range. We take as convention that if a metavariable v ranges over a set s then the metavariables v_i such that $i \geq 0$ and the metavariables $v', v'', etc.$ also range over s.

A binary relation is a set of pairs. Let rel range over binary relations. If $\langle x, y\rangle \in rel$ then we sometimes write it $x\ rel\ y$. Let $\mathrm{dom}(rel) = \{x \mid \langle x, y\rangle \in rel\}$ and $\mathrm{ran}(rel) = \{y \mid \langle x, y\rangle \in rel\}$. We write rel^* for the reflexive and transitive closure of the relation rel (see the first line of Figure 1). A function is a binary relation fun such that if $\{\langle x, y\rangle, \langle x, z\rangle\} \subseteq fun$ then $y = z$. Let fun range over functions. Let $s \to s' = \{fun \mid \mathrm{dom}(fun) \subseteq s \wedge \mathrm{ran}(fun) \subseteq s'\}$.

Given n sets $s_1, \ldots, s_n$, where $n \geq 2$, $s_1 \times \ldots \times s_n$ stands for the set of all the tuples built on the sets $s_1, \ldots, s_n$. If $x \in s_1 \times \ldots \times s_n$, then $x = \langle x_1, \ldots, x_n\rangle$ such that $x_i \in s_i$ for all $i \in \{1, \ldots, n\}$.

3 The λ-Calculus, its extensions and their properties

For an introduction to the λ-calculus see for example Barendregt [3] or Rosser [59].

Following the work done by Frege, Russell, Curry, etc., Church [10], introduced a system for "the foundation of formal logic". The set of terms of this system was defined as a superset of the so-called λI-calculus. In addition, Church introduced two sets of postulates. The first one called "rules of procedure" which allows one, among other things, to deal with conversion of λ-terms (these rules are presented in Section 3.2.1). The second set contains the "formal postulates" which are logical axioms. However, this system and some of its subsystems turned out to be inconsistent as shown by Kleene and Rosser in [47]. Nevertheless, the subsystem dealing only with functions (in fact, the generalization of which is the λ-calculus) appears to be a "successful model for computable functions" [3]. As stressed in the introduction of Barendregt's book, this theory presents functions as rules, and not as sets of pairs, in order to deal with their computational aspects. This λ-calculus appears to be a generalization of the definition of functions given, for example, by Russell ("propositional functions") as explained in [37].

We recall in a first section some common definitions of the λ-calculus (as defined for example by Barendregt [3]). One important property of the λ-calculus is the Church-Rosser property. In a later section, we present this property and the main lines of some of its proofs. During the second year, one of our interests has been the proofs of some of the properties of the λ-calculus (and mainly the proof of the Church-Rosser property) using a semantic argument (the reducibility method). Hence, we present the main lines of our proofs and restate them in the long story of the proofs of the Church-Rosser property.

3.1 Background on the λ-calculi

The λ-calculus and its variants are defined on set of terms and reduction relations. We first present different set of terms and reduction relations. Then we present different λ-calculi of interest. We finish by presenting different properties of the λ-calculus discussed in this report.

3.1.1 Sets of terms

Let x, y, z range over Var, a countable infinite set of term variables (or just variables). The set of terms of the λ-calculus is defined as follows:

$$M \in \Lambda ::= x \mid (\lambda x.M) \mid (M_1 M_2)$$

We let M, N range over Λ. We assume the usual convention for parenthesis and omit these when no confusion arises. In particular, we write $MM_0 \cdots M_n$ instead of $(\cdots((MM_0)M_1)\cdots M_{n-1})M_n$. We also assume the usual definition of subterms and write $N \subseteq M$ if N is a subterm of M ($M \subseteq M$). We call a term of the form $\lambda x.M$, a λ-abstraction (or just abstraction) and a term of the form M_1M_2 an application.

The α-conversion is the symmetric, reflexive, transitive and compatible (see Figure 1) closure of the following rule:

$$\lambda x.M =_\alpha \lambda y.M[x := y], \text{ where } y \text{ does not occur in } M$$

We take terms modulo α-conversion and use the Barendregt convention (BC) where the names of bound variables differ from the free ones. When two terms M and N are equal (modulo α), we write $M = N$. We write $\mathrm{fv}(M)$ for the set of the free variables of term M.

We define as usual the substitution $M[x := N]$ of N for all free occurrences of x in M. We let $M[x_1 := N_1, \ldots, x_n := N_n]$ be the simultaneous substitution of N_i for all free occurrences of x_i in M for $1 \leq i \leq n$.

Then the set of terms Λ_I ($\subset \Lambda$) is defined as follows: each x is in Λ_I, if $x \in \mathrm{fv}(M)$ and $M \in \Lambda_I$ then $\lambda x.M$ is in Λ_I and if $M_1, M_2 \in \Lambda_I$ then M_1M_2 is in Λ_I.

3.1.2 Reduction relations

β-reduction is the main evaluation process of the λ-calculus. It is defined as the compatible closure (see the last line of Figure 1) of the following rule:

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

βI-reduction is a restriction of the β-reduction defined as the compatible closure of the following rule:

$$(\beta I) : (\lambda x.M)N \rightarrow_{\beta I} M[x := N], \text{ where } x \in \mathrm{fv}(M)$$

η-reduction is defined as the compatible closure of the following rule:

$$(\eta) : \lambda x.Mx \rightarrow_\eta M, \text{ where } x \notin \mathrm{fv}(M)$$

This reduction allows the expression of the concept of extensionality in the λ-calculus (see Barendregt's book [3])

For $r \in \{(\beta), (\beta I), (\eta)\}$, the term on the left-hand-side of the rule r is called a *r-redex* (or just redex when no ambiguity arises) and the one on the right-hand-side is called *r-contractum* (or just contractum when no ambiguity arises). A $\beta\eta$-redex (resp. $\beta\eta$-contractum) is either a β-redex (resp. β-contractum) or an η-redex (resp. η-contractum).

The $\beta\eta$-reduction is defined as: $\rightarrow_\beta \cup \rightarrow_\eta$.

For $r \in \{\beta, \eta, \beta\eta\}$, we define the equivalence relation $=_r$ as the symmetric, reflexive and transitive closure of the following rule:

$$M =_r N \qquad \text{if} \qquad M \rightarrow_r N$$

let $\mathcal{R}$ be a binary relation on Λ.

$$\frac{}{M \mathrel{\mathcal{R}} M}\ (\mathit{refl}) \qquad \frac{M_1 \mathrel{\mathcal{R}} M_2 \quad M_2 \mathrel{\mathcal{R}} M_3}{M_1 \mathrel{\mathcal{R}} M_3}\ (\mathit{tr})$$

$$\frac{P \mathrel{\mathcal{R}} Q}{\lambda x.P \mathrel{\mathcal{R}} \lambda x.Q}\ (\mathit{abs}) \qquad \frac{Q \mathrel{\mathcal{R}} Q'}{PQ \mathrel{\mathcal{R}} PQ'}\ (\mathit{app_1}) \qquad \frac{P \mathrel{\mathcal{R}} P'}{PQ \mathrel{\mathcal{R}} P'Q}\ (\mathit{app_2})$$

Figure 1: Closure rules

3.1.3 λ-calculi and λ-theories

The λ-calculus is defined on the set of terms Λ and the reduction relation $\rightarrow_\beta$. The corresponding theory is called $\boldsymbol{\lambda}$.

The λI-calculus is defined in different ways in the literature. It is defined by Church [10] ("if $\mathbf{x}$ is a variable and $\mathbf{M}$ is well-formed then $\lambda\mathbf{x}[\mathbf{M}]$ is well-formed") on the set of terms Λ and the reduction relation $\rightarrow_{\beta I}$. It is defined by Barendregt [3] on the set of terms Λ_I and the reduction $\rightarrow_{\beta I}$ ("The theory $\boldsymbol{\lambda I}$ ("the $\boldsymbol{\lambda I}$-calculus") consists of equations between λI-terms provable by the axioms and rules of $\boldsymbol{\lambda}$ restricted to Λ_I."). We could also consider the set of terms Λ_I and the reduction $\rightarrow_\beta$. The three corresponding theories are equivalent.

The $\lambda\eta$-calculus is defined on the set of term Λ and the reduction relation $\rightarrow_{\beta\eta}$. The corresponding theory is called $\boldsymbol{\lambda\eta}$. When the $\beta\eta$-reduction is the considered reduction without ambiguity, we sometimes write λ-calculus instead of $\lambda\eta$-calculus.

3.1.4 Residuals, developments and normalisation

A *β-residual* of a β-redex is an occurrence of the propagation of the redex through a β-reduction (it is defined for example by Barendregt [3], Definition 11.2.4). For instance the two occurrences of $(\lambda x.x)y$ in $((\lambda x.x)y)((\lambda x.x)y)$ are residuals of the redex $(\lambda x.x)((\lambda x.x)y)$ in $(\lambda x.xx)((\lambda x.x)((\lambda x.x)y))$ w.r.t. the reduction:
$(\lambda x.xx)((\lambda x.x)((\lambda x.x)y)) \rightarrow_\beta (\lambda x.xx)((\lambda x.x)y) \rightarrow_\beta ((\lambda x.x)y)((\lambda x.x)y)$.

Although, as far as we know the definition of β-residuals is a set concept, it does not seem to be the case for $\beta\eta$-residuals. Different definitions may be found in the literature: the $\beta\eta$-residuals as defined by Curry and Feys [17] or the λ-residuals as defined by Klop [48].

A *development* is the reduction of an initial set of redexes in a term and its residuals w.r.t. the reduction. A development is said to be complete if all the redexes of the initial set of redexes and their residuals have been reduced.

A term is a *normal form* if it cannot be reduced further. We say that a term M is *weakly normalisable* if there exists a reduction from M to a normal form. We say that a term M is *strongly normalisable* if each reduction starting from M ends.

3.2 The Church-Rosser property

The Church-Rosser (or confluence) property is a property satisfied by the λ-calculus stating that if $M_1 =_\beta M_2$ then there exists M_3 such that $M_1 \rightarrow_\beta^* M_3$ and $M_2 \rightarrow_\beta^* M_3$ (this property can be more generally defined in the term rewriting systems setting [6]). (We also say that M_1 satisfies the Church-Rosser property.) It can equivalently be defined as follows: if $M_1 \rightarrow_\beta^* M_2$ and $M_1 \rightarrow_\beta^* M_3$ then there exists M_4 such that $M_2 \rightarrow_\beta^* M_4$ and $M_3 \rightarrow_\beta^* M_4$.

This property is also satisfied when, for example, one considers $\beta\eta$-reduction instead of β-reduction.

This property has first been proved by Church and Rosser [13]. It is among other things used to prove the consistency of the λ-calculus as first proved by Church [11]. This property has been extensively studied in the literature since its first proof. We recall in this section some of its proofs. First, we show how it allows to prove the consistency of the λ-calculus.

3.2.1 Consistency

As far as we can tell, the first one to give a proof of the consistency of the λ-calculus has been Church in 1935 [11]. Church considers the λI-calculus augmented with a special symbol δ which is used in his paper as a conditional (the rule for δ is the same for variables). Church considers a rule for α-conversion, two rules for β-conversion and four rules related to the conditional. These seven rules are stated as follows:

I To replace any part $\lambda x R$ by $\lambda y \mathrm{S}^x_y R$, where y is any variable which does not occur in R.

II To replace any part $\{\lambda x M\}(N)$ of a formula by $\mathrm{S}^x_N M$, provided that the bound variables in M are distinct both from x and from the free variables in N.

III To replace any part $\mathrm{S}^x_N M$ (not immediately following λ) of a formula by $\{\lambda x M\}(N)$, provided that the bound variables in M are distinct both from x and from the free variables in N.

IV To replace any part $\delta(M, N)$ of a formula by $\lambda \mathrm{fx.f(f(x))}$, where M and N are in normal form and contain no free variables and M conv-I N.

V To replace any part $\delta(M, N)$ of a formula by $\lambda \mathrm{fx.f(x)}$, where M and N are in normal form and contain no free variables and it is not true that M conv-I N.

VI To replace any part $\lambda \mathrm{fx.f(f(x))}$ of a formula by $\delta(M, N)$, where M and N are in normal form and contain no free variables and M conv-I N.

VII To replace any part $\lambda \mathrm{fx.f(x)}$ of a formula by $\delta(M, N)$, where M and N are in normal form and contain no free variables and it is not true that M conv-I N.

Where $\mathrm{S}^x_N M$ stands for the result of substituting N for x throughout M.

Then Church gives an encoding of the natural numbers (except 0, because Church considers a variant of the λI-calculus) into the λ-calculus. He chose $\lambda \mathrm{fx.f(x)}$ to stand for 1, $\lambda \mathrm{fx.f(f(x))}$ for 2, etc. In fact, those are defined as abbreviations for the corresponding λ-terms.

The negation is then encoded by the term: $\lambda \mathrm{x}.6 - [\delta(\mathrm{x}, 1) + 2 \times \delta(\mathrm{x}, 1)]$, denoted by $\sim$ and where $-$, $+$, $\times$ are the usual encodings of addition, subtraction and multiplication. He also defines an encoding of conjunction.

Church then proves that "There is no formula P such that both P and $\sim P$ are provable" (Therorem VI).

3.2.2 1936: Church and Rosser [13]

Church and Rosser set out to prove the following result of the λ-calculus (Theorem 1):

$$\text{if } M =_{\beta I\alpha} N \text{ then there exists } P \text{ such that } M \to^*_{\beta I\alpha} P \text{ and } N \to^*_{\beta I\alpha} P$$

where $=_{\beta I\alpha}$ is $=_{\beta I} \cup =_\alpha$ and $M \to_{\beta I\alpha} N$ iff $M =_\alpha M'$, $N =_\alpha N'$ and $M' \to_{\beta I} N'$

The main lines of the proof are as follows:

- The definition of residuals, developments and complete developments.
- The first results proved in [13] are the termination of the developments and of the confluence of the complete developments (Lemma 1). These two results are the basis to prove the Church-Rosser theorem.
- One of the main results that is useful to prove the Church-Rosser theorem is Lemma 2. It states among other things that if B_1 is the result of the reduction of a redex r in A_1 and $A_1 \to_{\beta I\alpha} A_2 \to_{\beta I\alpha} A_3 \to_{\beta I\alpha} \cdots$ and for all k, B_k is the result of a terminating sequence of contractions on the residuals in A_k of r then for all k, $B_k \to^*_{\beta I\alpha} B_{k+1}$.
- The proof of Theorem 1 consists in replacing the reductions $A_1 \to_{\beta I\alpha} \cdots \to_{\beta I\alpha} A_n$ and $A_1 \to_{\beta I\alpha} B$ ("a peak with a single reduction") by the reductions $A_n \to^*_{\beta I\alpha} C$ and $B \to^*_{\beta I\alpha} C$ ("a valley").
- Based on their first theorem, Church and Rosser obtained another important result about normal forms: the uniqueness of the normal forms modulo α-conversion, which is their Corollary 2.
- The last paragraph of [13] is devoted to the untyped λ-calculus (not only λ-calculus). The same results are claimed to be true as well but no proof is given.

3.2.3 1972: Tait and Martin-Löf [55, 3, 63]

The famous method developed by Tait and Martin-Löf is based on the parallel reduction. It is a new reduction relation based on the β-reduction defined as follows:

- $x \Rightarrow_\beta x$
- $\lambda x.M \Rightarrow_\beta \lambda x.M'$ if $M \Rightarrow_\beta M'$
- $MN \Rightarrow_\beta M'N'$ if $M \Rightarrow_\beta M'$ and $N \Rightarrow_\beta N'$
- $(\lambda x.M)N \Rightarrow_\beta M'[x := N']$ is $M \Rightarrow_\beta M'$ and $N \Rightarrow_\beta N'$

This parallel reduction also provides a definition of developments: $M \Rightarrow_\beta M'$ is a development.

The Church-Rosser property is then proved to be satisfied w.r.t. this new reduction. This can be proved by a simple induction on terms or using the complete developments (i.e. a complete parallel reduction: where the last rule of the definition of the parallel reduction is used as much as possible). Finally, by proving the equivalence between $\to^*_\beta$ and the transitive closure of $\Rightarrow_\beta$ they prove that the untyped λ-calculus satisfies the Church-Rosser property (w.r.t. the β-reduction).

3.2.4 1978: Hindley [35]

As far as we know Hindley was one of the first to give the proof of the finiteness of developments w.r.t. $\beta\eta$-reduction (see the introduction in [35]). In [35], Hindley first starts by giving a proof for the β-reduction (not only βI as in [13]). His proof tends to be more precise than the former ones.

At that time, as claimed by Hindley, "all the proofs of the Church-Rosser theorem for λ-calculi, slick or clumsy, turn out to be based on reductions of residuals, and the finiteness property is one of the two main underlying facts which make all such proofs work". It is not the case anymore that the finiteness result is required to prove the Church-Rosser property [25, 50, 42].

In his introduction, Hindley claims that his proof of the finiteness of developments uses the confluence property of the developments when others need the finiteness property to prove confluence. To prove the finiteness result, Hindley provides a method to transform any development of a term into another "equivalent" one such that the length of the latter one provides a bound of the length of the former one.

Though very similar to the proof provided by Church and Rosser, Hindley's proof is much more detailed. For example, the replacement of a sequence of reductions by another one (the "equivalence" of two sequences of reductions) is left unproved by Church and Rosser.

3.2.5 1985: Koletsos [49]

Koletsos proved the Church-Rosser property of the terms typable in the Simply Typed Lambda Calculus using the reducibility method. Koletsos provides an interpretation of types based on a predicate called "monovaluedness". (In this section, we consider $\rightarrow$ and CR as $\rightarrow_\beta$ and the set of terms satisfying the Church-Rosser property but on the set of typed terms of the Simply Typed Lambda Calculus recalled below.)

As usual [4], the set of types is defined as follows (with the constant $\mathit{0}$ as ground type instead of a set of type variables): $\sigma, \tau, \rho \in \mathsf{Ty} ::= \mathit{0} \mid \sigma \rightarrow \tau$.

Koletsos's definition of typable terms does not clearly match the usual one [4], because, for example, sometimes variables are considered as objects of the form x^σ, sometimes as objects of the form x, where x is not explicitly defined (usually, x is defined as a metavariable ranging over a countably infinite set of variables).

However, we can easily understand that Koletsos meant to define the set of typed terms of the Simply Typed Lambda Calculus, as defined below for example. Let us recall that x is a metavariable ranging over the countably infinite set of term variables Var. Then, let x^σ be a term of type σ, if a is a term of type τ then let $(\lambda x^\sigma.a)$ be a term of type $\sigma \rightarrow \tau$, and if a is a term of type $\sigma \rightarrow \tau$ and b is a term of type σ then let (ab) be a term of type τ (note that if $\sigma \neq \tau$ then x^σ and x^τ are two different terms).

For each type ρ and term a of type ρ, the monovaluedness predicate is defined by induction on ρ as follows: $\mathrm{MON}^{\mathit{0}}(a)$ iff a is in CR, and $\mathrm{MON}^{\sigma\rightarrow\tau}(a)$ iff a is in CR and for every term b of type σ, $\mathrm{MON}^\sigma(b)$ implies $\mathrm{MON}^\tau(a(b))$.

Koletsos's method is equivalent to the one consisting in defining a type interpretation as a function which associates to each type σ a set of terms $[\![\sigma]\!]$, such that $\mathrm{MON}^\sigma(a)$ iff $a \in [\![\sigma]\!]$, as is done in many other works following Koletsos's [50, 42].

Then, Koletsos proves two important results:

- If $a \in \mathsf{CR}$ and for each $\lambda x^\sigma.b$ such that $a \rightarrow^* \lambda x^\sigma.b$, $\mathrm{MON}^\rho(\lambda x^\sigma.b)$ then $\mathrm{MON}^\rho(a)$. (This result allows to prove among other things that for each x, $\mathrm{MON}^\sigma(x^\sigma)$.)
- If a is a term of type σ and for every term b, $\mathrm{MON}^\tau(b)$ implies $\mathrm{MON}^\sigma(a_{x^\tau}[b])$ then $\mathrm{MON}^{\tau\rightarrow\sigma}(\lambda x^\tau.a)$, where $a_{x^\tau}[b]$ is defined as the replacing of all the free occurrences of x^τ in a by b. (This result proves the saturation [52] of the type interpretation based on the monovaluedness predicate.)

Finally, using these results, Koletsos obtains the confluence of the set of terms typable in the Simply Typed Lambda Calculus by a simple induction on the structure of a term.

3.2.6 1988: Shankar [61]

This is a notable paper because of the formalisation and proof of the Church-Rosser property in the Boyer-Moore theorem prover (based on a first order, quantifier free logic of recursive functions). Shankar's proof is similar to Tait and Martin-Löf's one. In order not to have to deal with α-conversion, the proof is done using the de Bruijn [19] notation for λ-calculus (as is usually the case when using a theorem prover). The proof is then carried out into the usual notation. Using the Boyer-Moore theorem prover some of the proofs were proved automatically.

3.2.7 1989: Takahashi [63]

Takahashi's method is based on Tait and Martin-Löf's parallel method. He proves that the method extends easily to the $\beta\eta$-case. Even if different from the developments defined for example by Curry and Feys [17][1], Takahashi's method (as for Tait and Martin-Löf's method) consists in defining a new parallel reduction (non overlapping reductions) which allows to develop a term without defining residuals. The usual $\beta\eta$-reduction is then trivially proved to be the transitive closure of the parallel $\beta\eta$-reduction. Then, proving the Church-Rosser property of the untyped λ-calculus w.r.t. the parallel $\beta\eta$-reduction enables to prove the Church-Rosser property of the untyped λ-calculus w.r.t. $\beta\eta$-reduction. The Church-Rosser property of the untyped λ-calculus w.r.t. the parallel $\beta\eta$-reduction is obtained using complete developments (or complete parallel $\beta\eta$-reduction, i.e. all the parallel redexes are reduced): if M reduces to N by a parallel $\beta\eta$-reduction then N reduces to P where P is the unique term (modulo α-conversion) obtained from M by a complete parallel $\beta\eta$-reduction.

3.2.8 2001: Ghilezan and Kunčak [25]

Ghilezan and Kunčak's proof can be depicted by the diagram in Figure 2. This method is well explained by Ghilezan and Kunčak [25] and Kamareddine and Rahli [42]. The method consists of the following steps:

- The formalisation of a development: $\rightarrow_I$ (I in Figure 2). A development is defined as follows: all the redexes in a terms are blocked using two "differentiable" term variables; some of the blocked redexes are unblocked; some of these unblocked redexes are reduced; all the redexes are unblocked (removal of the "differentiable" variables).
- The proof of the confluence of the developments using a simple embedding of the developments into the Simply Typed Lambda Calculus (The blocked terms are proved to be typable in the Simply Typed Lambda Calculus). The confluence of the typable terms in the Simply Typed Lambda Calculus is a well know result (see for example Koletsos's proof recalled Section 3.2.5) and provides the confluence of the developments.
- As usual, β-reduction is proved to be the transitive closure of developments. This provides the confluence of the untyped λ-calculus.

This method provides an embedding of developments into the well known Simply Typed Lambda Calculus for which many properties have already been proved (such as confluence or strong normalisation). The defined developments can easily be

[1] For example, if $x \notin \text{fv}(\lambda y.M)$ then $\lambda x.(\lambda y.M)x$ reduces by the parallel $\beta\eta$-reduction to $\lambda y.M$ by reducing the η-redex $\lambda x.(\lambda y.M)x$. Hence, $(\lambda x.(\lambda y.M)x)N$ reduces by the parallel $\beta\eta$-reduction to $M[y := N]$. There is no corresponding development as defined by Curry and Feys, because $(\lambda y.M)N$ is not a residual of $(\lambda x.(\lambda y.M)x)N$ after reduction of the η-redex $\lambda x.(\lambda y.M)x$.

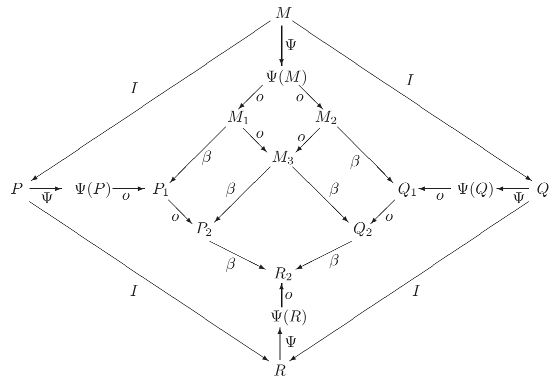

Figure 2: The method of Ghilezan and Kunčak for the confluence of $\rightarrow_I$

proved to be equivalent to the usual ones [3]. The advantages of this method over the similar method of Barendregt [3] (Section 11.2) using a labelled calculus is that it does not make use of the finiteness of developments, does not introduce new symbols (the labelled λ) and is based on an already well known background (the Simply Typed Lambda Calculus). We do not recall Barendregt's proof [3] (Section 11.2) of the confluence of his untyped λ-calculus using a labelled calculus because even if the proof is older than that of Ghilezan and Kunčak, the two proofs share the same steps (proof schemes). Moreover, we concentrate on Ghilezan and Kunčak's proof and not on Barendregt's proof (see [42]). Let us just recall that Barendregt's proof is based on the definition of a new calculus where a new labelled λ is introduced. To prove the confluence of developments, the redexes where the head λ (the head λ of $(\lambda x.M)N$ is the λ displayed on the left of the term) is labelled are the only ones allowed to be reduced.

3.2.9 2007: Koletsos and Stavrinos [50]

Koletsos and Stavrinos's proof is similar to Ghilezan and Kunčak's proof. They share the same proof scheme. However, their result is based on the embedding of the developments into Krivine's type system $\mathcal{D}$ [52] instead of the Simply Typed Lambda Calculus. Their formalisation of developments is more complicated (and sophisticated) than that of Ghilezan and Kunčak in the sense that they handle occurrences explicitly (even if not fully formalised) when Ghilezan and Kunčak handle them implicitly (without explicitly naming them). But their definition of developments is also more simple than that of Ghilezan and Kunčak in the sense that the calculus on which developments are based, is simpler: Koletsos and Stavrinos use one term variable to block the redexes when Ghilezan and Kunčak use two.

3.2.10 2007: Kamareddine, Rahli and Wells [44, 45]

In [44, 45], among other things, we adapt, extend and formalise the work done by Koletsos and Stavrinos. We adapt it to the case of the λI-calculus and extend it to the case of the $\lambda\eta$-calculus, using a formal definition of occurrences of redexes (we do not deal with them intuitively as Koletsos and Stavrinos do [50]). In this work we tried to use a definition of developments based on residuals which are as much close as possible to Klop's λ-residuals. We failed in formalising the concept of λ-residuals as defined by Klop and came up with a new definition that we believe can be regarded as less restrictive than the "common" one [17] and more restrictive than Klop's one.

3.2.11 2008: Kamareddine and Rahli [42]

The aim of this article was first to simplify a previous work based on Koletsos and Stavrinos's work [50] by using the Simply Typed Lambda Calculus (instead of the much more complicated intersection type system $\mathcal{D}$). We came up with a method very similar to the method designed by Ghilezan and Kunčak [25]. The observation that not all the types of the Simply Typed Lambda Calculus where needed in the method led us to the complete removal of the type system from the method. The side effect of the obtained method is that it is not based anymore on the well known framework of the Simply Typed Lambda Calculus. But since the power of this framework appeared to be not needed, the advantage is that we removed from the method the burden of the syntax coming along with the definition of the Simply Typed Lambda Calculus. The obtained method can then be compared to Barendregt's method [3] (Section 11.2). However, we believe our proof to be simpler for the same reasons that Ghilezan and Kunčak's method is simpler than Barendregt's one (listed in Section 3.2.8).

3.2.12 Summary of the proof methods of the Church-Rosser property

In the literature, most of the proof methods to establish the Church-Rosser property of the λ-calculus or its variants use the following scheme already detailed in the previous sections:

- Definition of the developments.
- Proof of the confluence of the developments.
- Proof of the confluence of the considered calculus (using the fact that the transitive closure of a confluent reduction relation is confluent).

The simplest method is the one designed by Tait and Martin-Löf (see Section 3.2.3). This proof is based on a new reduction called parallel reduction. The only disadvantage that we can observe in this method is that the concept of residuals is not as clear as with, for example, our formalisation of developments [42]. However, as far as we know this concept of residuals is interesting only in the context of proving the Church-Rosser property of λ-calculi.

The main point in the proof of the confluence of the λ-calculus (or one of its variants) is the proof of the confluence of the developments. Earlier works [25, 50] proved interesting embedding of developments into well known frameworks such as the Simply Typed Lambda Calculus from which, one can extract useful properties (such as the Church-Rosser property). During this second year we studied the connections between these different proofs and how they can be extended to handle η-reduction [44, 45, 42].

4 Semantics of intersection typed λ-calculi with expansion

In this section we merely recall the introduction of one of our paper [39] which is the recollection of the work we have done on the subject so far [41, 40, 39].

Intersection types were developed in the late 1970s to type λ-terms that are untypable with simple types; they do this by providing a kind of finitary type polymorphism where the usage of types is listed rather than quantified over. They have been useful in reasoning about the semantics of the λ-calculus, and have been investigated for use in static program analysis. *Expansion* was introduced at the end of the 1970s as a crucial procedure for calculating *principal typings* for λ-terms in type systems with intersection types, enabling support for compositional type inference. Coppo, Dezani, and Venneri [15] introduced the operation of *expansion* on *typings* (pairs of a type environment and a result type) for calculating the possible typings of a term when using intersection types. As a simple example, the λ-term $M = (\lambda x.x(\lambda y.yz))$ can be assigned the typing $\Phi_1 = \langle (z : a) \vdash (((a \to b) \to b) \to c) \to c \rangle$, which happens to be its principal typing. The term M can also be assigned the typing $\Phi_2 = \langle (z : a_1 \sqcap a_2) \vdash (((a_1 \to b_1) \to b_1) \sqcap ((a_2 \to b_2) \to b_2) \to c) \to c \rangle$, and an expansion operation can obtain Φ_2 from Φ_1.

Because the early definitions of expansion were complicated, *E-variables* were introduced in order to make the calculations easier to mechanise and reason about. For example, in System E [8], the typing Φ_1 from above is replaced by $\Phi_3 = \langle (z : ea) \vdash (e((a \to b) \to b) \to c) \to c \rangle$, which differs from Φ_1 by the insertion of the E-variable e at two places, and Φ_2 can be obtained from Φ_3 by substituting for e the *expansion term* $E = (a := a_1, b := b_1) \sqcap (a := a_2, b := b_2)$. Carlier and Wells [9] have surveyed the history of expansion and also E-variables.

In many kinds of semantics, the meaning of a type T is calculated by an expression $[T]_\nu$ that takes two parameters, the type T and also a valuation ν that assigns to type variables the same kind of meanings that are assigned to types. To extend this idea to types with E-variables, we would need to devise some space of possible meanings for E-variables. Given that a type $e\,T$ can be turned by expansion into a new type $S_1(T) \sqcap S_2(T)$, where S_1 and S_2 are arbitrary substitutions (they can be arbitrary further expansions), and that this can introduce an unbounded number of new variables (both E-variables and regular type variables), the situation is complicated. Because it is unclear how to devise a space of meanings for expansions and E-variables, we instead develop a space of meanings for types that is hierarchical in the sense of having many degrees. We specifically avoid trying to give a semantics to the operation of expansion, and instead treat only the E-variables. Although this idea is not perfect, it seems to go quite far in giving an intuition for E-variables, namely that each E-variable acts as a kind of capsule that isolates parts of the λ-term being analysed by the typing.

In the open problems published in the proceedings of the Lecture Notes in Computer Science symposium held in 1975 [28], it is suggested that an arrow type expresses functionality. Following this idea, a type's semantics is given as a set of closed λ-terms with behaviour related to the specification given by the type. Hence, the semantic approach we use is realisability semantics. Atomic types (e.g., type variables) are interpreted as sets of λ-terms that are *saturated*, meaning that they are closed under β-expansion (i.e., β-reduction in reverse). Arrow and intersection types are interpreted naturally by function spaces and set intersection. Realisability allows showing *soundness* in the sense that the meaning of a type T contains all closed λ-terms that can be assigned T as their result type. This has been shown useful for characterising the behaviour of typed λ-terms [51]. One also wants to show the converse of soundness which is called *completeness*, i.e., that every closed λ-term in the meaning of T can be assigned T as its result type.

Hindley [36, 33, 34] was the first to study this notion of completeness for a simple type system and he showed that all the types of that system have the completeness property. Then, he generalised his completeness proof for an intersection type system [32]. Using his completeness theorem for the realisability semantics based on the sets of λ-terms saturated by $\beta\eta$-equivalence, Hindley has shown that simple types are uniquely realised by the λ-terms which are typable by these types. However, Hindley's result does not hold for his intersection type system and the completeness theorems were established with the sets of λ-terms saturated by $\beta\eta$-equivalence. In our paper [39], our completeness result depends only on the weaker requirement of β-equivalence, and we have managed to make simpler proofs that avoid needing η-reduction, Church-Rosser (a.k.a. confluence), or strong normalisation (SN) (although we do establish both confluence and SN for both β and $\beta\eta$).

Other work on realisability we have consulted includes that by Labib-Sami [53], Farkh and Nour [20], and Coquand [16], although none of this work deals with intersection types or E-variables. Related work on realisability that deals with intersection types includes that by Kamareddine and Nour [38], which gives a realisability semantics with soundness and completeness for an intersection type system. This system is quite different from the three hierarchical systems we present in this our paper [39]. The main difference being the hierarchies which did not exist in [38].

Initially, we aimed to give a realisability semantics for the system of expansions proposed by Carlier and Wells in [9]. In order to simplify our study, we considered the system with the expansion variables but without the expansion rewriting rules. In essence, this meant that the syntax of terms is: $M ::= x \mid (M\,N) \mid (\lambda x.M)$ where x ranges over a countably infinite set of variables $\mathcal{V}$, that the syntax of types is: $T ::= a \mid \omega \mid T_1 \to T_2 \mid T_1 \sqcap T_2 \mid eT$ where a is a basic type ranging over a countably infinite set of type variables $\mathcal{A}$ and e is an expansion variable ranging over a countably infinite set of expansion variables $\mathcal{E}$, and that the typing rules are:

$$\frac{}{x : \langle (x:T) \vdash T\rangle}\ \text{var}$$

$$\frac{}{M : \langle () \vdash \omega\rangle}\ \omega$$

$$\frac{M : \langle \Gamma, (x:T_1) \vdash T_2\rangle}{\lambda x.M : \langle \Gamma \vdash T_1 \to T_2\rangle}\ \text{abs}$$

$$\frac{M_1 : \langle \Gamma_1 \vdash T_1 \to T_2\rangle \quad M_2 : \langle \Gamma_2 \vdash T_1\rangle}{M_1\ M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash T_2\rangle}\ \text{app}$$

$$\frac{M : \langle \Gamma_1 \vdash T_1\rangle \quad M : \langle \Gamma_2 \vdash T_2\rangle}{M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash T_1 \sqcap T_2\rangle}\ \sqcap$$

$$\frac{M : \langle \Gamma \vdash T\rangle}{M : \langle e\Gamma \vdash eT\rangle}\ \text{e-app}$$

In order to give a realisability semantics for this system, we needed to define the interpretation of a type to be a set of terms having this type. We were obviously forced to distinguish between the interpretation of T and eT. However, in the typing rule e-app, the term M is unchanged and this poses difficulties. For this reason, we modified slightly the above type system by indexing the terms of the λ-calculus giving us the syntax of terms as: $M ::= x^i \mid (M\,N) \mid (\lambda x^i.M)$ (where i are natural numbers and where M and N need to satisfy a certain condition before $(M\,N)$ is allowed as a term) and by slightly changing our type rules and in particular the rule e-app:

$$\frac{M : \langle \Gamma \vdash_i U\rangle}{M^+ : \langle e\Gamma \vdash_i eU\rangle}\ (exp)$$

In this rule, M^+ is M where all the indices are increased by 1. Obviously these indices needed a revision of the β-reduction and of the typing rules in order to preserve the desirable properties of the type system and the realisability semantics. For this, we defined the good terms and the good types and showed that these notions go hand in hand (e.g., a good type contains only good terms). We developed a realisability semantics where each use of an E-variable in a type corresponds to an index at which evaluation occurs in the λ-term that is assigned the type. This is an elegant solution that captures the intuition behind E-variables. However, in order for this new type system to function well, it was necessary to consider λI-terms only (removing a subterm from M also removes important information about M) and to drop ω completely. This led us to the introduction of $\lambda I^{\mathbb{N}}$-calculus and our first type system $\vdash_1$ for which we developed a sound realisability semantics for E-variables. However, although the first type system $\vdash_1$ is crucial to understand the intuition behind the indexing we propose, the realisability semantics for $\vdash_1$ does not satisfy completeness (and neither subject reduction). For this reason, we modified our system $\vdash_1$ by considering a smaller set of types (where intersections and expansions cannot occur directly to the right of an arrow), and by adding subtyping rules. This new system $\vdash_2$ has both soundness and subject reduction. As for completeness, we needed to limit the list of expansion variables to a single element list. This problem of completeness for $\vdash_2$ comes from the fact that the indexes (the natural numbers) do not permit us to differentiate between the types e_1T and e_2T for two different expansion variables e_1 and e_2. So, again, we were forced to revise our type system. For this, we decided to limit our λ-terms by indexing them by lists of natural numbers (where the natural number i represents the expansion variable e_i). This way the rule exp above will allow us to distinguish the interpretations of the types e_iT and e_jT when $e_i \neq e_j$. Furthermore, this way, our λ-terms are constructed in such a way that K-reductions do not limit the information on the starting terms (in fact, β-reduction is not always allowed). In order to obtain completeness with the ω-rule, we should also consider ω indexed by lists. This means that the new calculus becomes rather heavy but this is unavoidable. It is needed to obtain a complete realisability semantics where an arbitrary (possibly infinite) number of expansion variables is allowed and where the universal type ω is present. The use of lists complicates matters and hence, needs to be understood in the context of the first semantics where indices are natural numbers rather than lists of natural numbers. In addition to the above, we have considered three notions of saturations (in line with the literature) illustrating that these notions behave well in our complete realisability semantics.

The paper [39] is articulated as follows:

- We give the syntax of the indexed calculi we consider in our paper [39]: the $\lambda I^{\mathbb{N}}$-calculus, which is the λI-calculus with each variable marked by a natural number *degree*, and the full λ-calculus $\lambda^{\mathcal{L}_{\mathbb{N}}}$-calculus indexed with finite sequences of natural numbers. We show the confluence of β, $\beta\eta$ and weak head reduction h on our indexed λ-calculi.

- We introduce the syntax and terminology for types used in both indexed calculi.

- We introduce our three intersection type systems with E-variables $\vdash_i$ for $i \in \{1, 2, 3\}$, where in one, the syntax of types is not restricted (and hence subject reduction fails) but in the other two it is restricted but then extended with a subtyping relation.

- We study the type theoretical properties of our three type systems including subject reduction and expansion with respect to our various reduction relations $(\beta, \beta\eta, h)$.

- We introduce our realisability semantics and show its soundness for all the three type systems we consider (and for all the reduction relations).

- We establish the challenges of showing completeness for the realisability semantics of the first two systems. We show that completeness does not hold for the first system and that it also does not hold for the second system if more than one expansion variable is used, but does hold for a restriction of this system to one single E-variable. This is an important study in the semantics of intersection type systems with expansion variables since a unique expansion variable can be used many times and can occur nested.

- We establish the completeness of $\vdash_3$ by introducing a special interpretation and we conclude.

5 Type error slicing

In this section we explain our efforts so far at an in-depth formalisation of an existing implementation of a type error slicer by Haack and Wells [31] and at how far we have reached in studying its theoretical properties. This in-depth study prepares the stage for the design of a type error slicer for a rich programming language.

5.1 introduction

SML[2] is a higher-order function-oriented imperative programming language. One of the features of SML (like its predecessor ML) is that it has polymorphic types which permit considerable flexibility.

A language can be defined at different level: syntactic and semantic (static and dynamic). At the syntactic level, the syntax used by the language is defined. The syntax of a language is often given by a set of grammatical rules. At the semantic level, the meaning (static/abstract and dynamic) of the syntactic forms is given. A particular static meaning can be denoted by a type. The static meanings of the syntactic forms of a language are often given by a set of type inference rules. When writing a piece of code in such a programming language, a first step consists in checking that it is written in the syntax of the language. A second one consists often in checking that this piece of code possesses a static meaning w.r.t. the language. This is achieved by checking if the piece of code possesses a type w.r.t. the type inference rules.

In the literature, for some languages, many type inference algorithms may be found. Given an expression and sometimes some other parameters, these algorithms output among other things a type of the expression w.r.t. a given type system and a type environment. When the expression is not typable in the given type system, these algorithms may reject the expression and output an error message. These algorithms have sometimes as a secondary goal the efficiency and/or the accuracy of the location of these type errors. The concern of this section is on the latter point.

The algorithm $\mathcal{W}$ of Damas and Milner [18] is the original type-checking algorithm of the purely applicative part (variables, abstractions, applications and polymorphic let expressions) of the ML language. From a type environment and an expression, $\mathcal{W}$ outputs a type of the expression and a substitution (such that the outputted type is the principal type of the expression w.r.t. the application of the substitution to the type environment). Since then many algorithms have been developed, intending sometimes to give better locations for errors. As an example, the folklore-algorithm $\mathcal{M}$ [54] (the algorithm carries some constraints on the type of the expression down to the expression) is a variant of the $\mathcal{W}$ algorithm (the type is built bottom up). As for $\mathcal{W}$, $\mathcal{M}$ is proved sound and complete. Moreover Lee and

[2] ML is a higher-order functional programming language designed, as part of a proof system called LCF (Logic for Computable Functions), to perform proofs of facts within PPλ (Polymorphic Predicate λ-calculus), a formal logical system [29, 30]. As explained by Milner et al., Standard ML (SML)[57, 58] is the result of the re-design and extension of ML.

Yi proved that this algorithm finds errors "earlier" (this measure is based on the count of recursive calls of the algorithm) than $\mathcal{W}$ and claimed that the combination of these two algorithms "can generate strictly-more informative type-error messages than either of the two algorithms alone can". Notice that these two algorithms do not necessarily report the same error locations for the same piece of code. Variants of these algorithms are $\mathcal{W}'$ [56] or $\mathcal{U}$AE [68]. McAdam claims that $\mathcal{W}$ suffers a left-to-right bias and proposes to eliminate the bias by the replacement of the unification algorithm used in the application case of $\mathcal{W}$ by another one called "unification of substitutions". Yang claims that the primary advantage of $\mathcal{U}$AE is that it also eliminates the left-to-right bias. As explained by McAdam the left-to-right bias in $\mathcal{W}$ arises because in the case of an application, the substitution computed for the first component of the application is applied to the type environment when type-checking the second component.

As explained by Yang et al. [69], there exist different approaches toward the improving error reporting: error explanation systems [5] and error reporting systems [67].

Our contribution in the domain is the study and extension of the type error slicing framework developed by Haack and Wells [31]. More precisely, we have extracted the theory from the current implementation of the framework, we are formalising its properties and proofs and we are in parallel extending the framework to a richer language.

5.2 Background

Type error slicing (TES for short) considers a sub-language of SML. It is defined as follows:

- Syntax: a set of terms.
- A set of types/type environments.
- Static semantics: a type system.

The language we consider is the one used in the current implementation of TES which is larger than the one considered by Haack and Wells [31]. The language considered in the implementation adds nice features to the language considered by Haack and Wells [31], such as the possibility of defining recursive functions.

Throughout this report we heavily use the notation of Haack and Wells [31]. Furthermore, we call *term* or *piece of code* an object of the defined syntax and we call *program* a well typed term w.r.t. the given type system. The subterm notion is the usual one. We call *term point* an occurrence of a subterm in a term. In TES, these term points are represented by labels associated to terms (in a term, to each occurrence of a subterm is associated a unique label).

A *type constraint* is an object of the form $\tau_1 \overset{l}{=} \tau_2$, where τ_1 and τ_2 are types and l is a label. As we will see below, a type constraint is associated to a term point so it is annotated by the label corresponding to the term point. Such a constraint is said to be *solvable* (or *satisfiable*) if there exists a substitution *sub* (a function which associate types to type variables) such that the application of *sub* to τ_1 is equal to the application of *sub* to τ_2. The substitution *sub* is then a *solution* (or *unifier*) of the constraint $\tau_1 \overset{l}{=} \tau_2$. A set of type constraints is satisfiable if there exists a substitution *sub*, which is a solution to each type constraint in the set. The substitution *sub* is then a solution (or unifier) of the set of constraints. The substitution *sub* is a *most general unifier* (mgu) of a set of type constraints if it is a solution of the set such that for any solution *sub'* of the set, there exists a substitution such that the application of this substitution to *sub* is equal to sub'. Such common concepts are defined for example by Baader and Nipkow [2].

We sometimes call error (or type error) the set of term points associated to an unsolvable set of type constraints.

5.3 The steps of Type Error Slicing

Even if the language of the current theory of TES [31] and the one of its implementation are different, the steps of TES are the same in the theory and in its implementation. The language of the implementation being more complicated than the one of the theory, most of the different algorithms used through these steps are more complicated in the implementation (and so are the proofs of their properties). In this section we describe the different steps (the algorithms used for these steps and their properties) of TES extracted from its current implementation and explain what we have achieved for each of them.

The different steps of TES and what we achieved are as follows:

- **The assignment of a set of type constraints to a piece of code**. This is done using a constraint generator. Given a piece of code, the constraint generator outputs a type environment, a type and a set of constraints. The constraints are type constraints corresponding to term points (each type constraint is generated because of a term point).

  We aim to prove the correspondence between the type system and the constraint generator, i.e.:

  - If a piece of code is typable in the given type system then the constraint generator generates a satisfiable set of type constraints.
  - If given a piece of code, the constraint generator generates a satisfiable set of type constraints then the piece of code is typable in the given type system.

  Roughly speaking, this correspondence allows us to prove that the constraint generator "accepts" the same set of terms than the type system.

  This goal has largely been achieved and we hope that a complete version will be submitted soon.

  We are also interested in proving that the defined language is a subset of SML.

  This is left for the near future.

- **The unification of a set of type constraints**. This is done using a unification algorithm based on the Martelli-Montanari algorithm [6]. Given a set of type constraints, the algorithm outputs either an error if the set is not satisfiable (we say that the algorithm fails) or a unifier if the set is satisfiable (we say that the algorithm succeeds). If the set of constraints is satisfiable, from the outputted unifier it is then possible to compute a most general unifier of the set. When the algorithm outputs an error it outputs the term points responsible for the type error.

  The aim of TES being to provide a "good" representation of a type error of a piece of code, we are interested in the case when the unification fails.

  We aim to prove the termination and the correctness of the algorithm. Given an input (a set of type constraints), the correctness of the unification algorithm states that:

  - if the algorithm succeeds then it outputs a substitution from which it is possible to obtain a most general unifier of the set of type constraints.

– if the algorithm fails then it outputs a set of term points such that the corresponding set of type constraints is unsolvable.

This goal has been completed.

- **The minimisation of an error**. Given a set of term points corresponding to an unsolvable set of type constraints, the minimisation algorithm outputs a minimal (w.r.t. the set inclusion) subset of this set of term points. The type constraints are associated to term points but it might happen that not all the term points involved in the set of type constraints are necessary to generate an error. Hence, the minimisation algorithm, making use of the unification algorithm, tries to build recursively a set of term points necessary and minimal to obtain an error (the corresponding set of type constraints is unsolvable). This is done using a property of the unification algorithm which is that when failing, along with a set of term points responsible for the error, it returns the last point checked during the unification process which led to the error. This point has the property of being in all the sets of term points subsets of the returned set of points and corresponding to an error (an unsolvable set of type constraints).

  We aim to prove the termination and the correctness of the minimisation process. The correctness of the minimisation process states that given an unsolvable set of type constraints and a set of term points responsible for an error and involved in the set of type constraints, if the algorithm outputs a set of term points then this set is a minimal error of the set of constraints, included in the given set of term points.

  We extracted the algorithm from the implementation but all the proofs of their properties have to be provided.

- **The enumeration of type errors**. The enumeration algorithm consists in the enumeration of the set of errors of an unsolvable set of type constraints. It aims to enumerate all the set of term points corresponding to an error in the unsolvable set of type constraints. This is done using a filter on the term points already considered during the process and making an extensive use of the unification algorithm. As explained by Haack and Wells [31], in few cases this process can behave badly. In order to avoid the unpleasantness of these cases, the implementation of TES stops the process after a short period and returns the found set of errors. This is done using a timer. If the set of type constraints is unsolvable, the timer starts only after that one error has been found.

  We aim to prove the termination and the correctness of the enumeration process. The correctness of the enumeration process states that if it returns a set of errors then this set is the set of minimal errors of the unsatisfiable set of type constraints (in practice we can only ensure that it is a subset of the set of minimal errors of the unsatisfiable set of type constraints).

  We extracted the algorithm from the implementation but all the proofs of their properties have to be provided.

- **Slicing the piece of code**. This part consists in the displaying of the parts of the piece of code corresponding to a minimal error.

  This part is still at initial developments.

To summarise, from a piece of code, TES first generates a set of type constraints. Then, from this set of type constraints, it runs the enumeration of the set of minimal errors of the piece of code (as we explained above, for some time reasons, maybe not all of them will be enumerated). Finally, for each minimal error found, it displays the corresponding slice (parts of the piece of code corresponding to the minimal error).

6 Plan of the thesis

During the third year I will focus on the two following subjects:

- Type error slicing:
  - Finishing the proofs relative to the current implementation of type error slicing so that for the first time, a full and precise connection is given between the theory and implementation of a type error slicer. Although doing the formal foundation from scratch as explained in section 5.3 has been time consuming, the work so far, has only been carried our for a kind of the implemented toy language (without real application). The aim is to build the work for a rich and sophisticated programming language (closer to SML).
  - Extending the work outlined in section 5.3 for a rich and sophisticated programming language.
  - Implementing the type error slicer we will develop for the rich language so that we can make our development more practical and can have more impact.
- Semantics of expansion. The aim of this project is to provide the semantics of an intersection type system with expansion (to provide information on expansion). The idea was to use a realisability semantics to interpret types of an intersection type systems with expansion. The whole expansion being complicated, the project started with the study of expansion variables (instead of the whole expansion). The untyped λ-calculus was not suitable to give the semantics of such a system. So a new calculus based on the untyped λ-calculus augmented with levels have been developed. It seems that we are trying to give a semantics of expansion using some variants of the untyped λ-calculus (the calculi of the considered intersection type systems). However, expansion does not act on terms, only on proofs (Carlier's skeletons [9, 7]). It might be why the considered calculi are getting more and more complicated when trying to give a semantics of expansion. This complication leads to the question whether a suitable calculus exists. We should maybe consider giving a semantics of expansion using skeletons instead of terms. In any case, our study for expansions has only dealt with expansion variables and we urgently need to start the work on the rewriting operations of expansions. We will consider using skeletons in our further semantics.

Timetable:

- August 2008: Finish the part of type error slicing on the toy language.
- September 2008 to January 2009: Develop in parallel the type error slicer for a rich language and the semantics of expansion.
- February 2009 to March 2009: Submit papers based on these works.
- April 2009 to August 2009: write my PhD thesis.

7 Conclusion

In summary, the second year concentrated on:

- Simplifying and generalising/non trivially extending proofs of important properties of the λ-calculus and its variants.
- Proving semantics for intersection typed λ-calculi with expansion. We developed more and more complicated realisers to provide such semantics. Although we made some progress on this subject, we still have to provide a semantics of the mechanism of expansion.
- Extracting the theory behind the implementation of type error slicing. This theory is an extension of the theory presented by Haack and Wells [31]. We started in parallel the development of an extension of the current type error slicing framework. We plan to have this extension as the main topic of research during the third year. We hope to have ready in one year time the full theory of this extension as well as a working implementation of the theory. We believe that the extraction of the theory of the implementation was of real importance for a deep understanding of the tools used by type error slicing. We also believe that it will appear to be useful as a basis for the theory of the extension.

References

[1] S. N. Artemov. Explicit provability and constructive semantics. *The Bulletin of Symbolic Logic*, 7(1), 2001.

[2] F. Baader, T. Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

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

[4] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, vol. 2. Oxford University Press, Inc., New York, NY, USA, 1992.

[5] M. Beaven, R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4), 1993.

[6] M. Bezem, J. W. Klop, R. de Vrijer, E. Barendsen, I. Bethke, J. Heering, R. Kennaway, P. Klint, V. van Oostrom, F. van Raamsdonk, F.-J. de Vries, H. Zantema. *Term Rewriting Systems.* Cambridge University Press, 2003.

[7] S. Carlier. *Expansion Algebra: a Foundational Theory with Applications to Type Systems and Type-Based Program Analysis.* PhD thesis, Heriot Watt University, School of Mathematical and Computing Sciences, 2008.

[8] S. Carlier, J. Polakow, J. B. Wells, A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages & Systems, 13th European Symp. Programming*, vol. 2986 of *LNCS*. Springer-Verlag, 2004.

[9] S. Carlier, J. B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. In *Proc. 3rd Int'l Workshop Intersection Types & Related Systems (ITRS 2004)*, 2005. The ITRS '04 proceedings appears as vol. 136 (2005-07-19) of *Elec. Notes in Theoret. Comp. Sci.*

[10] A. Church. A set of postulates for the foundations of logic. *The Annals of Mathematics*, 33(2), 1932.

[11] A. Church. A proof of freedom from contradiction. *Proceedings of the National Academy of Sciences of the United States of America*, 21(5), 1935.

[12] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2), 1940.

[13] A. Church, J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3), 1936.

[14] M. Coppo, M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archive for Mathematical Logic*, 19(1), 1978.

[15] M. Coppo, M. Dezani-Ciancaglini, B. Venneri. Principal type schemes and λ-calculus semantic. In J. R. Hindley, J. P. Seldin, eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[16] T. Coquand. Completeness theorems and lambda-calculus. In P. Urzyczyn, ed., *TLCA*, vol. 3461 of *Lecture Notes in Computer Science*. Springer, 2005.

[17] H. B. Curry, R. Feys. *Combinatory Logic I.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.

[18] L. Damas, R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 1982. ACM.

[19] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 5(34), 1972.

[20] S. Farkh, K. Nour. Résultats de complétude pour des classes de types du système AF2. *Theoretical Informatics and Applications*, 31(6), 1998.

[21] J. Gallier. On the correspondance between proofs and λ-terms. *Cahiers du centre de logique*, 1997. Available at `http://www.cis.upenn.edu/~jean/gbooks/logic.html` (last visited 2007–05–15).

[22] J. Gallier. Proving properties of typed λ-terms using realisability, covers, and sheaves. *Theoretical Computer Science*, 142(2), 2003. Available at `http://www.cis.upenn.edu/~jean/gbooks/logic.html` (last visited 2007–05–15).

[23] J. Gallier. Typing untyped λ-terms, or realisability strikes again!. *Annals of Pure and Applied Logic*, 91, 2003. Available at `http://www.cis.upenn.edu/~jean/gbooks/logic.html` (last visited 2007–05–15).

[24] J. H. Gallier. On girard's "candidats de reductibilité". 2002. Available at `http://www.cis.upenn.edu/~jean/gbooks/logic.html` (last visited 2007–05–15).

[25] S. Ghilezan, V. Kunčak. Confluence of untyped lambda calculus via simple types. *Lecture Notes in Computer Science*, 2202, 2001.

[26] J.-Y. Girard. Une extension de l'interpretation de godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. 1971.

[27] J.-Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur.* PhD thesis, Universite de Paris VII, 1972.

[28] G. Goos, J. Hartmanis, eds. *λ-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, March 15-27, 1975*, vol. 37 of *Lecture Notes in Computer Science*. Springer-Verlag, 1975.

[29] M. Gordon, R. Milner, L. Morris, M. Newey, C. Wadsworth. A metalanguage for interactive proof in lcf. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, 1978. ACM.

[30] M. J. C. Gordon, R. Milner, C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, vol. 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[31] C. Haack, J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3), 2004.

[32] J. R. Hindley. The simple semantics for Coppo-Dezani-Sallé types. In M. Dezani-Ciancaglini, U. Montanari, eds., *International Symposium on Programming, 5th Colloquium*, vol. 137 of *LNCS*, Turin, 1982. Springer-Verlag.

[33] J. R. Hindley. Curry's types are complete with respect to F-semantics too. *Theoretical Computer Science*, 22, 1983.

[34] J. R. Hindley. *Basic Simple Type Theory*, vol. 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[35] R. Hindley. Reductions of residuals are finite. *Transaction of the American Mathematical Society*, 240, 1978.

[36] R. Hindley. The completeness theorem for typing lambda-terms. *Theoretical Compututer Science*, 22, 1983.

[37] F. Kamareddine, T. Laan, R. Nederpelt. *A modern Perspective on Type Theory. From its Origins until Today.*, vol. 29. Applied Logic Series, 2004.

[38] F. Kamareddine, K. Nour. A completeness result for a realisability semantics for an intersection type system. *Annals of Pure and Applied Logic*, 146, 2007.

[39] F. Kamareddine, K. Nour, V. Rahli, J. B. Wells. Challenges and solutions to realisability semantics for intersection types with expansion variables. Submitted to Fundamenta Informaticae, 2008.

[40] F. Kamareddine, K. Nour, V. Rahli, J. B. Wells. A complete realisability semantics for intersection types and arbitrary expansion variables. In *ICTAC'08: 5th International Colloquium on Theoretical Aspects of Computing, The Marmara, Istanbul, Turkey, 1-3 September 2008*, vol. 5160 of *Lecture Notes in Computer Science*, 2008.

[41] F. Kamareddine, K. Nour, V. Rahli, J. B. Wells. Developing realisability semantics for intersection types and expansion variables. Presented to ITRS'08, 4th Workshop on Intersection Types and Related Systems, Turin, Italy, 25 March 2008, 2008.

[42] F. Kamareddine, V. Rahli. Simplified reducibility proofs of church-rosser for β- and $\beta\eta$-reduction. Accepted at LSFA'08, Third Workshop on Logical and Semantic Frameworks, with Applications, Salvador, Bahia, Brasil, 26 August 2008, 2008.

[43] F. Kamareddine, V. Rahli, J. B. Wells. The foundation of the implemented type error slicer and its extension to a rich subset of SML. In preparation, 2008.

[44] F. Kamareddine, V. Rahli, J. B. Wells. Reducibility proofs in the λ-calculus. Presented to ITRS'08, 4th Workshop on Intersection Types and Related Systems, Turin, Italy, 25 March 2008, 2008.

[45] F. Kamareddine, V. Rahli, J. B. Wells. Reducibility proofs in the λ-calculus. Submitted to Fundamenta Informaticae, 2008.

[46] S. C. Kleene. On the interpretation of intuitionistic number theory. *The Journal of Symbolic Logic*, 10(4), 1945.

[47] S. C. Kleene, J. B. Rosser. The inconsistency of certain foraml logics. *The Annals of Mathematics*, 36(3), 1935.

[48] J. W. Klop. *Combinatory Reductions Systems.* PhD thesis, Mathematisch Centrum, Amsterdam, 1980.

[49] G. Koletsos. Church-rosser theorem for typed functional systems. *Journal of Symbolic Logic*, 50(3), 1985.

[50] G. Koletsos, G. Stavrinos. Church-rosser property and intersection types. *Australasian Journal of Logic*, 2007.

[51] J. Krivine. *Lambda-Calcul : Types et Modèles.* Etudes et Recherches en Informatique. Masson, 1990.

[52] J. L. Krivine. *Lambda-calcul, types et modèles.* Dunod, 1990.

[53] R. Labib-Sami. Typer avec (ou sans) types auxilières.

[54] O. Lee, K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4), 1998.

[55] J.-J. Lévy. An algebraic interpretation of the *lambda beta* k-calculus; and an application of a labelled *lambda* -calculus. *Theoretical Compututer Science*, 2(1), 1976.

[56] B. J. McAdam. On the unification of substitutions in type inference. *Lecture Notes in Computer Science*, 1595, 1999.

[57] R. Milner, M. Tofte, R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[58] R. Milner, M. Tofte, R. Harper, D. Macqueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, MA, USA, 1997.

[59] J. B. Rosser. Highlights of the history of the lambda-calculus. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, New York, NY, USA, 1982. ACM Press.

[60] B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3), 1908.

[61] N. Shankar. A mechanical proof of the church-rosser theorem. *Journal of the ACM*, 35(3), 1988.

[62] W. W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2), 1967.

[63] M. Takahashi. Parallel reductions in lambda-calculus. *Journal of Symbolic Computation*, 7(2), 1989.

[64] A. S. Troelstra. History of constructivism in the 20th century. Available at `http://staff.science.uva.nl/~anne/` (last visited 2007–05–15).

[65] A. S. Troelstra, D. van Dalen. *Constructivism in Mathematics.* 1988.

[66] J. van Heijenoort, ed. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931.* Harvard University Press, Cambridge, Massachusetts, 1967.

[67] M. Wand. Finding the source of type errors. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, 1986. ACM.

[68] J. Yang. Explaining type errors by finding the source of a type conflict., 2000.

[69] J. Yang, J. Wells, P. Trinder, G. Michaelson. Improved type error reporting., 2000.