Incremental Inference of Partial Types

Mario Coppo and Daniel Hirschkoff

Novembre 1998

N^o 98-139

Incremental Inference of Partial Types

Mario Coppo and Daniel Hirschkoff

Résumé

Nous présentons une procédure d'inférence de types partiels *pour un λ–calcul étendu avec des structures de données. Notre langage de types comprend des types de données, une notion de sous–typage, et un plus petit et un plus grand élément, désignés respectivement par ⊥ et ω; ω correspond à l'absence d'information de typage (si l'on veut, "tous les types sont possibles"). Par rapport aux études existantes, la singularité de notre approche réside dans son caractère* incrémental, *l'information de typage étant progressivement mise à jour au fur et à mesure que de nouvelles constantes sont définies dans le contexte. Cette manière de procéder est bien adaptée par exemple pour des systèmes dans lesquels on définit des fonctions (partielles en général) sur les types de données par l'intermédiaire d'équations. Nous illustrons le fonctionnement de nos algorithmes sur une implémentation qui a été réalisée en vue d'une intégration à la CuCh machine, développée à l'Université de Rome.*

Abstract

We present a type inference procedure for partial types *for a λ–calculus equipped with datatypes. Our procedure handles a type languages containing greatest and lesser types (ω and ⊥ respectively), subtyping, and datatypes (yielding constants at the level of terms). The main feature of our algorithm is* incrementality*; this allows us to progressively analyse successive term definitions, which is of interest in the setting of a system like the CuCh machine (cuurently being developped at the University of Rome). The methods we describe have led to an implemention; we illustrate its use on a few examples.*

1 Introduction

This paper focuses on the problem of type inference for *partial types*. Partial types have been introduced in [Tha94] (following [Gom90]), to describe some terms that are usually considered as ill–typed in a classical setting. Examples of such terms are auto–applications (e.g. $\lambda x.\,(x\,x)$), or polymorphic lists (e.g. $[\mathsf{true};\ \lambda f\,x.\,(f\,x)]$). In partial types, the language of types is equipped with a special type, called ω, to represent the absence of type information (or alternatively *"any possible type"*); ω is associated to the "weirdly typed" subterms of a given term, and allow one to avoid rejecting terms that contain such ill–typed subparts.

In this setting, to provide some kind of flexibility to the type system, and to capture by doing so as many terms as possible, the type language is enriched with a notion of *subtyping*, ω being naturally considered as the greatest type. This way, the type of any term can be coerced to a greater type, which allows one to preserve the soundness of the typing as applications are performed. One is able for example to infer the judgment

$$\lambda x.\,(x\,x)\ :\quad (\omega{\to}\alpha){\to}\alpha$$

(where α is a type variable), the type associated to the occurrence of x in argument position being coerced from $\omega{\to}\alpha$ to ω in order to permit the auto–application, yielding final type α.

The question of partial type inference, as adressed in [Tha94], is shown to be decidable in [WO92], and [KPS94] provides an efficient algorithm to solve the problem. Our study differs from these works by two main aspects. First of all, the language we focus on is equipped with user–defined datatypes (as well as with a least type, written $\bot$, that has to be introduced mainly for technical reasons). The introduction of (parametrised) datatypes somehow increases the complexity in the structure of the typing information that has to be dealt with, as will be seen thorough this study.

The second main original aspect of our work is the stress that is put on *incrementality* in defining the type inference method. Indeed, the traditional approach to type inference in presence of subtyping (not only for partial type inference, but also in other frameworks, e.g. the study of object–oriented paradigms) consists in exploring the structure of the term to be typed, and, while doing so, in collecting the corresponding subtyping constraints. Once all these constraints are put together, one can attack the problem of constraints satisfiability using many different approaches ([WO92] for example builds a "table" to represent the typing structure of the term, and [KPS94] uses an automata–based method).

In this paper, we try on the contrary to preserve the readability of the type information along the exploration of the term. Our approach, inspired by [WO92], consists in representing the typing information on a table; in doing this, however (and this is were our study differs from [WO92]), we are interested in inferring the consequences of the type constraints as soon as they are generated, and in resolving immediately the possible resulting inconsistencies. To achieve this, we introduce a notion of *guarded* constraint, that, in conjunction with an additional axiom for subtyping (stating $\alpha{\to}\omega \leq \omega{\to}\omega$ for any type variable α), allows us to define an incremental and quite flexible type inference procedure, as will be shown below. Intuitively, the guards are used to get rid of "contradictory" type constraints, in order to work only with *consistent* tables, i.e. those tables for which a non–trivial solution can be found.

Such a type inference method is interesting in the framework of a programming language, where the user can succesively define several objects, possibly using previous definitions for the introduction of new functions. This is typically the case for the *CuCh machine*, a system developped at the University of Rome in the team of Corrado Böhm (the design of a type inference procedure for this system actually originated our work).

The CuCh machine is a programming language based on the untyped λ-calculus. There are two modes to define objects in CuCh, called `@lam` and `@env`; in `@lam`, the user defines λ-terms using abstraction, application, and some built-in constants including natural numbers, strings, lists and boolean tests. The `@env` mode is used to define functions on free algebras by sets of equations, following [BB85]. The introduction of free algebras and of recursive definitions over these algebras is akin to the classic second-order encoding of datatypes; however, in CuCh, the solutions to (possibly recursive) definitions are not defined using a fixpoint operator, but rather following the Böhm-Piperno technique of [BPG], using self-application. In this setting, more freedom is allowed in the construction of terms, and "traditional" type systems for functional languages *à la* ML can sometimes be too restrictive.

As said above, this work stems from the will to define a type system that is well suited for the CuCh. Such a type system is by definition not part of the *design* of the programming language, as is usually the case, but should rather be seen as a *feature*. Following this remark, the typing judgment in the CuCh machine is essentially seen as giving a *descriptive* information, rather than some kind of advise about correctness (in particular, no term should be ruled out using the typing relation). Consequently, two aspects of the type system are important: first, a "pure" type inference approach

should be adopted, as opposed to the type checking method: we do not want the typing relation to interfer with the activity of the CuCh user, through the requirement of type annotations or other such informations that could be useful for the typing procedure. Secondly, the type system should be adaptive: in a pure calculus setting, it can be the case that one defines some terms that "look weird" from a classical typing point of view, but then uses these terms in a certain fashion (akin in some way to a type discipline), that actually expresses their meaning. These two remarks led to the design of the type inference procedure that is presented here.

The paper is organised as follows. In Section 2, we introduce our system, defined by the terms, the (possibly recursive) definitions, the language of types (including user–defined datatypes), and the two judgments corresponding to the typing and subtyping relations. Section 3 is devoted to the technical definitions we need for our type inference procedure, i.e. tables (to represent the type constraints), properties of tables, and various functions over tables. We define our type inference method in Section 4, as well as an heuristic to recover consistency where an inconsistent table is generated during the type inference process (in general, the type inference procedure is indeed defined in a non–deterministic way, in order to preserve completeness; the heuristic is hence given for the purpose of implementation). We finally conclude, and present in the appendix the implementation of our methods that has been designed (an example is given to illustrate the behaviour of our algorithms).

2 The system

Terms The terms we use are defined by the following syntax:

$$M \;=\; \lambda x.M \mid x \mid M\,N \mid c\,.$$

In the definition above, c ranges over datatype constructors, that shall be introduced below.

Recall that in CuCh, recursive functions are not introduced with a fixpoint–like construct, but are instead given by recursive equations on datatypes (introduced below).

Datatypes Following [BB85], a datatype is introduced by defining its name, parameters, and constructors (which in turn are characterized by a name and a list of types for their arguments).

The syntax we will adopt for datatype definitions is:

$$\begin{array}{lll} \text{Datatype } D[X_1,\ldots,X_k] & \underline{is} & c_1^D : arg[T_1^1;\ldots;T_{m_1}^1] \\ & & \vdots \\ & & c_n^D : arg[T_1^n;\ldots;T_{m_n}^n] \end{array}$$

where the X_is are the parameters of the datatype and each T_j^i is either a parameter X_j or another datatype (possibly D itself) having the shape $T_{i,j}[X_{p_1},\ldots,X_{p_l}]$ where the X_{p_j}s denote the free occurrences of $X_1,\ldots,X_k$ in $T_{i,j}$ ($\{X_{p_1},\ldots,X_{p_l}\} \subseteq \{X_1,\ldots,X_k\}$).

Remark: It is natural in CuCh to manipulate objects like polymorphic lists (e.g. $[\mathsf{true};\ 1;\ \lambda x.\, x]$), i.e. objects in which the datatype parameter (here X in $cons : X \to list[X] \to list[X]$) would be considered as being misused in a second–order setting (à la [GLT89]). The subtyping relation given below allows us to accomodate with such objects, in order to give as much type information as possible (in the case of $[\mathsf{true};\ 1;\ \lambda x.\, x]$, it is preferable to say "list of anything" rather than just "type error").

The definition above reads *"D is a datatype that has n constructors and k parameters $X_1,\ldots,X_k$; each constructor c_i^D, for $1 \leq i \leq n$ has type*

$$T_1^i \to \ldots \to T_{m_i}^i \to D[X_1,\ldots,X_k]$$

where the T_is are either parameters or datatypes". Note that nested arrow types are not allowed in the definition of constructors.

Example: In this framework, the declaration of the dataype $List$ would be as follows:

$$\begin{array}{lll} \text{Datatype } List[X] \ \underline{is} & c_1^{List} : arg[] & (Nil) \\ & c_2^{List} : arg[X;\ List[X]] & (Cons) \end{array}$$

Definitions The CuCh definitions are given by the following syntax:

$$< eq > \quad = \quad < \mathtt{f}\ (c_i^D\ x_1 \ldots x_{m_i}) = e > \ .$$

e is an expression possibly containing occurrences of the x_is and of $\mathtt{f}$, hence we deal in general with recursive equations. Recursive equations are used as an alternative to the case construct (a case–like definition can easily be translated into a set of recursive equations).

We write $< eqs >$ to range over a sequence of definitions.

Types Types are either type variables, data types, arrow types, or two special types, $\bot$ and ω, that respectively represent the empty type (or the least type, see below), and the union of all types.

$$T \;=\; \alpha \mid D[T_1, \ldots, T_n] \mid T \to U \mid \omega \mid \bot$$

Remark 2.1 *The intuitive semantics of types in our approach relies on the notion of types as topologically closed subsets (ideals) of the domain of interpretation of the language [CC90]. This model also support the notion of recursive type and recursive type equation. In this case the least (undefined) element of the domain belongs to every type. The type ω is then interpreted as the whole domain while $\bot$ is interpreted as the singleton containing only the least element of the domain. This provides a justification of the consistency and semantic correctness of our subtyping assumptions.*

Subtyping relation We define a subtyping relation between types; we shall use it to coerce types to a greater type (possibly involving ω and/or $\bot$) in such a way as to be able to give as much type information as possible about a "weirdly typed" term.

The subtyping relation is decomposed into two parts; the main judgment relies on a set of subtyping assumptions of the form $T \leq T'$, where T and T' are type expressions, and writes as follows:

$$\Sigma \vdash A \leq B\,.$$

Such a relation reads *"under typing assumptions Σ, it holds that $A \leq B$"*. This judgment is defined in a mutual recursive way together with a form of "structural subtyping" on datatypes, written $D \sqsubseteq D'$, meaning that datatype D is "structurally smaller" than D'. The rules that define both these judgments are given on Figure 1.

Let us make a few comments about the definition of the subtyping relation. Regarding relation $\sqsubseteq$, it has to be noted that one can always suppose that both datatypes have the same number of parameters, some of them possibly being unused in the smaller type. Moreover, as the context of typing assumptions is empty in $\emptyset \vdash T_l^i \leq T'^i_l$ (rule $(D_\sqsubseteq)$), this condition means that either T_l^i and T'^i_l are comparable datatypes, or they represent *the same* type variable.

Examples: we illustrate the meaning of relation $\sqsubseteq$ on two examples.

$$(D_{\sqsubseteq})\quad \frac{\begin{array}{l} D[X_1,\ldots,X_k]\ \underline{is}\ (c_1 : arg[T^1_1;\ldots;T^1_{m_1}]\ldots c_n : arg[T^n_1;\ldots;T^n_{m_n}]) \\ D'[X_1,\ldots,X_k]\ \underline{is}\ (c_1 : arg[{T'}^1_1;\ldots;{T'}^1_{m_1}]\ldots c_n : arg[{T'}^n_1;\ldots;{T'}^n_{m_n}]\ldots \\ \qquad\qquad \ldots c_{n+l} : arg[{T'}^{n+l}_1;\ldots;{T'}^{n+l}_{m_{n+l}}]) \end{array}}{D \sqsubseteq D'}$$

where $\emptyset \vdash T^i_l \leq {T'}^i_l$ $(1 \leq l \leq n,\ 1 \leq i \leq m_l)$.

$$(S_\perp)\ \ \Sigma \vdash \perp \leq T \qquad (S_\omega)\ \ \Sigma \vdash T \leq \omega \qquad (S_{id})\ \ \Sigma \vdash \alpha \leq \alpha$$

$$(S_{var})\ \ \Sigma.\{T \leq T'\} \vdash T \leq T' \qquad (S_{ax})\ \ \Sigma \vdash A{\rightarrow}\omega \leq \omega{\rightarrow}\omega$$

$$(S{\rightarrow})\ \ \frac{\Sigma \vdash A_2 \leq A_1 \quad \Sigma \vdash B_1 \leq B_2}{\Sigma \vdash A_1{\rightarrow}B_1 \leq A_2{\rightarrow}B_2}$$

$$(S_D)\ \ \frac{D \sqsubseteq D' \qquad \forall i,\ (1 \leq i \leq k)\,.\,\Sigma \vdash A_i \leq A'_i}{\Sigma \vdash D[A_1;\ldots;A_k] \leq D'[A'_1;\ldots;A'_k]}$$

Figure 1: Subtyping relation

- Consider the datatypes of booleans and tri–valued tags, defined as follows:

  Datatype $Bool$ $\underline{is}$ $\texttt{true} : arg[]$, $\texttt{false} : arg[]$;
  Datatype $Bool'$ $\underline{is}$ $\texttt{true}' : arg[]$, $\texttt{false}' : arg[]$, $\texttt{unknown} : arg[]$.

  It holds that $Bool \sqsubseteq Bool'$, because $Bool'$ has two constructors in common with $Bool$, and one extra constructor (no parameter is involved here).

- Suppose now we want to tag a term (of any type) with an element of $Bool$ or of $Bool'$; this would lead to the following definitions:

  Datatype $Tagged[X]$ $\underline{is}$ $c : arg[X, Bool]$;
  Datatype $Tagged'[X]$ $\underline{is}$ $c' : arg[X, Bool']$.

  We can derive $Tagged \sqsubseteq Tagged'$: indeed, they have the same number of parameters, the constructors c and c' have the same shape, and we

can derive both subtyping judgments $\emptyset \vdash X \leq X$ and $\emptyset \vdash Bool \leq Bool's$ for their first and second argument respectively.

Let us now consider the definition of relation $\leq$: ω is the *universal type*, as expressed by rule (S_ω); dually, $\bot$ is the least type (sometimes referred to as *unit*). As usual, the arrow constructor is antimonotonic in its first argument, and monotonic in its second argument. As said above, the subtyping relation for datatypes (instanciated with their parameters) is factorised into the "structural" relation $\sqsubseteq$ and rule (S_D) for deriving actual type inclusions; in contrast with rule $(S_\rightarrow)$, rule (S_D) introduces monotonicity with respect to all the parameters of a datatype. Axiom (S_{ax}) somehow weakens this opposition, as it says that any arrow type can be coerced to the most general type for functions, namely $\omega \rightarrow \omega$ (which could be seen as a form of monotonicity of the arrow construct towards its left–hand side argument).

Typing rules

- *Typing terms*

  The typing judgment for terms is of the form

  $$\Gamma, \Sigma \vdash M : T\,,$$

  where M is a term, T is a type, Γ is a set of typing assumptions for the free variables of M, and Σ is a set of inequalities between types. The rules defining this judgment are given below:

  $$(var)\ \ \Gamma.x : A, \Sigma \vdash x : A \qquad (omega)\ \ \Gamma, \Sigma \vdash M : \omega$$

  $$(\rightarrow_I)\ \ \frac{\Gamma.x : A, \Sigma \vdash M : B}{\Gamma, \Sigma \vdash \lambda x.\, M : A \rightarrow B}$$

  $$(\rightarrow_E)\ \ \frac{\Gamma, \Sigma \vdash M : A \rightarrow B \quad \Gamma, \Sigma \vdash N : C \quad \Sigma \vdash C \leq A}{\Gamma \vdash (M\ N) : B}$$

- *Typing definitions*

  We introduce a judgment, written $\Sigma\ :\ \Gamma, < eqs > \Rightarrow \Gamma'$, to express that adding definitions $< eqs >$ to the context Γ assuming the type inqualities of Σ we obtain an extended context Γ'. This judgment is defined by the rules of Figure 2.

  In general, we will be in the situation where we process a sequence of definitions $< eqs >$, starting from an initial context Γ_0 and a set of inequalities Σ, and trying to infer a judgment of the form $\Sigma\ :\ \Gamma_0, < eqs > \Rightarrow \Gamma$.

$$(C_{\mathtt{f}})\ \frac{\Gamma, \mathtt{f} : \alpha, x_1 : \beta_1, \ldots, x_n : \beta_n, \Sigma \vdash e : \gamma \quad \Gamma, \mathtt{f} : \alpha, x_1 : \beta_1, \ldots, x_n : \beta_n, \Sigma \vdash f(c(x_1 \ldots x_n)) : \delta \quad \Sigma \vdash \gamma \leq \delta}{\Sigma \ : \ \Gamma, < \mathtt{f}\,(c\,x_1 \ldots x_n) \ = \ e > \Rightarrow \ \Gamma.f : \alpha}$$

$$(C_{trans})\ \frac{\Sigma \ : \ \Gamma, < eqs > \Rightarrow \ \Gamma' \quad \Sigma \ : \ \Gamma', < eqs' > \Rightarrow \ \Gamma''}{\Sigma \ : \ \Gamma, < eqs > . < eqs' > \Rightarrow \ \Gamma''}$$

Figure 2: Compatibility relation between contexts and definitions

3 Systems of type constraints

3.1 Type constraints and tables

Our inference procedure is based on the representation of relations between types by sets of inequalities between types called type constraints in the literature. In this section we define the procedures to handle them.

A substitution is defined here as a finite mapping σ between type variables and types, that is naturally extended to a congruence between all types. A single substitution is denoted $[t := A]$: it replaces t by A and behaves like the identity on all other variables. Similarly, $[t_1 := A_1, \ldots, t_n := A_n]$ (where t_i does not occur in A_j for all $1 \leq i, j \leq n$) denotes the composition of n single substitutions. If all the type expressions A_i are single variables, we say that $[t_1 := A_1, \ldots, t_n := A_n]$ is *trivial*.
A substitution is *ground* if all types in its domain are ground.

Definition 3.1 (Constraints) *A* type constraint *is an expression of the form $t \leq D(u_1, \ldots, u_k)$, $D(u_1, \ldots, u_k) \leq t$ or ($0 \leq n$), where D is an algebraic datatype constructor or the $\rightarrow$ constructor. and $t, u, u_1, \ldots, u_k$ are type variables.*

To handle the rule (S_ω) and the axiom (S_{ax}), we need the notion of guard an some operators on it.

Definition 3.2 *(i) A* guard *is an ordered list of type variables. Let w range over guards.*

(ii) If w_1, w_2 are guards then $w_1 \triangleright w_2$ is the guard obtained by concatenating w_1 and w_2 and by eliminating from w_2 the variables which already occur in w_1.

(iii) A guarded constraint *(g.c. for short) is an expression of the form $w :_G (A \leq B)$ where w is a guard and $A \leq B$ is a type constraint.*

If $S = \{w_i{:}_G\,(A_i \leq B_i) \mid 1 \leq i \leq n\}$ is a set of g.c. and w is a guard then $w \triangleright S$ denotes the set $\{w \triangleright w_i{:}_G\,(A_i \leq B_i) \mid 1 \leq i \leq n\}$.

A guard hides the constraint associated to it in an expression $w{:}_G\,(A \leq B)$ whenever at least one of the variables occurring in it is an ω-type.

A *solution* of a set S of g.c. in a pair $\langle \sigma_g, \Sigma \rangle$ where σ_g is a ground substitution and Σ a set of subtyping assumptions such that for all $w{:}_G\,(A \leq B) \in S$ in which $\sigma_g(t) \neq \omega$ for all variables t in w we have $\Sigma \vdash \sigma_g(A) \leq \sigma_g(B)$. A *strong* solution of S is a ground substitution σ_g such that for all $w :_G (A \leq B) \in S$ we have $\sigma_g(A) \leq \sigma_g(B)$ (ignoring guards).

The inference algorithm keeps the information about the types involved in a deduction using the notion of *table*, which has been inspired by [WO92].

A table is simply a structured set of type constraints, which are represented in a slightly different way via the notion of guarded elementary expression.

Definition 3.3 *(i) A* guarded elementary expression *(g.e. for short) is an expression of the shape* $w :_G (D(v_1, \ldots, v_k))$ *where* $v_1, \ldots, v_k$ *are variables and* D *is a type constructor.*

(ii) A table Θ *is a set of triples* $\langle t, L, U \rangle$ *(called the* entries *of the table), where* t *is a variable and* L *and* U *are sets of g.e. which are said, respectively, the* lower *and* upper *sets of* t *in* Θ*. If* $\langle t, L, U \rangle \in \Theta$ *we denote* L *as* $L_\Theta(t)$*, or simply* $L(t)$ *(when* Θ *is understood) and* U *as* $U_\Theta(t)$*, or simply* $U(t)$*. Moreover define* $\mathtt{dom}(\Theta) = \{t \mid \langle t, L, U \rangle \in \Theta\}$.

A table is just a structured way of representing a set of elementary g.c.s. In fact each $w :_G (A) \in L(t)$ represents a g.c. $w :_G (A \leq t)$, and each $w{:}_G\,(A) \in U(t)$ also represents a g.c. $w{:}_G\,(t \leq A)$. A solution of a table is a solution of the corresponding set of g.e.s.

A *simplified* table (s-table for short) Ξ is a structure which has the same form of a table but without guards. So the elements of the upper and lower sets are type expressions (containing only one type constructor) instead of g.e.s. The *kernel* of a table Θ, written $\mathtt{kernel}(\Theta)$ is a s-table Ξ obtained from Θ by erasing all guards. S-tables will be useful to build solutions of tables.

The following operation of closure on tables puts them in a sort of normal form.

Definition 3.4 *A table* Θ *is* closed *if it satisfies the following condition. For all* $t \in \mathtt{dom}(\Theta)$ *such that both* $L(t)$ *and* $U(t)$ *are non nonempty and for all* $w_1 :_G (D_1(u_1, \ldots, u_k)) \in L(t)$ *and* $w_2 :_G (D_2(v_1, \ldots, v_k)) \in U(t)$ *such that* $D_1 \sqsubseteq D_2$ *we must have:*

1. *If D_1, D_2 are algebraic data type constructors*

   - $w_1 \triangleright w_2 \triangleright L(u_i) \subseteq L(v_i)$ *for* $1 \leq i \leq k$.
   - $w_1 \triangleright w_2 \triangleright U(v_i) \subseteq U(u_i)$ *for* $1 \leq i \leq k$

2. *If* $D_1, D_2 = \rightarrow$ *(and then* $k = 2$*):*

   - $w_1 \triangleright w_2 \triangleright L(u_2) \subseteq L(v_2)$.
   - $w_1 \triangleright w_2 \triangleright U(v_2) \subseteq U(u_2)$
   - $v_2 \triangleright w_1 \triangleright w_2 \triangleright L(v_1) \subseteq L(u_1)$
   - $v_2 \triangleright w_1 \triangleright w_2 \triangleright U(u_1) \subseteq U(v_1)$

A table obtained from a set of elementary g.c. is in general not closed. It is easy to define an algorithm `closure` that takes a table Θ in input and returns its closure $\texttt{closure}(\Theta)$ by adding elements to the sets $L(t)$, $U(t)$ according to the above definition. Since all the new constraints added to a table by `closure` are simply consequences of the definition of the $\leq$ relation we have immediately the following lemma:

Lemma 3.5 *A table* Θ *and its closure* $\texttt{closure}(\Theta)$ *have the same solutions.*

We define now a criterium to decide whether a table has a solution. To this end recall that the set of type constructors is a partially ordered set. The notions of inf and sup of a subset of a poset as well as those of l.u.b. and g.l.b. are standard.

Note that any set of type constructors has a sup (ω) and an inf ($\perp$).

Let now $\mathcal{S}$ be a set of g.e.. Define $\texttt{constructors}(\mathcal{S})$ the set of all type constructors occurring in the type expressions of the g.e. of $\mathcal{S}$ (ignoring guards).

Definition 3.6 *(i) A closed table* Θ *is* consistent *with respect to a variable* $t \in \texttt{dom}(\Theta)$ *if there is a type constructor* D *which is a sup for the set* $\texttt{constructors}(L_\Theta(t))$ *and an inf for* $\texttt{constructors}(U_\Theta(t))$. *We also say the the entry* $\langle t, L_\Theta(t), U_\Theta(t)\rangle$ *is consistent.*

(ii) A closed table Θ *is* consistent *(tout court) if it is consistent with respect to every* $t \in \texttt{dom}(\Theta)$.

Note that a table Θ is always consistent with respect to a variable t if either $L_\Theta(t)$ or $U_\Theta(t)$ are empty. In this case, ω ($\perp$) is actually a sup (inf) of the empty set.

3.2 Solving tables

In this subsection we show that every consistent table admits a strong solution, and we give an algorithm to represent it in a general way. To obtain it we need to define some more transformations on tables. Since we are interested in a strong solution we consider here only simplified tables.

For technical reasons we will need, besides the usual notion of substitution, the following notion of substitution path. Let an *elementary substitution* (e.s. for short) $\mathtt{e}$ be an expression of the form $[t := D(u_1, \ldots, u_n)]$ where D is a type constructor and $u_1, \ldots, u_n$ are variables. A *substitution path* $\mathtt{s}$ (s.p. for short) is a list $\langle \mathtt{e}_1, \ldots, \mathtt{e}_n \rangle$ of e.s. such that for all $1 \leq i \neq j \leq n$, e_i and e_j have a different variable in the l.h.s.. Let $\mathtt{e}_i = [t := D(u_1, \ldots, u_n)]$ be an e.s. in a s.p. $\mathtt{s}$. We say that a variable u *depends* on t in $\mathtt{s}$ if either $u = u_k$ for some $1 \leq k \leq n$ or u depends on u_k in $\mathtt{s}$. Define $\mathtt{s} \setminus t$ as the s.p. obtained by eliminating from $\mathtt{s}$ the e.s. whose l.h.s. is t.

A substitution path $\langle \mathtt{e}_1, \ldots, \mathtt{e}_n \rangle$ such that none of its variables depends on themselves defines a substitution obtained as $\mathtt{e}_n \circ \ldots \circ \mathtt{e}_1$ (where $\circ$ denotes function composition). In this case we will identify substitution paths with the corresponding substitutions. Indeed even if there are variables t depending on themselves in a substitution path $\mathtt{s}$, we can always identify $\mathtt{s}$ with $\mathtt{e}_1 \circ \ldots \circ \mathtt{e}_n$, but in this case $\mathtt{s}(t)$ is a type expression containing t, so it is not a substitution in a strict sense. With some abuse of notation we will use this in the following.

A substitution path represents in general a substitution and a set of type equations in the following way.

Definition 3.7 *Let* $\mathtt{s}$ *a s.p. and let* $t_1, \ldots, t_h$ $(0 \leq h)$ *be the variables occurring in* $\mathtt{s}$ *that depend on themselves. Define*

$$\begin{array}{rcl} \sigma_{\mathtt{s}} & = & \mathtt{s} \setminus t_1 \ldots t_h \\ \Sigma_{\mathtt{s}} & = & \{t_i = \mathtt{s}(t_i) \mid 0 \leq i \leq h\} \end{array}$$

To make more readable the following constructions we will consider only the case of algebraic type constructors. All definitions and results of this section apply as well to the $\rightarrow$ type constructor. In this case it is enough, when considering the first argument of $\rightarrow$, to systematically exchange U and L, $\leq$ and $\geq$.

Existence of a solution

The aim of this subsection is to show that every consistent table has a strong solution. We will see that this is all we need to insure the soundness and

completeness of the inference algorithm.

First we define a function `solve`, that takes a simplified table Ξ, and returns a pair $\langle \Xi', \mathtt{s} \rangle$ where Ξ' is another simplified table extending Ξ, and $\mathtt{s}'$ is a s.p. which represents, in some sense, the basic step towards the solution of Ξ. We define function `solve` by giving an algorithm to compute it.

Definition 3.8 *The function* `solve` *is defined by the following steps. The basic operation is to build a sequence of s-tables* Ξ_i *and substitution paths* $\mathtt{s}_i$ *($i \geq 0$). During the construction, we "mark" some entries of the table (to remember that the substitution for the corresponding variables has already been generated).*

1. *Set $i = 0$. Let $\Xi_0 = \Xi$, $\mathtt{s}_0$ be the empty list* `nil`*. All entries of Ξ_0 are unmarked.*

2. *Take any unmarked entry $\langle t, L(t), U(t) \rangle$ of Ξ_i such that both $L(t)$ and $U(t)$ are not empty, mark it and define Ξ_{i+1} and σ_{i+1} in the following way:*

   (a) *If there is a marked entry $\langle v, L(v), U(v) \rangle$ of Ξ_i such that $L(v) = L(t)$ and $U(v) = U(t)$, and such that t depends on v in $\mathtt{s}_i$, then set $\mathtt{s}_{i+1} = \mathtt{s}_i[t := v]$ and $\Xi_{i+1} = \Xi_i$.*

   (b) *Otherwise let D be a type constructor that is a sup for the set* `constructors`$(L(t))$ *and an inf for* `constructors`$(U(t))$*. Let $k \geq 0$ be the arity of D (which is by hypothesis equal to the arity of any other type constructor in $L(t)$, $U(t)$) and let $t_1, \ldots, t_k$ be new fresh variables. Then:*

   - *Define $\mathtt{s}_{i+1}$ as the s.p. obtained by appending $[t := D(t_1, \ldots, t_k)]$ to the end of $\mathtt{s}_i$.*
   - *Define Ξ_{i+1} by adding to Ξ_i k entries for the new variables t_i ($1 \leq i \leq k$) and set:*

$$L(t_i) = \bigcup \{ L_{\Xi_i}(x_i) \mid D'(x_1, \ldots, x_k) \in L(t) \}$$
$$U(t_i) = \bigcup \{ U_{\Xi_i}(x_i) \mid D'(x_1, \ldots, x_k) \in U(t) \}$$

3. *Repeat step 2. until there are no more unmarked entries with both a lower and an upper set empty. Let n be the last value of i.*

4. *Return $\langle \Xi', \mathtt{s} \rangle$ where $\Xi' = \Xi_n$ and $\mathtt{s} = \mathtt{s}_n$.*

We have the following properties.

Lemma 3.9 *The construction in Def. 3.8 is always terminating.*

Proof hint. The construction builds new upper and lower sets but using only those type expressions that occur in the original table. Then there can only be a finite number of possible upper and lower sets. $\diamondsuit$

Indeed the new sets are built by composing the old ones. So their actual number is much less than the number of all possible subsets of the type expressions occurring in Ξ.

The proof of the following lemma follows directly from the definition of closure.

Lemma 3.10 *Let* $\langle \Xi', \mathtt{s} \rangle = \mathtt{solve}(\Xi)$. *Then* Ξ' *is consistent if* Ξ *is consistent.*

Lemma 3.11 *A consistent closed s-table* Ξ *has a solution.*

Proof hint. Let $\mathtt{solve}(\Xi) = \langle \Xi', \mathtt{s} \rangle$. Take the ground substitution γ defined by:

$$\gamma(t) = \begin{cases} \omega & \text{if } U(t) = \emptyset \\ \bot & \text{otherwise} \end{cases}$$

where t ranges over all variables having a non–marked entry in Ξ' (referring to Def. 3.8). Then it is easy to see that $\langle \gamma \circ \sigma_{\mathtt{s}}, \gamma(\Sigma_{\mathtt{s}}) \rangle$ solves the table. $\diamondsuit$

In the end we (immediately) get our final result

Corollary 3.12 *A consistent table has a strong solution.*

Finding a better solution

In Lemma 3.11, it is proved that a consistent table admits *at least* one ground solution. But in presenting the output of a typechecker, we are rather interested in showing a polymorphic type, possibly a "most general" type scheme such that all its instances represent legal typings of a term, in the style of the ML family of languages (see the discussion in the next section). In our case we define a *solution scheme* for an s-table Ξ as a pair $\langle \sigma, \Sigma \rangle$ such that for all (ground) substitutions γ $\langle \gamma \circ \sigma, \gamma(\Sigma) \rangle$ is a solution for Ξ.

We define a simple algorithm that builds a solution scheme for a given table, and which will provide a useful presentation of the type of an expression. Our goal in this case is to obtain a synthetic presentation of the functional properties of a term, rather than a complete one. So the solution scheme that we obtain could fail to capture some possible typings of

the term; these could be represented only at the cost of introducing more complex subtyping expressions. We remark however that the typechecking procedure defined in the next section keeps the whole table as an internal representation of the typing of a term, and so the loss of information takes place only at the level of interface with the user. The table itself is indeed the "most general" solution scheme for a given table.

We will build a solution scheme of an s-table Ξ through a function `get`, that is defined essentially in two steps. In the first step, formalised by a function `collapse`, the table is "flattened" into a simpler one, preserving most of the solutions. In some cases, however, this flattening could fail to produce a meaningful output.

Definition 3.13 *Let Ξ be a simplified table. Then* `collapse`(Ξ, V) *is a function that returns either a pair $\langle \Xi', \rho \rangle$ where Ξ' is an s-table and ρ is a trivial substitution (which will identify some variables of V), or* `failure`*. The function* `collapse` *is defined by the steps given below. Also in this case the basic operation is to build a succession of s-tables Ξ_i and trivial substitutions ρ_i ($i \geq 0$). During the construction, we assume that we are able to mark (and unmark) some entries of the considered tables.*

1. *Let $\Xi_0 = \Xi$ and ρ_0 be the empty substitution. Mark all entries in Ξ_0.*
2. *Take any marked entry $\langle t, L_t, U_t \rangle$ of Ξ_i. If it is not consistent then report* `failure`*.*
3. *If $L_t \cup U_t$ is empty or contains only one element then unmark the entry $\langle t, L_t, U_t \rangle$ in Ξ_i, define $\Xi_{i+1} = \Xi_i$, $\rho_{i+1} = \rho_i$ and go to step 2.*
4. *Otherwise let k be the arity of the type constructors in $L_t \cup U_t$. For all $1 \leq j \leq k$, perform the following operations.*
   - *Let $var_j = \{a \mid D(\underbrace{\ldots}_{j-1}, a, \underbrace{\ldots}_{k-j}) \in L \cup U\} = \{a_1, \ldots, a_p\}$ $(1 \leq p)$. Take a fresh variable u and let $\rho^* = [a_1 := u, \ldots, a_p := u]$.*
   - *Add to Ξ_i an entry $\langle u, L_u, U_u \rangle$ and set*
     - $L_u = \bigcup\{L_{\Xi_i}(a) \mid a \in var_j\}$
     - $U_u = \bigcup\{U_{\Xi_i}(a) \mid a \in var_j\}$
   - *Then eliminate from Ξ_i all entries for the variables in var_j, and replace Ξ_i by $\rho^*(\Xi_i)$, and ρ_i by $\rho^* \circ \rho_i$. Finally, mark the new entry for u.*

Define Ξ_{i+1}, ρ_{i+1} *as the result of this process.*

5. *As long as there are marked entries, go back to step 2.*

6. *Let* n *be the current value of* i. *Return* Ξ_n *and* ρ_n

Basically, the key step in procedure `collapse` consists in reading the various expressions that occur in the L and U set of a given row "transversally", and map all the variables read this way to a single, fresh variable (called u above). Of course, this corresponds to a simplification, that can imply the loss of completeness, and sometimes even lead to inconsistent tables (hence the check at step 2).

It is easy to see that if $\texttt{collapse}(\Xi, V) = \langle \Xi', \rho \rangle$ (and hence `collapse` does not return `failure`), then Ξ' is a closed table. Moreover we have the following lemma.

Lemma 3.14 *Let* $\texttt{collapse}(\Xi, V) = \langle \Xi', \rho \rangle$. *If* $\langle \gamma, \Sigma \rangle$ *is a solution of* Ξ', *then* $\langle \gamma \circ \rho, \rho(\Sigma) \rangle$ *is a solution of* Ξ.

Owing to the simple structure of the table produced by `collapse`, it is now easier to find type schemes to represent a solution. This is achieved via another algorithm `get'`, that first applies `collapse`, and then finds a solution scheme for the s-table. The function `get` takes an s-table Ξ and either fails (if `collapse` fails) or yields a pair $\langle \sigma, \Sigma \rangle$, that is a solution scheme for Ξ. `get` is also defined in two steps via the function `get'`.

Definition 3.15 *(i) Let* Ξ *be a simplified table. Then* $\texttt{get}'(\Xi)$ *is defined by the following steps. If* $\texttt{collapse}(\Xi)$ *returns* `failure` *then so does* $\texttt{get}'(\Xi)$. *Otherwise* `get'` *yields a pair* $\langle \texttt{s}, \rho \rangle$ *where* `s` *is a substitution path and* ρ *is a trivial substitution defined in the following way. Let* $\texttt{collapse}(\Xi) = \langle \Xi', \rho \rangle$. *Apply to* Ξ' *the following construction (during the construction, we mark some entries in* Ξ'*):*

1. *Let* $\texttt{s}_0 = \texttt{nil}$. *All entries in* Ξ' *are unmarked.*

2. *Take any unmarked entry* $\langle t, L, U \rangle$ *of* Ξ' *such that either* L *or* U *or both are not empty. Then build an e.s.* `e` *in the following way:*

   (a) *If both* L *or* U *are not empty, then let* D *be a type constructor that is a sup for the set* $\texttt{constructors}(L(t))$ *and an inf for* $\texttt{constructors}(U(t))$. *Let* $k \geq 0$ *be the arity of* D *and let* $v_1, \ldots, v_k$ *be the variables occurring in the type expressions* $D'(v_1, \ldots, v_h)$ *in* $L \cup U$ *(note that they must be the same for all elements of* $L \cup U$*). Then take* $\texttt{e} = [t := D(v_1, \ldots, v_k)]$.

(b) If L is empty but $\mathtt{constructors}(U)$ *has an inf D (possibly a lub) different from $\bot$ or U is empty but* $\mathtt{constructors}(L)$ *has a sup (possibly a glb) D different from ω define* $\mathtt{e}$ *as in case 2a.*

(c) If L is empty but $\mathtt{constructors}(U)$ *does not have an inf different from $\bot$, then define $\mathtt{e} = [t := \bot]$. Similarly if U is empty but the set* $\mathtt{constructors}(L)$ *has no sup different from ω, then define $\mathtt{e} = [t := \omega]$.*

3. Then append $\mathtt{e}$ *at the end of* $\mathtt{s}_i$ *and mark the entry of t. Let* $\mathtt{s}_{i+1}$ *be the p.e. obtained this way.*

4. As long as there are unmarked entries in Ξ', return to step 2.

5. Let n the last value of i. Return $\langle \mathtt{s}_n, \rho \rangle$.

(ii) Finally, $\mathtt{get}(\Xi)$ *returns the pair $\langle \sigma_{\mathtt{s}} \circ \rho, \Sigma_{\mathtt{s}} \rangle$.*

Note that the output of $\mathtt{get}$ can have several different forms, depending on the order in which the function $\mathtt{get}'$ chooses the e.s to be appended to the substitution path. In particular, different orders can result in a different shape of Σ. It is possible to prove, however, that these different sets of equations have semantically equivalent solutions.

The main property of this construction is the following.

Lemma 3.16 *Let* $\mathtt{get}(\Xi) = \langle \sigma, \Sigma \rangle$. *Then for all (ground) substitution γ, $\langle \gamma \circ \sigma, \gamma(\Sigma) \rangle$ is a solution of Ξ.*

Then $\mathtt{get}(\Xi)$ is a solution scheme for Ξ.

3.3 ω reductions

In order to take into account the non–homogeneous nature of the type assignment system (the ω type has a somewhat particular behaviour), we equip the system with a *reduction rule* for tables, that non–deterministically generates all the solutions of a given table.

Indeed, at any stage of the reduction process, we have to face the fact that by using the type ω and its associated properties and rules, we can assign different and in general incomparable types to a term. We will define later an heuristic which corresponds to a reasonable strategy for applying the ω-reductions.

The reduction of tables is written

$$\Theta \Rightarrow_R \Theta' ,$$

where Θ, Θ' are tables. A step in the reduction process consists in replacing a variable by ω:

Definition 3.17 *If Θ is a table and t a variable occurring in some guards of Θ such that $U_\Theta(t)$ is empty (or contains only ω), then $\mathtt{red}_\omega(\Theta, t)$ is the table obtained by applying to Θ the following steps.*

1. *Eliminate from Θ all the g.e.s that have an occurrence of t in their guard.*
2. *Set both the upper and lower set of t to $\{\omega\}$.*
3. *Apply the function* `closure` *to the resulting table.*

The application of `closure` in step 3. plays the rôle of propagating ω in the table.

The notion of reduction for tables is defined by the rules of Figure 3. Note in particular that the reduction step can be applied with any variable in $\mathtt{dom}(\Theta')$.

$$(ax1) \quad \Theta \Rightarrow_R \Theta$$

$$(\omega - red) \quad \frac{\Theta \Rightarrow_R \Theta' \qquad t \in \mathtt{dom}(\Theta')}{\Theta \Rightarrow_R \mathtt{red}_\omega(\Theta', t)}$$

Figure 3: Table reduction

Note as well that for some expressions, infinitely many non-comparable typings are possible. An example is given by $\lambda x.(x\ x)$ which has (among others) types $(\omega{\to}\omega){\to}\omega$, $(\omega{\to}\omega{\to}\omega){\to}\omega{\to}\omega$, $(\omega{\to}\omega{\to}\omega{\to}\omega){\to}\omega{\to}\omega{\to}\omega$ etc. (see [KPS94]). When this happens, all these typings involve ω and therefore are not extremely interesting. We shall define in Section 4.4 an heuristic to make the choice between them deterministic.

Note that if $\Theta \Rightarrow_R \Theta'$, then, for all variables $t \in \mathtt{dom}(\Theta)$, if $t \in \mathtt{dom}(\Theta')$ then $L_{\Theta'}(t) = L_\Theta(t)$ and $U_{\Theta'}(t) = U_\Theta(t)$, i.e. the reduction procedure does neither add nor erase entries from the table.

A basic property of table reduction, which is immediately proved, is the following.

Lemma 3.18 *Let $\Theta \Rightarrow_R \Theta'$. Then any strong solution of Θ' is a solution of Θ.*

3.4 Operators on tables

We will need in the following a couple of operators to handle tables.

Definition 3.19 *(i) If Θ_1 and Θ_2 are two (closed) tables, then $\Theta_1 \uplus \Theta_2$ is the table defined by merging them and applying* `closure`.

(ii) If g is an elementary g.c. and Θ is a table, then `addtable`(Θ, g) *is the table obtained from Θ by adding the constraints in g and applying* `closure` *to the resulting table.*

Definition 3.20 *If Θ is a table and w a guard, then $w \triangleright \Theta$ is the table obtained by replacing the guard w' of each g.e. occurring in the L and U sets of Θ by $w \triangleright w'$.*

Let Θ be a table and $\mathcal{V}$ a set of type variables. The function `simplify` extracts from a table Θ the subpart of it which is relevant for finding the solution relative to the variables in $\mathcal{V}$. In particular

$$\texttt{simplify}(\Theta, \mathcal{V})$$

is the table Θ' obtained from the empty table through the following steps:

1. Put in Θ' all the entries for variables in $\mathcal{V}$.
2. Add to Θ' all the entries of variables which occur in the upper or lower sets of variables already in Θ'.
3. Repeat step 2. until no other entry can be added to Θ'.

It is easy to see that if Θ is a closed table, so is the case for `simplify`$(\Theta, \mathcal{V})$ as well.

Moreover, the basic property of `simplify`(,i)s the following.

Lemma 3.21 *Let $\Theta' =$* `simplify`$(\Theta, \mathcal{V})$. *Then all solutions of Θ' can be extended in a solution of Θ.*

4 Type inference

4.1 From terms to tables

Our type inference method is defined by a set of rules in Natural Semantics through a judgment of the form

$$M \Rightarrow_{TI} \Gamma \mid t \mid \Theta$$

where t is a variable, Γ a typing context and Θ a *consistent* table. The definition of judgment $\Rightarrow_{TI} \mid \mid$ involves the application of the reduction relation $\Rightarrow_R$ in a nondeterministic way. This is essential to have a complete inference procedure, owing to the fact that a notion of principal type scheme does not exists in our system. We will define later a heuristic to avoid nondeterminism producing a quite general typing procedure. Indeed, as it will be shown by examples, the cases in which the use of nondeterministic reduction is needed are rather rare.

Informally, type constraints will be brought along in the computation and progressively updated as we get new information about the term, thus insuring the incrementality of our approach.

In a statement $M \Rightarrow_{TI} \Gamma \mid t \mid \Theta$, one should interpret the type parameters involved in the set of type assignment Γ as *meta variables*, rather than as actual type expressions. The intended meaning of such a judgment is indeed to define the *type schema* of all possible typings that are inferable for M.

4.2 Inference Framework

For technical reasons, in the inference rules a *context* is a set Δ of statements of the shape $x : t$ where x is a term variable and t a type variable. We define on contexts an auxiliary function, called `fondi`, to merge the assignment contexts. In particular

$$\texttt{fondi}(\Delta_1, \Delta_2) =< \Delta, \rho >$$

Where ρ is the trivial substitution (which is only a variable renaming) which ientifies all (and only) the type variables which are predicates of the same term variable in Δ_1 and Δ_2 and $\Delta = \rho(\Delta_1) \cup \rho(\Delta_2)$. We will use `fondi` (which is associative) also with more than two arguments.

We give first a set of natural deduction rules defining a nondeterministic and terminating inference procedure that is complete with respect to the inference rules.

Definition 4.1 (Canonical table for datatypes constructors)
Consider a datatype D, with its parameters $X_1, \ldots, X_k$. Recall (Section 2) that a datatype constructor c_i^D (we shall abbreviate it simply to c) is defined by arg L, where L is a list of type expressions (see corresponding Definition). We define $\Theta^{can}(c)$, called the canonical table *associated to constructor c, as follows:*

- *let $u_1, \ldots, u_k$ be fresh variables, we let $\sigma = \{u_1/X_1, \ldots, u_k/X_k\}$, and define Θ_{param} as the table consisting in the k rows of the form $u_j|\emptyset|\emptyset$, $1 \leq j \leq k$;*
- *$\Theta^{can}(c)$ is then defined by recursion over the list L of "arguments" of c:*
  - *if $L = []$, take a fresh variable v; $\Theta^{can}(c)$ is then equal to*

    $$\texttt{addtable}(\Theta_{param}, \{v :_G (D[u_1, \ldots, u_k] \leq v)\}),$$

    and the "output type variable" (associated to constructor c) is v;
  - *if $L = x :: L'$, compute Θ, the canonical table associated to L', with its corresponding output type variable t, and distinguish two cases, according to the shape of x:*
    - *if x is X_p for some p, then let v be a fresh variable; the canonical table is then equal to*

      $$\texttt{addtable}(v \triangleright \Theta, \{v :_G (u_p \rightarrow t \leq v)\}),$$

      the output type variable being v;
    - *otherwise, let v and v' be two fresh variables, the canonical table is*

      $$\texttt{addtable}(v' \triangleright \Theta, \{v' :_G (v \rightarrow t \leq v'), [v'; v; t] :_G (v \leq \sigma(x))\}),$$

      and the output variable is v'.

Note that by definition, $\Theta^{can}(c)$ is already closed and consistent (consistency is insured by the fact that there is no application).

$\Theta^{can}(c)$ is a *table schema* rather than a simple table: we shall indeed instanciate its type variables with newly created variables each time we shall encounter term c in the type inference procedure. To evidenciate the dependency towards these variables, we will sometimes adopt the notation $\Theta^{can}(c)[u_1, \ldots, u_k, v_1, \ldots, v_m, t]$, t being the "output type variable" of $\Theta^{can}(c)$ (and $v_1, \ldots, v_m$ the type variables introduced during the analysis of L described above).

Example: consider the list constructor *cons*, of type $X \rightarrow list[X] \rightarrow list[X]$; its associated canonical table is

	u_1	
$t : (u_1 \rightarrow v_1)$	t	
$t, v_1 : (t_2 \rightarrow v_2)$	v_1	
	t_2	$t, v_1, v_2, t_2 : list[u_1]$
$t, v_1, v_2 : list[u_1]$	v_2	

Note that variables t_1 and u_1 collapsed, and we only keep u_1. Here are the scopes of the type variables:

$$cons \ : \ \underbrace{\underbrace{X}_{u_1} \rightarrow \overbrace{\underbrace{list[X]}_{t_2} \rightarrow \overbrace{list[X]}^{v_2}}^{v_1}}_{t} \quad D = list[X],\ u_1 = X\ .$$

We are now ready to define our type inference procedure, given by the rules of Figure 4. Note that this procedure is still nondeterministic.

$$(T_{var})\ x \Rightarrow_{TI} \{x : t\} \mid t \mid \emptyset$$

$$(T_{const})\ c_i^D \Rightarrow_{TI} \emptyset \mid t \mid \Theta^{can}(c_i^D)[u_1, \ldots, u_k, v_1, \ldots, v_{m_i}, t]$$

where $\Theta^{can}(c)$ is the canonical table associated to c_i^D
$t, u_1, \ldots, u_k, v_1, \ldots, v_{m_i}, t$ are fresh variables

$$(T_\lambda)\ \frac{M \Rightarrow_{TI} \Delta, x : u \mid v \mid \Theta}{\lambda x.\, M \Rightarrow_{TI} \Delta \mid t \mid \texttt{addtable}(t \triangleright \Theta,\ \{t :_G (u \rightarrow v \leq t)\})}$$

where u is a fresh variable

$$(T_{app})\ \frac{M \Rightarrow_{TI} \Delta_1 \mid u \mid \Theta_1 \quad N \Rightarrow_{TI} \Delta_2 \mid v \mid \Theta_2 \quad \texttt{fondi}(\Delta_1, \Delta_2) =< \Delta, \rho >}{M\ N \Rightarrow_{TI} \rho(\Delta) \mid t \mid \texttt{simplify}(\Theta', t)}$$

where t is a fresh variable,
$\Theta = \texttt{addtable}(\rho(\Theta_1) \uplus \rho(\Theta_2), \{t :_G (\rho(u) \leq \rho(v) \rightarrow t)\})$,
$t \triangleright \Theta \Rightarrow_R \Theta'$, and Θ' is consistent

Figure 4: Type inference procedure

Note that rule (T_{app}) is the only case in which we can reduce the size of the table by applying `simplify`.

Out inference procedure is correct and complete with respect to the typing rules in the following sense.

Lemma 4.2 $\Gamma, \Sigma \vdash M : T$ *iff* $M \Rightarrow_{TI} \Delta \mid t \mid \Theta$ *and* $\langle \sigma, \Sigma' \rangle$ *is a solution of* Θ *such that* $T = \sigma(t)$, Σ *is equivalent to an extension of* Σ' *and* Γ *extends* $\sigma(\Delta)$

4.3 Typing definitions

We apply the type inference procedure that we just defined in order to progressively build a table that goes along with a sequence of CuCh definitions. This is expressed by a judgment of the form

$$\Delta, \Theta \xrightarrow{<eqs>} \Delta', \Theta' ,$$

where $< eqs >$ is a sequence of term definitions of the form $< \mathtt{f}\ (c\, x_1 \ldots x_n) = e >$. The above statement means that adding CuCh definitions $< eqs >$ to a context Δ and a table Θ yields to context Δ' and table Θ'. The rules that define this judgment are given on Figure 5.

$$(TD_{\mathtt{f}})\ \frac{\begin{array}{l} e \Rightarrow_{TI} \Delta_1, x_1 : t_1 \ldots x_n : t_n, f : v \mid t \mid \Theta_1 \\ f\,(c\, x_1 \ldots x_n) \Rightarrow_{TI} \Delta_2, x_1 : \delta_1 \ldots x :_n, f : u : \delta_n \mid t' \mid \Theta_2 \\ \mathtt{fondi}((\Delta_1, x_1 : t_1, \ldots x_n : t_n, f : v), \\ \qquad (\Delta_2, x_1 : t_1, \ldots, x_n : t_n, f : u),\ \Delta) \ = < \Delta', \rho > \end{array}}{\Delta, \Theta \xrightarrow{<eqs>.<\mathtt{f}\,(c\, x_1 \ldots x_n) \,=\, e>} \Delta'.\mathtt{f} : t, \Theta'} \quad \mathtt{f} \notin dom(\Delta_0)$$

where $\mathtt{addtable}((\rho(t' \triangleright \Theta_1) \uplus \rho(\Theta_2) \uplus \rho(\Theta)), \{t' :_G (t \leq t')\}) \Rightarrow_R \Theta'$
and Θ' is consistent

$$(TD_{comp})\ \frac{\Delta, \Theta \xrightarrow{<eqs>} \Delta', \Theta' \quad \Delta', \Theta' \xrightarrow{<eqs'>} \Delta'', \Theta''}{\Delta, \Theta \xrightarrow{<eqs>.<eqs'>} \Delta'', \Theta''}$$

Figure 5: Building a CuCh context

We have the following soundness and completeness result.

Lemma 4.3 *Let* $< eqs >$ *be a sequence of CuCh definitions. Let* $\emptyset, \emptyset \xrightarrow{<eqs>} \Delta, \Theta$ *and let* $\langle \sigma_g, \Sigma \rangle$ *be any solution of* Θ. *Then* $\Sigma : \emptyset, < eqs > \Rightarrow \sigma_g(\Delta)$.

Conversely if $\Sigma \ : \ \emptyset, < eqs > \Rightarrow \Gamma$ *then* $\emptyset, \emptyset \xrightarrow{<eqs>} \Delta, \Theta$ *and there is a solution* $\langle \sigma_g, \Sigma_0 \rangle$ *of* Θ *such that* $\Gamma = \sigma_g(\Delta)$ *and* Σ *is equivalent to an extension of* Σ_0.

4.4 Heuristic

The relation $\Rightarrow_{TI} \ | \ |$ as defined in Subsection 4.2 is not deterministic, due to the presence of ω-reductions, but we are interested in turning it into a deterministic process, in order to get a reasonably efficient implementation of the inference process. Of course, we do this at the cost of losing the completeness of the inference procedure.

We present here our heuristic to transform a (closed) table that is not consistent into a consistent one. The idea is to apply rule $(\omega - red)$ to eliminate the constraints on variables with respect to which the table is not consistent. This actually means simulating an application of rule (*omega*) to the subterms for which we are not able to find a meaningful type. Since we want to preserve as much information as possible, our strategy is to try to apply rule (*omega*) starting from the inner subterms. It is intuitive that the deeper the subterm of a given term that is assigned type ω by rule (*omega*), the more information shall be contained in the resulting type of the term.

We do this using the notion of guard. For a non-empty guard, we use the notation $w.u$ to isolate the last element u of the list (i.e. the innermost one).

Note that in the construction of the table each type variable introduced in the inference procedure corresponds to a subterm of the expression to be typed. *We assume that in the implementation of the inference procedure the guards are kept topologically sorted with respect to the inclusion of the corresponding subterms. I.e. if* u *is the type variable corresponding to a subexpression of a subterm to which corresponds a type variable* v *then in every guard containing both* u *and* v*, the occurrence of* u *follows the one of* v. This can be naturally achieved simply by listing the variables in the order in which they have been created, but more sophisticated ordering could be possible.

We formalize our heuristic under the form of a reduction relation $\Rightarrow_D$, defined on Figure 6.

We apply this transformation whenever the inference process yields an inconsistent table.

$$(D_{id}) \quad \Theta \Rightarrow_D \Theta$$

$$(D_U) \quad \frac{\begin{array}{l}\Theta_0 \Rightarrow_D \Theta \quad U_\Theta(t) \text{ not consistent} \\ w.u{:}_G\,(D(v_1,\ldots,v_k)) \in U_\Theta(t) \\ U(u) = \emptyset\end{array}}{\Theta_0 \Rightarrow_D \mathtt{red}_\omega(\Theta, u)}$$

$$(D_L) \quad \frac{\begin{array}{l}\Theta_0 \Rightarrow_D \Theta \quad U_\Theta(t) \text{ consistent} \\ \Theta \text{ not consistent with respect to } t \\ w.u{:}_G\,(D(v_1,\ldots,v_k)) \in L_\Theta(t)\end{array}}{\Theta_0 \Rightarrow_D \mathtt{red}_\omega(\Theta, u)}$$

Figure 6: Deterministic reduction relation

5 Conclusion

We have presented a partial type inference procedure for a language equipped with datatypes. The effectiveness of our methods have made it possible to develop an implementation, that could in principle be smoothly integrated to the CuCh machine, due to the incrementality of the inference.

This report presents a preliminary version of our work on this subject. We indeed plan to extend it in order to include a denotational description for our language of types, as well as possibly improve our heuristics, and of course make the implementation available together as part of the CuCh machine. It seems also that most of the work on the implementation could be reused to achieve *dead–code analysis* (see e.g. [Dam98]).

References

[BB85] C. Böhm and A. Berarducci. Automatic Synthesis of Typed λ–programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BPG] C. Böhm, A. Piperno, and S. Guerrini. λ–definition of Function(al)s by Normal Forms.

[CC90] F. Cardone and M. Coppo. Two Extensions of Curry's Inference System. In P. Odifreddi, editor, *Logic and Computer Science*, pages 19–75. Accademic Press, 1990.

[Dam98] F. Damiani. Redundant–code detection and elimination for PCF with algebraic Datatypes. In *To appear in the Proceedings of TLCA '99*, LNCS, 1998.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[Gom90] C. Gomard. Partial Type Inference for Untyped Functional Programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 282–287, 1990.

[KPS94] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994. also in Proceedings of FOCS'92, pages 363–371.

[Tha94] S. Thatte. Type inference with partial types. *TCS*, 124:127–148, 1994. also in Proceedings of ICALP '88, pages 615–629.

[WO92] Mitchell Wand and Patrick M. O'Keefe. Type inference for partial types is decidable. In B. Krieg-Brückner, editor, *European Symposium on Programming '92*, volume 582 of *LNCS*, pages 408–417. Springer Verlag, 1992.

A A detailed example

We illustrate the way our procedure works on an example, that shows the building of the table and the treatment of what would be considered as typing errors in a classical setting.

This example is treated with an implementation of the algorithms we describe (a WWW interface to the implementation is available at

`http://cermics.enpc.fr/~dh/Types3/`).

The user has the opportunity to incrementally introduce definitions of terms, and the system answers by showing the table that is constructed by the type inference procedure starting with this terms, and its evolution (as the closure function is applied). The initial environment contains several constants, such as elements of types `bool`, `int`, `float`, lists, and a few constant functions of types `int`→`bool`, `int`→`int`, etc.

For the moment, we only have the implementation of the type inference procedure (table construction), and of the closure function. The generation of solutions and the heuristic to resolve inconsistencies are in beta version, and we leave their presentation to a later presentation of this work.

We shall work with the term

$$M \;=\; \lambda x\, y.y\ (x\, 3)\ (x\, x)\,.$$

The idea here is to use x both in an auto–application and as function on integers, to force an typing conflict in the table that is generated (not leading to an inconsistency, though).

The ascii translation of this term is `\x y.((y (x 3)) (x x))`; the type inference procedure yields the following table:

```
< i | {[i]. b->h} | {} >
< h | {[i;h]. a->g} | {} >
< g | {} | {} >
< f | {} | {} >
< e | {} | {[i;h;g]. f->g} >
< d | {} | {} >
< c | {[i;h;g;e;d]. INT} | {} >
< b | {} | {[i;h;g;f]. b->f;[i;h;g;e;d]. c->d} >
< a | {} | {[i;h;g;e]. d->e} >
```

Each line corresponds to a row in the table, and for each row, we successively give the corresponding type variable, and its sets L and U. For example, we can see that type variable a has an empty L set, and has the guarded expression $[i;h;g;e]:_G d{\to}e$ in its U set.

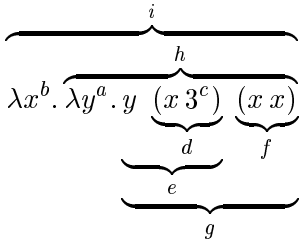
The system also gives some extra information on the typing procedure:

```
Current context is:

M:i
To help debugging, here are the "old" variable assumptions

x:b /  y:a
```

This means that to term variables `x` and `y` have been associated type variables `b` and `a` respectively. Using this information, and the typing rules of Section 2, we can reconstruct the structure of the term, by establishing a correspondence between the type variables and every subterm of the term M, as follows:



The steps in the construction of this decoration of the type can be illustrated on an example: $y\ (x\,3)$ being of type e, we read in the row corresponding to e in the table that e is less than $f{\to}g$ (omitting the guards); $(x\,x)$ being effectively of type f, we get that $y\ (x\,3)\ (x\,x)$ is of type g.

As the schema above shows, we can reconstruct the type of the whole expression by looking at the row of i (note as well that type variable i guards all the expressions of the table, which means that if we decide to put the whole term to type ω, there is no use to keep any information that is contained in the table). We read in the row of i that i is "equal to" $b{\to}h$ (i.e. without other constraints, i will be replaced by $b{\to}h$). The row for h tells us that h is $a{\to}g$, and we also see that b is less than $b{\to}f$ and than $c{\to}d$ (the constraint "$b{\to}f$" comes from the auto–application, hence the occurrence of b in the expression, while $c{\to}d$ comes from the application to constant 3: c is indeed "equal to" `INT`): we are facing a problem, since there is indeed a conflict between `INT` and the arrow construct. However, we cannot detect immediately this conflict on the table, due to the shape of M.

The system applies next the closure rules; the previous table being already closed, we get the same table:

```
The closed table is:

  < i | {[i]. b->h} | {} >
  < h | {[i;h]. a->g} | {} >
  < g | {} | {} >
  < f | {} | {} >
  < e | {} | {[i;h;g]. f->g} >
  < d | {} | {} >
  < c | {[i;h;g;e;d]. INT} | {} >
  < b | {} | {[i;h;g;f]. b->f;[i;h;g;e;d]. c->d} >
  < a | {} | {[i;h;g;e]. d->e} >
```

As said before, the conflicts coming from the "non–standard" use of x in M are not visible on the table above. We can evidenciate them by *applying*

M, which will have more or less the effect of instiantiating x, thus bringing to light the typing information about x.

We first apply M to the identity; the treatment of type $M\,(\lambda z.\,z)$ yields table:

```
< l | {} | {} >
< k | {[l;k]. j->j} | {} >
< j | {} | {} >
< i | {[l;i]. b->h} | {[l]. k->l} >
< h | {[l;i;h]. a->g} | {} >
< g | {} | {} >
< f | {} | {} >
< e | {} | {[l;i;h;g]. f->g} >
< d | {} | {} >
< c | {[l;i;h;g;e;d]. INT} | {} >
< b | {} | {[l;i;h;g;f]. b->f;[l;i;h;g;e;d]. c->d} >
< a | {} | {[l;i;h;g;e]. d->e} >

To help debugging, here are the "old" variable assumptions

z:j /  x:b /  y:a
```

Still, no conflict can be seen on this table; but this changes if we apply the closing procedure:

```
The closed table is:

  < l | {[l;i;h]. a->g} | {} >
  < k | {[l;k]. j->j} | {[l;i;h;g;f]. b->f;[l;i;h;g;e;d]. c->d} >
* < j | {[l;i;h;g;f;k]. j->j;[l;i;h;g;e;d;k]. INT} | {} >
  < i | {[l;i]. b->h} | {[l]. k->l} >
  < h | {[l;i;h]. a->g} | {} >
  < g | {} | {} >
* < f |
 {[l;i;k;h;g;f]. j->j;[l;k;i;h;g;f]. j->j;
  [l;k;i;h;g;f;e;d]. INT;[l;i;k;h;g;f;e;d]. INT}
 | {} >
  < e | {} | {[l;i;h;g]. f->g} >
* < d |
 {[l;i;k;h;g;e;d;f]. j->j;[l;i;k;h;g;e;d]. INT;
  [l;k;i;h;g;e;d;f]. j->j;[l;k;i;h;g;e;d]. INT}
 | {} >
  < c | {[l;i;h;g;e;d]. INT} | {} >
  < b | {[l;i;k]. j->j} | {[l;i;h;g;f]. b->f;[l;i;h;g;e;d]. c->d} >
  < a | {} | {[l;i;h;g;e]. d->e} >
```

A star – * – evidenciates the rows where a conflict is apparent (between an arrow type and datatype `INT`): this is the case for type variables `d`, `f` and `j` (that intuitively correspond to the points where the information about the two different typings for x is "communicated"). However, the resulting table is still consistent, since the set U is empty for these entries (which means that we can put the corresponding variables to ω).

A possible type for M $(\lambda z.\, z)$ is then $(\omega{\to}j{\to}j){\to}g$ (setting d to ω has the effect of eliminating some constraints, thus resolving some conflicts).

If we apply $M\,I$ (the term we just considered) to the combinator K, i.e. we consider the term $(M\ I\ (\lambda uv.\, u))$, we get the following result:

```
The type inference procedure yields table:

< q | {} | {} >
< p | {[q;p]. m->o} | {} >
< n | {} | {} >
< o | {[q;p;o]. n->m} | {} >
< m | {} | {} >
< l | {} | {[q]. p->q} >
< k | {[q;l;k]. j->j} | {} >
< j | {} | {} >
< i | {[q;l;i]. b->h} | {[q;l]. k->l} >
< h | {[q;l;i;h]. a->g} | {} >
< g | {} | {} >
< f | {} | {} >
< e | {} | {[q;l;i;h;g]. f->g} >
< d | {} | {} >
< c | {[q;l;i;h;g;e;d]. INT} | {} >
< b | {} | {[q;l;i;h;g;f]. b->f;[q;l;i;h;g;e;d]. c->d} >
< a | {} | {[q;l;i;h;g;e]. d->e} >

To help debugging, here are the "old" variable assumptions

u:m /  v:n /  z:j /  x:b /  y:a
```

We omit here the resulting closed table, where type conflicts appear in L sets (resp. U sets) for rows whose U sets (resp. L sets) are empty, and hence do not lead to "real" inconsistencies.

We get some true inconsistencies if we apply a constant `f1`, of type `INT->INT`, to our term $(M\,I\,K)$; indeed, the resulting closed table is:

```
  < u | {[u;t]. INT} | {} >
```

```
  < t | {[u;t]. s->r} | {[u]. q->u} >
  < r | {[u;t]. INT} | {} >
* < s |
 {[u;t;i;h;l;q;p;g;d;o;k;c;f]. INT;[u;t;i;h;l;q;p;g;d;o;k;c]. j->j;
  [u;t;i;l;h;q;p;g;d;o;k;c;f]. INT;[u;t;i;l;h;q;p;g;d;o;k;c]. j->j;
  [u;t;i;h;l;q;p;o;g;d;k;c;f]. INT;[u;t;i;h;l;q;p;o;g;d;k;c]. j->j;
  [u;t;i;l;h;q;p;o;g;d;k;c;f]. INT;[u;t;i;l;h;q;p;o;g;d;k;c]. j->j}
 | {[u;t;r]. INT} >
* < q |
 {[i;h;l;q;p;g;d;o;k;c;f]. INT;[i;h;l;q;p;g;d;o;k;c]. j->j;
  [i;l;h;q;p;g;d;o;k;c;f]. INT;[i;l;h;q;p;g;d;o;k;c]. j->j;
  [i;h;l;q;p;o;g;d;k;c;f]. INT;[i;h;l;q;p;o;g;d;k;c]. j->j;
  [i;l;h;q;p;o;g;d;k;c;f]. INT;[i;l;h;q;p;o;g;d;k;c]. j->j}
 | {[u;t;r]. INT} >
  < p | {[q;p]. m->o} | {[i;l;q;h;g;d]. c->d;[q;i;l;h;g;d]. c->d} >
* < n |
 {[i;h;g;q;p;l;d;o;k;f]. INT;[i;h;g;q;p;l;d;o;k;f;c]. j->j;
  [i;h;g;l;q;p;d;o;k;f]. INT;[i;h;g;l;q;p;d;o;k;f;c]. j->j;
  [i;h;g;q;l;p;d;o;k;f]. INT;[i;h;g;q;l;p;d;o;k;f;c]. j->j;
  [q;p;i;l;h;g;d;o;k;f]. INT;[q;p;i;l;h;g;d;o;k;f;c]. j->j;
  [i;l;q;h;p;g;d;o;k;f]. INT;[i;l;q;h;p;g;d;o;k;f;c]. j->j;
  [q;i;l;h;p;g;d;o;k;f]. INT;[q;i;l;h;p;g;d;o;k;f;c]. j->j}
 | {} >
  < o | {[q;p;o]. n->m} |
 {[q;p;i;l;h;g;d]. f->g;[i;l;q;h;p;g;d]. f->g;[q;i;l;h;p;g;d]. f->g} >
* < m |
 {[i;l;q;h;g;d;p;k;c;f]. INT;[i;l;q;h;g;d;p;k;c]. j->j;
  [q;i;l;h;g;d;p;k;c;f]. INT;[q;i;l;h;g;d;p;k;c]. j->j;
  [i;h;g;d;l;q;p;k;c;f]. INT;[i;h;g;d;l;q;p;k;c]. j->j;
  [i;h;g;d;q;l;p;k;c;f]. INT;[i;h;g;d;q;l;p;k;c]. j->j}
 |
 {[q;p;o;i;l;h;g;d;u;t;r]. INT;[q;p;i;l;h;g;d;o;u;t;r]. INT;
  [i;l;q;h;p;g;d;o;u;t;r]. INT;[q;i;l;h;p;g;d;o;u;t;r]. INT}
 >
  < l | {[i;l;h]. a->g} | {[q]. p->q} >
* < k | {[l;k]. j->j} |
 {[l;i;h;g;d;c;k;q;p;o;u;t;r]. INT;[l;i;h;g;f]. e->f;[l;i;h;g;d;c]. b->c} >
* < j |
 {[l;i;h;g;f;k]. INT;[l;i;h;g;d;c;k]. j->j;
  [i;h;g;f;l;k]. INT;[i;h;g;d;c;l;k]. j->j}
 | {[l;i;k;h;g;d;c;q;p;o;u;t;r]. INT;[l;k;i;h;g;d;c;q;p;o;u;t;r]. INT} >
  < i | {[i]. b->h} | {[l]. k->l} >
  < h | {[i;h]. a->g} | {[i;l;q]. p->q} >
* < g |
 {[q;p;i;l;h;g;d;o;k;c;f]. INT;[q;p;i;l;h;g;d;o;k;c]. j->j;
  [i;l;q;h;p;g;d;o;k;c;f]. INT;[i;l;q;h;p;g;d;o;k;c]. j->j;
  [q;i;l;h;p;g;d;o;k;c;f]. INT;[q;i;l;h;p;g;d;o;k;c]. j->j;
  [q;p;o;i;l;h;g;d;k;c;f]. INT;[q;p;o;i;l;h;g;d;k;c]. j->j}
 | {[i;h;l;q;u;t;r]. INT;[i;l;h;q;u;t;r]. INT} >
```

```
* < f |
 {[l;i;k;h;g;f]. INT;[l;k;i;h;g;f]. INT;
  [l;k;i;h;g;f;d;c]. j->j;[l;i;k;h;g;f;d;c]. j->j}
 | {} >
```

Here the row for `s` (as well as the row for `q`, actually) is inconsistent, because of the conflict between the arrow type in the lower set and the datatype `INT` in the upper set.

As we did not implement the heuristics to perform the reduction automatically yet, we provide a feature to allow the user to put some variables to ω "by hand"; here a possible choice is to reduce with respect to type variable `f` (whose upper set is empty), since `f` occurs in the guard of type expression `j->j`, but not in the guard of `INT` (`f` actually corresponds to a term that is deeper than the terms corresponding to the type variables that occur in the guard for `INT`). We then recover a consistent table, through this simulation of an ω–reduction.