Type Inference for Metaslang

Alessandro Coglio

October 26, 2005

DRAFT; PLEASE DO NOT DISTRIBUTE

1 Introduction

The logic of Metaslang [1] includes the notion of well-typed expressions, defined in terms of inference rules. However, [1] does not provide a type-checking algorithm, i.e. an algorithm to build proofs that expressions are well-typed. In fact, such an algorithm cannot exist, because well-typedness proofs may involve theorems (e.g. to show that a subtype predicate holds), and theoremhood in Metaslang is undecidable.

Despite this undecidability barrier, it is possible to develop algorithms that build incomplete well-typedness proofs, in the sense that subproofs of certain judgements are missing. Such judgements represent theorems that must hold in order for the well-typedness proof to be complete. Given proofs for those theorems, the incomplete proof can be completed. This is the strategy implemented in Specware: specs are type-checked modulo proof obligations, which must be discharged by the user.

The abstract syntax in [1] includes explicit types for all bound variables, op instances, etc. It also assumes that all op names are distinct. Building well-typedness proofs in that abstract syntax is indeed type *checking*. However, [2] allows type information to be omitted. It also allows overloading. Therefore, in the implementation of Specware it is necessary to perform type *inference*, not just type checking, i.e. it is necessary to infer missing type information while building well-typedness proofs.

This document attempts to define a type inference algorithm for Metaslang. This document is separate from [1] to emphasize the distinction between the "pure" notion of well-typedness and the various and varying algorithms that can be used to build (incomplete) well-typedness proofs.

We start with a small subset of the Metaslang language, which should nonetheless capture salient features of type inference in full Metaslang. The subset covered by this document will grow to the point of covering full Metaslang.

1.1 Notation

The (meta-)logical notations $=$, $\forall$, $\exists$, $\wedge$, and $./.$ have the usual meaning.

The set-theoretic notations $\in$, $\emptyset$, $\{\ldots \mid \ldots\}$, $\{\ldots\}$, $\cup$, $\cap$, and $\subseteq$ have the usual meaning.

$\mathbf{N}$ is the set of natural numbers, i.e. $\{0, 1, 2, \ldots\}$.

If A and B are sets, $A - B$ is their difference, i.e. $\{a \in A \mid b \notin B\}$.

If A and B are sets, $A \times B$ is their cartesian product, i.e. $\{\langle a, b\rangle \mid a \in A \wedge b \in B\}$. This generalizes to $n > 2$ sets.

If A and B are sets, $A + B$ is their disjoint union, i.e. $\{\langle 0, a\rangle \mid a \in A\} \cup \{\langle 1, b\rangle \mid b \in B\}$. The "tags" 0 and 1 are always left implicit. This generalizes to $n > 2$ sets.

If A and B are sets, $A \xrightarrow{\text{p}} B$ is the set of all partial functions from A to B, i.e. $\{f \subseteq A \times B \mid \forall \langle a, b_1\rangle, \langle a, b_2\rangle \in f.\ b_1 = b_2\}$; $A \to B$ is the set of all total functions from A to B, i.e. $\{f \in A \xrightarrow{\text{p}} B \mid \forall a \in A.\ \exists b \in B.\ \langle a, b\rangle \in f\}$; $A \xrightarrow{\text{f}} B$ is the set of all finite functions from A to B, i.e. $\{f \in A \xrightarrow{\text{p}} B \mid f \text{ is a finite set}\}$; and $A \hookrightarrow B$ is the set of all total injective functions from A to B, i.e. $\{f \in A \to B \mid \forall \langle a_1, b\rangle, \langle a_2, b\rangle \in f.\ a_1 = a_2\}$.

If f is a function from A to B, $\mathcal{D}(f)$ is the domain of f, i.e. $\{a \in A \mid \exists b \in B.\ \langle a, b\rangle \in f\}$.

If f is a function and $a \in \mathcal{D}(f)$, $f(a)$ denotes the unique value such that $\langle a, f(a)\rangle \in f$.

We write $f : A \xrightarrow{\text{p}} B$, $f : A \to B$, $f : A \xrightarrow{\text{f}} B$, and $f : A \hookrightarrow B$ for $f \in A \xrightarrow{\text{p}} B$, $f \in A \to B$, $f \in A \xrightarrow{\text{f}} B$, and $f \in A \hookrightarrow B$, respectively.

If A is a set, $\mathcal{P}_\omega(A)$ is the set of all finite subsets of A, i.e. $\{S \subseteq A \mid S \text{ finite}\}$.

If A is a set, A^* is the set of all finite sequences of elements of A, i.e. $\{x_1, \ldots, x_n \mid x_1 \in A \wedge \ldots \wedge x_n \in A\}$; A^+, $A^{(*)}$, and $A^{(+)}$ are the subsets of A^* of non-empty sequences, sequences without repeated elements, and non-empty sequences without repeated elements, respectively. The empty sequence is written ϵ. A sequence $x_1, \ldots, x_n$ is often written $\overline{x}$, leaving n implicit. The length of a sequence s is written $|s|$. When a sequence is written where a set is expected, it stands for the set of its elements.

2 Syntax

2.1 Names

We postulate the existence of an infinite set of names

$$\mathcal{N}$$

2.2 Types and expressions

We inductively define the set of types as

$$\begin{aligned} \mathit{Type} = {} & \{\mathsf{Bool}\} \\ & + \{\tau[\overline{T}] \mid \tau \in \mathcal{N} \ \wedge \ \overline{T} \in \mathit{Type}^*\} \\ & + \{T_1 \to T_2 \mid T_1, T_2 \in \mathit{Type}\} \\ & + \{T|r \mid T \in \mathit{Type} \ \wedge \ r \in \mathit{Exp}\} \end{aligned}$$

where *Exp* is defined below.[1] We may write $\tau[\epsilon]$ as just τ.

We inductively define the set of expressions as

$$\begin{aligned} \mathit{Exp} = {} & \{o[\overline{T}] \mid o \in \mathcal{N} \ \wedge \ \overline{T} \in \mathit{Type}^*\} \\ & + \{e_1 \ e_2 \mid e_1, e_2 \in \mathit{Exp}\} \\ & + \{\lambda v{:}T.\ e \mid v \in \mathcal{N} \ \wedge \ T \in \mathit{Type} \ \wedge \ e \in \mathit{Exp}\} \end{aligned}$$

We may write $o[\epsilon]$ as just o.

Unlike [1], we do not have separate syntactic categories for (type) variables. The reason is that, for type inference, (type) variables should behave like monomorphic type and op names.[2]

2.3 Contexts

We define the set of context elements as

$$\begin{aligned} \mathit{CxElem} = {} & \{\mathsf{ty}\ \tau{:}n \mid \tau \in \mathcal{N} \ \wedge \ n \in \mathbf{N}\} \\ & + \{\mathsf{op}\ o{:}\{\overline{\beta}\}\ T \mid o \in \mathcal{N} \ \wedge \ \overline{\beta} \in \mathcal{N}^{(*)} \ \wedge \ T \in \mathit{Type}\} \end{aligned}$$

We may write $(\mathsf{ty}\ \tau{:}0)$ as just $(\mathsf{ty}\ \tau)$ and we may write $(\mathsf{op}\ o{:}\{\epsilon\}\ T)$ as just $(\mathsf{op}\ o{:}T)$.

We define the set of contexts as

$$\mathit{Cx} = \mathit{CxElem}^*$$

In other words, a context is a finite sequence of context elements. We may write $(\mathsf{ty}\ \beta_1, \ldots, \mathsf{ty}\ \beta_n)$ as just $(\mathsf{ty}\ \overline{\beta})$.

[1] Types depend on expressions, which depend on types. Thus, types and expressions are inductively defined together, not separately. Their definitions are presented separately only for readability.

[2] As it has been pointed out by Lindsay Errington, it should be possible to eliminate (type) variables as separate syntactic categories from [1] too. However, that requires a little more thought.

Op definitions and axioms are not included because they should be irrelevant to type inference. The fact that we do not include type definitions amounts to the assumption that all type definitions in Metaslang are fully expanded prior to type checking. In order for this expansion to be possible, we assume the absence of recursive type definitions in Metaslang. Even though [1] and [2] allow recursive type definitions, Metaslang is moving towards the direction of disallowing recursive types in favor of a PVS-like data type mechanism to achieve the same or a better effect as recursive type definitions.

2.4 Occurrences

The function $\mathcal{T} : \mathit{Type} + \mathit{Exp} \to \mathcal{P}_\omega(\mathcal{N})$ returns the monomorphic type names in a type or expression

$$\begin{array}{ll}
\mathcal{T}(\mathsf{Bool}) & = \emptyset \\
\mathcal{T}(\tau[\overline{T}]) & = \begin{cases} \{\tau\} & \text{if} \quad \overline{T} = \epsilon \\ \bigcup_i \mathcal{T}(T_i) & \text{otherwise} \end{cases} \\
\mathcal{T}(T_1 \to T_2) & = \mathcal{T}(T_1) \cup \mathcal{T}(T_2) \\
\mathcal{T}(T|r) & = \mathcal{T}(T) \cup \mathcal{T}(r)
\end{array}$$

$$\begin{array}{ll}
\mathcal{T}(o[\overline{T}]) & = \bigcup_i \mathcal{T}(T_i) \\
\mathcal{T}(e_1\ e_2) & = \mathcal{T}(e_1) \cup \mathcal{T}(e_2) \\
\mathcal{T}(\lambda v{:}T.\ e) & = \mathcal{T}(T) \cup \mathcal{T}(e)
\end{array}$$

The function $\mathcal{FO} : \mathit{Type} + \mathit{Exp} \to \mathcal{P}_\omega(\mathcal{N})$ returns the free monomorphic op names in a type or expression

$$\begin{array}{ll}
\mathcal{FO}(\mathsf{Bool}) & = \emptyset \\
\mathcal{FO}(\tau[\overline{T}]) & = \bigcup_i \mathcal{FO}(T_i) \\
\mathcal{FO}(T_1 \to T_2) & = \mathcal{FO}(T_1) \cup \mathcal{FO}(T_2) \\
\mathcal{FO}(T|r) & = \mathcal{FO}(T) \cup \mathcal{FO}(r)
\end{array}$$

$$\begin{array}{ll}
\mathcal{FO}(o[\overline{T}]) & = \bigcup_i \mathcal{FO}(T_i) \\
\mathcal{FO}(e_1\ e_2) & = \mathcal{FO}(e_1) \cup \mathcal{FO}(e_2) \\
\mathcal{FO}(\lambda v{:}T.\ e) & = \mathcal{FO}(T) \cup (\mathcal{FO}(e) - \{v\})
\end{array}$$

The function $\mathcal{DT} : \mathit{Cx} \to \mathcal{P}_\omega(\mathcal{N})$ returns the type names declared in a context

$$\begin{array}{ll}
\mathcal{DT}(\epsilon) & = \emptyset \\
\mathcal{DT}(\mathit{cxel}, \mathit{cx}) & = \begin{cases} \mathcal{DT}(\mathit{cx}) \cup \{\tau\} & \text{if} \quad \mathit{cxel} = \mathsf{ty}\ \tau{:}n \\ \mathcal{DT}(\mathit{cx}) & \text{otherwise} \end{cases}
\end{array}$$

The function $\mathcal{DO} : \mathit{Cx} \to \mathcal{P}_\omega(\mathcal{N})$ returns the op names declared in a context

$$\begin{array}{ll}
\mathcal{DO}(\epsilon) & = \emptyset \\
\mathcal{DO}(\mathit{cxel}, \mathit{cx}) & = \begin{cases} \mathcal{DO}(\mathit{cx}) \cup \{o\} & \text{if} \quad \mathit{cxel} = \mathsf{op}\ o{:}\{\overline{\beta}\}\ T \\ \mathcal{DO}(\mathit{cx}) & \text{otherwise} \end{cases}
\end{array}$$

2.5 Substitutions

The function $\_[\_] : (\mathit{Type}^* + \mathit{Exp}) \times (\mathcal{N} \xrightarrow{\text{f}} \mathit{Type}) \to \mathit{Type}^* + \mathit{Exp}$ substitutes each monomorphic type name $\beta \in \mathcal{D}(\sigma)$, where $\sigma : \mathcal{N} \xrightarrow{\text{f}} \mathit{Type}$, with the type $\sigma(\beta)$ in a type (sequence) or expression x (written $x[\sigma]$)

$$\begin{array}{rl}
\mathsf{Bool}[\sigma] & = \mathsf{Bool} \\
\tau[\overline{T}][\sigma] & = \begin{cases} \sigma(\tau[\overline{T}]) & \text{if} \quad \overline{T} = \epsilon\ \wedge\ \tau \in \mathcal{D}(\sigma) \\ \tau[\overline{T}[\sigma]] & \text{otherwise} \end{cases} \\
(T_1 \to T_2)[\sigma] & = T_1[\sigma] \to T_2[\sigma] \\
(T|r)[\sigma] & = T[\sigma]|r[\sigma]
\end{array}$$

$$(T_1, \ldots, T_n)[\sigma] = T_1[\sigma], \ldots, T_n[\sigma]$$

$$
\begin{aligned}
o[\overline{T}][\sigma] &= o[\overline{T}[\sigma]] \\
(e_1\ e_2)[\sigma] &= e_1[\sigma]\ e_2[\sigma] \\
(\lambda v{:}T.\ e)[\sigma] &= \lambda v{:}T[\sigma].\ e[\sigma]
\end{aligned}
$$

The function $\_[\_/\_] : (\mathit{Type}^* + \mathit{Exp}) \times \mathcal{N} \times \mathit{Exp} \to \mathit{Exp}$ substitutes a monomorphic op name u with an expression d in a type (sequence) or expression x (written $x[u/d]$)

$$
\begin{aligned}
\mathsf{Bool}[u/d] &= \mathsf{Bool} \\
\tau[\overline{T}][u/d] &= \tau[\overline{T}[u/d]] \\
(T_1 \to T_2)[u/d] &= T_1[u/d] \to T_2[u/d] \\
(T|r)[u/d] &= T[u/d]|r[u/d]
\end{aligned}
$$

$$
(T_1, \ldots, T_n)[u/d] = T_1[u/d], \ldots, T_n[u/d]
$$

$$
\begin{aligned}
o[\overline{T}][u/d] &= \begin{cases} d & \text{if} \quad \overline{T} = \epsilon \ \wedge \ o = u \\ o[\overline{T}[u/d]] & \text{otherwise} \end{cases} \\
(e_1\ e_2)[u/d] &= e_1[u/d]\ e_2[u/d] \\
(\lambda v{:}T.\ e)[u/d] &= \begin{cases} \lambda v{:}T[u/d].\ e & \text{if} \quad u = v \\ \lambda v{:}T[u/d].\ e[u/d] & \text{otherwise} \end{cases}
\end{aligned}
$$

The function $\mathcal{CO} : (\mathit{Type} + \mathit{Exp}) \times \mathcal{N} \to \mathcal{P}_\omega(\mathcal{N})$ returns the monomorphic op names that would be captured if a monomorphic op name u were substituted with those monomorphic op names in a type T or expression e (i.e. all the monomorphic op names bound in T or e at the free occurrences of u in T or e)

$$
\begin{aligned}
\mathcal{CO}(\mathsf{Bool}, u) &= \emptyset \\
\mathcal{CO}(\tau[\overline{T}], u) &= \textstyle\bigcup_i \mathcal{CO}(T_i, u) \\
\mathcal{CO}(T_1 \to T_2, u) &= \mathcal{CO}(T_1, u) \cup \mathcal{CO}(T_2, u) \\
\mathcal{CO}(T|r, u) &= \mathcal{CO}(T, u) \cup \mathcal{CO}(r, u)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{CO}(o[\overline{T}], u) &= \textstyle\bigcup_i \mathcal{CO}(T_i, u) \\
\mathcal{CO}(e_1\ e_2, u) &= \mathcal{CO}(e_1, u) \cup \mathcal{CO}(e_2, u) \\
\mathcal{CO}(\lambda v{:}T.\ e, u) &= \mathcal{CO}(T, u) \cup \begin{cases} \{v\} \cup \mathcal{CO}(e, u) & \text{if} \quad u \in \mathcal{FO}(e) - \{v\} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

The relation $\mathit{OKsbs} \subseteq (\mathit{Type} + \mathit{Exp}) \times \mathcal{N} \times \mathit{Exp}$ captures the condition that the substitution $x[u/d]$ causes no free variables in d to be captured

$$
\mathit{OKsbs}(x, u, d) \ \Leftrightarrow \ \mathcal{FV}(d) \cap \mathcal{CV}(x, u) = \emptyset
$$

3 Proof theory

We define a unary relation $\vdash \_ : \textsc{context} \subseteq \mathit{Cx}$ to capture well-formed contexts as

$$
\frac{}{\vdash \epsilon : \textsc{context}} \quad (\textsc{cxMT})
$$

$$
\frac{\begin{array}{c} \vdash cx : \textsc{context} \\ \tau \notin \mathcal{DT}(cx) \end{array}}{\vdash cx, \mathsf{ty}\ \tau{:}n : \textsc{context}} \quad (\textsc{cxTDec})
$$

$$
\frac{\begin{array}{c} \vdash cx : \textsc{context} \\ o \notin \mathcal{DO}(cx) \\ cx, \mathsf{ty}\ \overline{\beta} \vdash T : \textsc{type} \end{array}}{\vdash cx, \mathsf{op}\ o{:}\{\overline{\beta}\}\ T : \textsc{context}} \quad (\textsc{cxODec})
$$

We define a binary relation $\_ \vdash \_ : \textsc{type} \subseteq \mathit{Cx} \times \mathit{Type}$ to capture well-formed types as

$$\frac{\vdash cx : \textsc{context}}{cx \vdash \mathsf{Bool} : \textsc{type}} \quad (\textsc{tyBool})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \mathsf{ty}\ \tau{:}n \in cx \\ |\overline{T}| = n \\ \forall i.\ \ cx \vdash T_i : \textsc{type}\end{array}}{cx \vdash \tau[\overline{T}] : \textsc{type}} \quad (\textsc{tyInst})$$

$$\frac{\begin{array}{c}cx \vdash T_1 : \textsc{type} \\ cx \vdash T_2 : \textsc{type}\end{array}}{cx \vdash T_1 \to T_2 : \textsc{type}} \quad (\textsc{tyArr})$$

$$\frac{cx \vdash r : T \to \mathsf{Bool}}{cx \vdash T|r : \textsc{type}} \quad (\textsc{tyRestr})$$

We define a ternary relation $\_ \vdash \_ \prec \_ \subseteq \mathit{Cx} \times \mathit{Type} \times \mathit{Type}$ to capture subtyping as

$$\frac{cx \vdash T|r : \textsc{type}}{cx \vdash T|r \prec T} \quad (\textsc{stRestr})$$

$$\frac{cx \vdash T : \textsc{type}}{cx \vdash T \prec T} \quad (\textsc{stRefl})$$

$$\frac{\begin{array}{c}cx \vdash T : \textsc{type} \\ cx \vdash T_1 \prec T_2\end{array}}{cx \vdash T \to T_1 \prec T \to T_2} \quad (\textsc{stArr})$$

We do not need a transitivity rule for subtyping. As defined below, subtyping judgements are only used to assign types to expressions, e.g. to assign a supertype to an expression of a subtype. So, instead of using transitivity of subtyping, rules for well-typed expressions can be applied multiple times, achieving the same effect.

We define a ternary relation $\_ \vdash \_ : \_ \subseteq \mathit{Cx} \times \mathit{Exp} \times \mathit{Type}$ to capture well-typed expressions as

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \mathsf{op}\ o{:}\{\overline{\beta}\}\ T \in cx \\ \forall i.\ \ cx \vdash T_i : \textsc{type} \\ \sigma = \{\langle \beta_i, T_i \rangle \mid 1 \le i \le n\}\end{array}}{cx \vdash o[\overline{T}] : T[\sigma]} \quad (\textsc{exOp})$$

$$\frac{\begin{array}{c}cx \vdash e_1 : T_1 \to T_2 \\ cx \vdash e_2 : T_1\end{array}}{cx \vdash e_1\ e_2 : T_2} \quad (\textsc{exApp})$$

$$\frac{cx, \mathsf{op}\ v{:}T \vdash e : T'}{cx \vdash \lambda v{:}T.\ e : T \to T'} \quad (\textsc{exAbs})$$

$$\frac{\begin{array}{c}cx \vdash e : T \\ cx \vdash T \prec T'\end{array}}{cx \vdash e : T'} \quad (\textsc{exSuper})$$

$$\frac{cx \vdash e : T' \quad cx \vdash T \prec T'}{cx \vdash e : T} \quad (\text{exSub})$$

$$\frac{cx \vdash \lambda v{:}T.\ e : T' \quad v' \notin \mathcal{FO}(e) \cup \mathcal{CV}(e, v)}{cx \vdash \lambda v'{:}T.\ e[v/v'] : T'} \quad (\text{exAbsAlpha})$$

The proof theory of this subset of the Metaslang logic does not include theorems and the proof obligation $(r\ e)$ for rule exSub. However, the rule exSub itself, in contrast to rule exSuper, captures the existence (if not the exact form) of a proof obligation. Once a proof in this reduced proof theory has been built, it is easy to generate proof obligation with a pass through the proof.

4 The type inference problem

- Consider a well-formed context $cx \in Cx$. Well-formedness means that there is a proof of the judgement $\vdash cx : \text{CONTEXT}$.
- Consider an expression $e \in Exp$ such that e contains no monomorphic type names that are declared polymorphic in cx, i.e. such that if $\tau \in \mathcal{T}(e)$ then cx contains no declaration $(\mathsf{ty}\ \tau{:}n)$ with $n \neq 0$. This condition is very easy to check; it amounts to disallowing overloading of type names.
- Consider the monomorphic type names in e that are not declared in cx, i.e. the set $V = \mathcal{T}(e) - \mathcal{DT}(cx)$. The elements of V are *type variables*, i.e. placeholders for types that must be inferred in order to establish the well-typedness of e.
- The type inference problem is to find an assignment $\sigma : V \to Type$ and a type $T \in Type$ such that there is a proof of the judgement $cx \vdash e[\sigma] : T$.
- In general, there may be zero, one, or many such assignments σ. If there are zero, type inference fails and an error is signaled. If there is one, that is the solution. If there are many, either an error is signaled because of ambiguity, or one assignment is chosen according to some criterion (e.g. minimize proof obligations).

When the user enters Metaslang text in Specware according to the grammar in [2], the types that instantiate polymorphic ops are often missing and the types of bound variables are sometimes missing too. Those missing types must be inferred. In order to do that, we insert fresh monomorphic type names into all the spots where type information is missing; since these fresh names are not in the context (because they are fresh), those are exactly the type variables in V. The assignment σ, the type T, and the proof of $cx \vdash e[\sigma] : T$ witness the success of type inference.

For now we do not consider overloading resolution and infix op application resolution, which are two other important aspects of Metaslang type inference. We will tackle those after solving the simpler problem of type inference stated above, which is a necessary problem to solve in order to solve the type inference problem of full Metaslang.

5 A type inference algorithm

The proposed type inference algorithm solves the type inference problem by solving a set of constraints that are generated from the expression e and the context cx.

5.1 Constraints

We define the set of atomic constraints as

$$\begin{aligned} AC = &\{T_1 \approx T_2 \mid T_1, T_2 \in \mathit{Type}\} \\ &+ \{T_1 \prec T_2 \mid T_1, T_2 \in \mathit{Type}\} \end{aligned}$$

We define the set of conjunctive constraints as

$$\begin{aligned} CC = &\{\bot\} \\ &+ \{ac_1 \wedge \cdots \wedge ac_n \mid \overline{ac} \in AC^*\} \end{aligned}$$

If $n = 0$, we may write $\bigwedge_i ac_i$ as just $\top$.

We define the set of disjunctive constraints as

$$DC = \{cc_1 \vee cc_2 \mid cc_1, cc_2 \in CC\}$$

5.2 Constraint generation

We formalize the constraint generation process via a quaternary relation $\_;\_ \vdash \_ : \_ \subseteq \mathcal{P}_\omega(DC) \times \mathit{Cx} \times \mathit{Exp} \times \mathit{Type}$ defined as

$$\frac{\mathsf{op}\ o{:}\{\overline{\beta}\}\ T \in cx \qquad |\overline{\beta}| = |\overline{T}| \qquad \sigma = \{\langle \beta_i, T_i \rangle \mid 1 \le i \le n\}}{\emptyset; cx \vdash o[\overline{T}] : T[\sigma]}$$

$$\frac{v \notin \mathcal{DO}(cx) \qquad \widetilde{dc}; cx, \mathsf{op}\ v{:}T \vdash e : T'}{\widetilde{dc}; cx \vdash \lambda v{:}T.\ e : T \to T'}$$

$$\frac{v \in \mathcal{DO}(cx) \qquad v' \notin \mathcal{FO}(e) \cup \mathcal{CO}(e, v) \qquad \widetilde{dc}; cx \vdash \lambda v'{:}T.\ e[v/v'] : T'}{\widetilde{dc}; cx \vdash \lambda v{:}T.\ e : T'}$$

$$\frac{\widetilde{dc}_1; cx \vdash e_1 : T_\mathrm{D} \to T_\mathrm{R} \qquad \widetilde{dc}_2; cx \vdash e_2 : T_\mathrm{A} \qquad \widetilde{dc} = \widetilde{dc}_1 \cup \widetilde{dc}_2 \cup \{T_\mathrm{A} \prec T_\mathrm{D} \vee T_\mathrm{D} \prec T_\mathrm{A}\}}{\widetilde{dc}; cx \vdash e_1\ e_2 : T_\mathrm{R}}$$

The intended meaning of $\widetilde{dc}; cx \vdash e : T$ is that for all type variable assignments σ that satisfy all the constraints in $\widetilde{dc}$, it is the case that $cx \vdash e : T[\sigma]$. This formulation is similar to [3, 4].

Of course, the rules above define a computable function $\mathit{gen} : \mathit{Cx} \times \mathit{Exp} \to (\mathcal{P}_\omega(DC) \times \mathit{Type}) + \{\mathsf{fail}\}$ that, given a context and an expression, returns a type and a set of disjunctive constraints, unless it fails. This function realizes the first phase of the proposed type inference algorithm.

Note that we generate a disjunctive constraint for an application $(e_1\ e_2)$. The constraint says that either the type of the argument e_2 is a subtype of the domain type of the function e_1, or vice versa. The first case corresponds to the use of rule exSuper in the well-typedness proof of the application $(e_1\ e_2)$: if e_2 has type T_A, it also has type T_D by exSuper and then rule exApp can be applied. The second case corresponds instead to exSub: if e_2 has type T_A, then it also has type T_D provided that the associated proof obligation is satisfied. In order to minimize the number of proof obligations generated

by type inference, the constraint solving phase of the proposed type inference algorithm should try to satisfy the left disjuncts in preference to the right disjuncts.

Constraint generation and type inference fails if *gen* returns fail. In the implementation, if we associate to type variables some information about where those type variables occur (e.g. as the type of a certain abstraction within a certain expression), we might be able to generate more informative error messages than typical type inference algorithms for polymorphic functional languages.

5.3 Constraint solving

The constraint generation phase generates a set of disjunctive constraints where each disjunct is an atomic inequality constraint. We can visualize these constraints as a list of disjunctions

$$\begin{array}{c} ac_1 \vee ac'_1 \\ \vdots \\ ac_n \vee ac'_n \end{array}$$

There are 2^n possible sets of n inequalities each to be checked for satisfaction. Preference should be given to sets containing more left disjuncts than right disjuncts, in order to minimize the proof obligations generated.

Fortunately, there are constraint reduction rules that can be used to simplify the set of n disjunctive constraints prior to attempt the exponential search. These reduction rules operate on atomic constraints and return conjunctive constraints. They are captured by a binary relation $\longrightarrow \subseteq AC \times CC$ defined as

$$\begin{array}{rcl}
\mathsf{Bool} \prec T & \longrightarrow & \mathsf{Bool} \approx T \\
\tau[\overline{T}] \prec T & \longrightarrow & \tau[\overline{T}] \approx T \\
T_1 \to T_2 \prec T'_1 \to T'_2 & \longrightarrow & T_1 \approx T'_1 \wedge T_2 \prec T'_2 \\
T_1 \to T_2 \prec T & \longrightarrow & \bot \\
T|r \prec T'|r & \longrightarrow & T \approx T' \\
T|r \prec T' & \longrightarrow & T \prec T' \\
\mathsf{Bool} \approx \mathsf{Bool} & \longrightarrow & \top \\
\mathsf{Bool} \approx T & \longrightarrow & \bot \quad \text{where } T \notin V \\
\tau[\overline{T}] \approx \tau[\overline{T}'] & \longrightarrow & \bigwedge_i T_i \approx T'_i \\
\tau[\overline{T}] \approx T & \longrightarrow & \bot \quad \text{where } T \notin V \\
T_1 \to T_2 \approx T'_1 \to T'_2 & \longrightarrow & T_1 \approx T'_1 \wedge T_2 \approx T'_2 \\
T_1 \to T_2 \approx T & \longrightarrow & \bot \quad \text{where } T \notin V \\
T|r \approx T'|r & \longrightarrow & T \approx T' \\
T|r \approx T' & \longrightarrow & \bot \quad \text{where } T \notin V
\end{array}$$

The order of the above reduction rules is important: each rule applies only if the previous ones do not. Of course, atomic constraints that result from a reduction in the rules above should be interpreted as singleton conjunctive constraints.

The above reduction rules can be applied exhaustively to the generated constraints. Each disjunct in the list of disjunctive constraints will become a conjunctive constraint whose atomic constraints have one of the forms

- $\alpha \approx T$
- $T \approx \alpha$
- $\alpha \prec T$
- $T \prec \alpha$

where $\alpha \in V$. Of course, some conjunctive constraints may be $\top$ or $\bot$.

If a disjunctive constraint has the form $(\bot \vee \bot)$, type inference fails. If a disjunctive constraint has the form $(\bot \vee cc)$ or $(cc \vee \bot)$, with $cc \neq \bot$, then the disjunctive constraint can be reduced to the conjunctive constraint cc; eliminating disjunctive constraints helps reduce the exponential complexity

of the algorithm. If a disjunctive constraint has $\top$ as one of its disjuncts, the constraint can be eliminated altogether.

An atomic equality constraint $(\alpha \approx T)$ or $(T \approx \alpha)$ that does not appear in a disjunctive constraint (because one of the disjuncts has been eliminated) can be eliminated by replacing α with T in all other constraints. When this is done, the reduction rules given earlier need to be applied on the residual constraints.

We can proceed in that way until either a failure occurs or we end up with a list of atomic inequality constraints (obtained by flattening the conjunctive constraints) plus a list of disjunctive constraints

$$\begin{array}{c} ac_1 \\ \vdots \\ ac_n \\ cc_1 \vee cc'_1 \\ \vdots \\ cc_m \vee cc'_m \end{array}$$

At this point, one can consider all 2^m combinations of conjunctive constraints. For each of them, we do the following.

First, all equalities are eliminated by replacing the type variables with the types that the constraints equate them to, as described earlier. Again, the reduction rules are exhaustively applied after each elimination. The result of this process is a set of inequalities (to be taken conjunctively) of the form $\alpha \prec T$ or $T \prec \alpha$. If both constraints $\alpha \prec T$ and $T \prec \alpha$ are present, they can be replaced by $\alpha \approx T$ (or, equivalently, $T \approx \alpha$) and eliminated according to the usual replacement method.

[[[TO BE CONTINUED]]]

[[[NEED TO CHECK FOR CIRCULAR CONSTRAINTS]]]

References

[1] Alessandro Coglio. The logic of Metaslang. Available in the Specware development directory.

[2] Kestrel Institute and Kestrel Technology LLC. *Specware 4.1 Language Manual*. Available at www.specware.org.

[3] Alex Aiken and Edward Wimmers. Type inclusion constraints and type inference. In *Proc. 7th ACM Conference on Functional Programming and Computer Architecture*, pages 31–41, June 1993.

[4] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999.