Type Inference for Metaslang

Alessandro Coglio*

January 23, 2006

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. This Metaslang subset is defined in §2 and §3, where for brevity we omit the informal explanations that can be found in [1].

1.1 Notation

The (meta-)logical notations $=$, $\forall$, $\exists$, $\wedge$, $\Leftrightarrow$, $.\not.$ (e.g. $\neq$), **true**, and **false** 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. $\{x \in A \mid x \notin B\}$.

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.

*Thanks to Lambert Meertens for useful feedback and discussions.

$\{f \in A \xrightarrow{\text{p}} B \mid f \text{ is a finite set}\}$; 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, $\mathcal{P}(A)$ is the set of all subsets of A, i.e. $\{S \mid S \subseteq A\}$, and $\mathcal{P}_\omega(A)$ is the set of all finite subsets of A, i.e. $\{S \in \mathcal{P}(A) \mid S \text{ finite}\}$.

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.[1] 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

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

and the set of expressions as

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

We may write $\tau[\epsilon]$ as just τ and we may write $o[\epsilon]$ as just o.

2.3 Contexts

We define the set of context elements as

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

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

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

[1] Strictly speaking, our $x_1, \ldots, x_n$ notation for sequences may lead to ambiguities, e.g. if s_1 and s_2 are sequences, is s_1, s_2 the sequence of length 2 whose elements are s_1 and s_2, or is it the concatenation of s_1 and s_2? However, in this document, the intended meaning should be always clear from the symbols used and from mathematical context.

We may write $(\mathsf{tvar}\ \beta_1, \ldots, \mathsf{tvar}\ \beta_n)$ as just $(\mathsf{tvar}\ \overline{\beta})$.

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 for now.

2.4 Occurrences

The function $\mathcal{FV} : \mathit{Exp} \to \mathcal{P}_\omega(\mathcal{N})$ returns the free variables of an expression

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

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

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

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

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

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

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

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

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

$$\begin{array}{ll} \mathcal{TV}(\epsilon) & = \emptyset \\ \mathcal{TV}(\mathit{cxel}, \mathit{cx}) & = \begin{cases} \mathcal{TV}(\mathit{cx}) \cup \{\beta\} & \text{if} \quad \mathit{cxel} = \mathsf{tvar}\ \beta \\ \mathcal{TV}(\mathit{cx}) & \text{otherwise} \end{cases} \end{array}$$

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

$$\begin{array}{ll} \mathcal{V}(\epsilon) & = \emptyset \\ \mathcal{V}(\mathit{cxel}, \mathit{cx}) & = \begin{cases} \mathcal{V}(\mathit{cx}) \cup \{v\} & \text{if} \quad \mathit{cxel} = \mathsf{var}\ v{:}T \\ \mathcal{V}(\mathit{cx}) & \text{otherwise} \end{cases} \end{array}$$

2.5 Substitutions

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

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

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

where of course $(T_1, \ldots, T_n)[\sigma] = T_1[\sigma], \ldots, T_n[\sigma]$. Given $\overline{\beta} \in \mathcal{N}^{(*)}$ and $\overline{T} \in \mathit{Type}^*$ such that $|\overline{\beta}| = |\overline{T}|$, we may write $T[\{\langle \beta_i, T_i \rangle \mid 1 \le i \le n\}]$ as just $T[\overline{\beta}/\overline{T}]$.

The function $\_[\_/\_] : \mathit{Exp} \times \mathcal{N} \times \mathit{Exp} \to \mathit{Exp}$ substitutes a variable u with an expression d in a type (sequence) or expression e (written $e[u/d]$)

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

The function $\mathcal{CV} : \mathit{Exp} \times \mathcal{N} \to \mathcal{P}_\omega(\mathcal{N})$ returns the variables that would be captured if a variable u were substituted with those variables in an expression e (i.e. all the variables bound in e at the free occurrences of u in e)

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

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

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

3 Proof theory

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

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

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

$$\frac{\vdash cx : \textsc{context} \quad o \notin \mathcal{ON}(cx) \quad cx, \mathsf{tvar}\ \overline{\beta} \vdash T : \textsc{type}}{\vdash cx, \mathsf{op}\ o{:}\{\overline{\beta}\}\ T : \textsc{context}} \quad (\textsc{cxOdec})$$

$$\frac{\vdash cx : \textsc{context} \quad \beta \notin \mathcal{TV}(cx)}{\vdash cx, \mathsf{tvar}\ \beta : \textsc{context}} \quad (\textsc{cxTVdec})$$

$$\frac{\vdash cx : \textsc{context} \quad v \notin \mathcal{V}(cx) \quad cx \vdash T : \textsc{type}}{\vdash cx, \mathsf{var}\ v{:}T : \textsc{context}} \quad (\textsc{cxVdec})$$

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

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

$$\frac{\vdash cx : \textsc{context} \quad \beta \in \mathcal{TV}(cx)}{cx \vdash \beta : \textsc{type}} \quad (\textsc{tyVar})$$

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

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

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

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

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

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

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

The above rules slightly depart from [1] because subtyping judgements do not include subtyping predicates here. Subtyping predicates should not matter for type inference; also see explanation for rule exSub below.

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

$$\frac{\vdash cx : \textsc{context} \quad \mathsf{var}\ v{:}T \in cx}{cx \vdash v : T} \quad (\textsc{exVar})$$

$$\frac{\vdash cx : \textsc{context} \quad \mathsf{op}\ o{:}\{\overline{\beta}\}\ T \in cx \quad \forall i.\ \ cx \vdash T_i : \textsc{type}}{cx \vdash o[\overline{T}] : T[\overline{\beta}/\overline{T}]} \quad (\textsc{exOp})$$

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

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

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

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

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

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

We postulate an infinite subset

$$\mathcal{M} \subset \mathcal{N}$$

such that $\mathcal{N} - \mathcal{M}$ is also infinite. In other words, we partition the space of names $\mathcal{N}$ into two (infinite) subspaces $\mathcal{M}$ and $\mathcal{N} - \mathcal{M}$.

In the sequel, we regard type variables from $\mathcal{M}$ as *meta type variables* ("*metyvars*" for short), i.e. placeholders for types to be instantiated in order to solve the type inference problem. We assume that the type variables that appear in contexts are drawn from $\mathcal{N} - \mathcal{M}$, i.e. they are not metyvars. Type variables that are not metyvars are treated differently from metyvars: they are not instantiated to solve the type inference problem.

We exclusively use the symbol α (and decorated variants, e.g. α' and α_1) for metyvars in $\mathcal{M}$. Thus, the (non-)use of such symbols imply the condition that a type variable is (not) a metyvar.

Since types may contain expressions, the type inference problem consists of finding not only well-typedness proofs for expressions, but also well-formedness proofs for types:

- given a well-formed context $cx \in Cx$ and an expression $e \in Exp$, find an assignment $\sigma : \mathcal{M} \xrightarrow{\text{f}} Type$, a type T, and a proof of $cx \vdash e[\sigma] : T$;
- given a well-formed context $cx \in Cx$ and a type $T \in Type$, find an assignment $\sigma : \mathcal{M} \xrightarrow{\text{f}} Type$ and a proof of $cx \vdash T[\sigma] : \text{TYPE}$.

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 metyvars into all the spots where types are missing and then we solve the type inference problem as stated above in order to instantiate the metyvars into suitable types.

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 target expression e (or type T) and the context cx.

5.1 Constraints

We define an extension of the types defined in §2

$$\begin{aligned} Type_{\text{X}} = &\ \ldots \\ &+ \{max\ T \mid T \in Type_{\text{X}}\} \end{aligned}$$

and consequently an extension Exp_{X} of the expressions defined in §2. In other words, we add a new kind of type, of the form $max\ T$, which can occur anywhere a type can occur. The extension of the functions $\mathcal{TV}()$ and $\_[\_]$ is straightforward

$$\begin{aligned} \mathcal{TV}(max\ T) &= \mathcal{TV}(T) \\ (max\ T)[\sigma] &= max\ T[\sigma] \end{aligned}$$

The intuition behind a type of the form $max\ T$ is that it denotes the maximal supertype of T. This intuition is formalized by the function $R_{\text{M}} : Type_{\text{X}} + Exp_{\text{X}} \to Type_{\text{X}} + Exp_{\text{X}}$ defined by the following reduction rules[2]

$$\begin{aligned} max\ \mathsf{Bool} &\longrightarrow \mathsf{Bool} \\ max\ \beta &\longrightarrow \beta \\ max\ \tau[\overline{T}] &\longrightarrow \tau[\overline{T}] \\ max\ (T_1 \to T_2) &\longrightarrow T_1 \to max\ T_2 \\ max\ (T|r) &\longrightarrow max\ T \\ max\ max\ \alpha &\longrightarrow max\ \alpha \end{aligned}$$

These reduction rules are consistent with the subtyping rules in §3. They always eliminate max unless it is applied to a metyvar, because in that case we need to instantiate the metyvar first. Repeated applications of max to a metyvar are always reduced to one, because taking the maximal supertype twice is like taking it once. A type variable β "behaves" like a monomorphic type name τ (it does not get instantiated, because it is not a metyvar); so, $max\ \beta$ reduces to β.

[2]The function being defined by the reduction rules means that $R_{\text{M}}(x)$ is obtained by applying the reduction rules to all the types that occur in x exhaustively, i.e. until no more change takes place.

We define a predicate $\mathit{Maximal} \subseteq \mathit{Type}_{\mathrm{X}}$ as

$$\begin{array}{l}
\mathit{Maximal}(\mathit{max}\ T) \\
\mathit{Maximal}(\mathsf{Bool}) \\
\mathit{Maximal}(\beta) \\
\mathit{Maximal}(\tau[\overline{T}]) \\
\mathit{Maximal}(T_1 \to T_2) \ \Leftrightarrow\ \mathit{Maximal}(T_2) \\
\neg \mathit{Maximal}(T|r) \\
\neg \mathit{Maximal}(\alpha)
\end{array}$$

The predicate captures the fact that a type is *definitely* maximal, i.e. that we know for sure that it is maximal. In particular, the predicate is false for metyvars, because we do not know whether they will be instantiated to maximal types or not.

We define the set of all possible constraints as

$$\begin{array}{l}
\mathcal{C} = \{T_1 \approx T_2 \mid T_1, T_2 \in \mathit{Type}_{\mathrm{X}}\} \\
\quad + \{e_1 \approx e_2 \mid e_1, e_2 \in \mathit{Exp}_{\mathrm{X}}\}
\end{array}$$

In other words, a constraint consists of a pair of (extended) types or expressions, to be interpreted as an equation. It is straightforward to lift the functions $\mathcal{TV}$ and $\_[\_]$ to (sets of) constraints

$$\begin{array}{ll}
\mathcal{TV}(x_1 \approx x_2) & = \mathcal{TV}(x_1) \cup \mathcal{TV}(x_2) \\
(x_1 \approx x_2)[\sigma] & = x_1[\sigma] \approx x_2[\sigma] \\
\mathcal{TV}(C) & = \{\mathcal{TV}(x_1 \approx x_2) \mid x_1 \approx x_2 \in C\} \\
C[\sigma] & = \{(x_1 \approx x_2)[\sigma] \mid x_1 \approx x_2 \in C\}
\end{array}$$

5.2 Constraint generation

We generate constraints of the form $\mathit{max}\ T \approx \mathit{max}\ T'$. A constraint of this form asserts that the types T and T' are *compatible*, in the sense that expressions of either type can be cast to the other type by first applying rule exSuper zero or more times to reach the (common) maximal supertype and then applying rule exSub zero or more times to reach the type we want to cast the expression to.

We formalize the constraint generation process via two relations $\_\ \_\ \_ \rightsquigarrow \_ \subseteq \mathcal{P}_\omega(\mathcal{C}) \times \mathit{Cx} \times \mathit{Type} \times \mathcal{P}_\omega(\mathcal{C})$ and $\_\ \_\ \_ \rightsquigarrow \_\ \_ \subseteq \mathcal{P}_\omega(\mathcal{C}) \times \mathit{Cx} \times \mathit{Exp} \times \mathit{Type} \times \mathcal{P}_\omega(\mathcal{C})$. The first relation captures the type inference process that establishes whether types are well-formed; the second relation captures the type inference process that establishes whether expressions are well-typed.

The assertion $C\ \ cx\ \ T \rightsquigarrow C'$ captures a computation whose inputs are the set of constraints C generated so far, the context cx in which we are processing type T, and the type T whose well-formedness we are trying to establish, and whose output is a set of constraints $C' \supseteq C$ that must hold for T to be well-formed in cx.

The assertion $C\ \ cx\ \ e \rightsquigarrow T\ \ C'$ captures a computation whose inputs are the set of constraints C generated so far, the context cx in which we are processing expression e, and the expression e whose well-typedness we are trying to establish, and whose outputs are an inferred type T for e and a set of constraints $C' \supseteq C$ that must hold for e to have type T in cx.

The first relation is defined as

$$\frac{}{C\ \ cx\ \ \alpha \rightsquigarrow C} \quad (\textsc{conMeTyVar})$$

$$\frac{}{C\ \ cx\ \ \mathsf{Bool} \rightsquigarrow C} \quad (\textsc{conTyBool})$$

$$\frac{\beta \in \mathcal{TV}(cx)}{C\ \ cx\ \ \beta \rightsquigarrow C} \quad (\textsc{conTyVar})$$

$$\frac{\begin{array}{c}\mathsf{ty}\ \tau : |\overline{T}| \ \in\ cx \\ \forall i.\ \ C_{i-1} \quad cx \quad T_i \ \rightsquigarrow\ C_i\end{array}}{C_0 \quad cx \quad \tau[\overline{T}] \ \rightsquigarrow\ C_n} \quad (\textsc{conTyInst})$$

$$\frac{\begin{array}{c}C \quad cx \quad T \ \rightsquigarrow\ C' \\ C' \quad cx \quad T' \ \rightsquigarrow\ C''\end{array}}{C \quad cx \quad T \to T' \ \rightsquigarrow\ C''} \quad (\textsc{conTyArr})$$

$$\frac{\begin{array}{c}C \quad cx \quad T \ \rightsquigarrow\ C' \\ C' \quad cx \quad r \ \rightsquigarrow\ T' \quad C''\end{array}}{C \quad cx \quad T|r \ \rightsquigarrow\ C'' \cup \{max\ T' \approx max\ (T \to \mathsf{Bool})\}} \quad (\textsc{conTyRestr})$$

Explanation:

- A metyvar is always well-formed and generates no constraints.
- Type Bool is always well-formed and generates no constraints.
- A type variable is well-formed if it is declared in the context; no constraints are generated.
- A type instance is well-formed if its type name is declared in the context with an arity that matches the number of type arguments[3] and if all the type arguments are well-formed. The generated constraints are joined.
- An arrow type is well-formed if its domain and range types are. The generated constraints are joined.
- A subtype is well-formed if its base type is well-formed and in addition the expression has a type that is compatible with the type of predicates over the base type.

The second relation is defined as

$$\frac{\mathsf{var}\ v : T \in cx}{C \quad cx \quad v \ \rightsquigarrow\ T \quad C} \quad (\textsc{conExVar})$$

$$\frac{\begin{array}{c}\mathsf{op}\ o : \{\overline{\beta}\}\ T \ \in\ cx \\ |\overline{\beta}| = |\overline{T}| \\ \forall i.\ \ C_{i-1} \quad cx \quad T_i \ \rightsquigarrow\ C_i\end{array}}{C_0 \quad cx \quad o[\overline{T}] \ \rightsquigarrow\ T[\overline{\beta}/\overline{T}] \quad C_n} \quad (\textsc{conExOp})$$

$$\frac{\begin{array}{c}C \quad cx \quad e \ \rightsquigarrow\ T \quad C' \\ \alpha_1, \alpha_2 \notin \mathcal{TV}(C') \ \wedge\ \alpha_1 \neq \alpha_2 \\ C' \cup \{max\ T \approx max\ (\alpha_1 \to \alpha_2)\} \quad cx \quad e' \ \rightsquigarrow\ T' \quad C''\end{array}}{C \quad cx \quad e\ e' \ \rightsquigarrow\ \alpha_2 \quad C'' \cup \{max\ \alpha_1 \approx max\ T'\}} \quad (\textsc{conExApp})$$

$$\frac{\begin{array}{c}C \quad cx \quad T \ \rightsquigarrow\ C' \\ v \notin \mathcal{V}(cx) \\ C' \quad cx, \mathsf{var}\ v : T \quad e \ \rightsquigarrow\ T' \quad C''\end{array}}{C \quad cx \quad \lambda v : T.\ e \ \rightsquigarrow\ T \to T' \quad C''} \quad (\textsc{conExAbs})$$

[3]Since the definition of constrain generation does not require the context to be well-formed, there may be multiple declarations of the same type name in the context. This causes no problem in the definition of constraint generation. However, when we generate constraints for Metaslang we always use well-formed contexts.

$$\frac{\begin{array}{c} v \in \mathcal{V}(cx) \\ v' \notin \mathcal{V}(cx) \cup \mathcal{FV}(e) \cup \mathcal{CV}(e, v) \\ C \quad cx \quad \lambda v'{:}T.\, e[v/v'] \quad \rightsquigarrow \quad T' \quad C' \end{array}}{C \quad cx \quad \lambda v{:}T.\, e \quad \rightsquigarrow \quad T' \quad C'} \quad (\textsc{conExAbsAlpha})$$

Explanation:

- A variable is well-typed if it is declared in the context. No constraints are generated and the type of the variable is the one declared in the context.
- An op instance is well-typed if the op name is declared in the context and the number of type variables $\overline{\beta}$ in which the op is polymorphic matches the number of type arguments. In addition, the type arguments must be well-formed. The constraints are joined and the type of the op instance is obtained by substituting $\overline{\beta}$ with the type arguments in the declared type of the op.
- In order to check an application, we first check the function (i.e. the first expression of the application). We generate a constraint that the type of the function be compatible with some arrow type, whose domain and range are two fresh, distinct metyvars. Then we check the argument (i.e. the second expression of the application) and we generate a constraint that the domain and argument types are compatible.
- In order to check a lambda abstraction, we first check that the type of the bound variable is well-formed. After that, there are two cases. If the bound variable is not declared in the context, we move the variable into the context and check the body of the abstraction; the resulting constraints are returned, along with the obvious arrow type. If instead the bound variable is already declared in the context, we pick a fresh variable that does not occur in the context or in the body of the abstraction and that cannot be captured if subtituted for the old variable, and check the lambda abstraction obtained by renaming the variable (i.e. we perform an alpha conversion).

These relations can be readily implemented as mutually recursive, executable procedures, perhaps with the set of generated constraints stored into a global program variable. When no rule applies (e.g. a type name τ is not declared in the context) the procedures should terminate with an error.

5.3 Constraint solving

The function R_{M} is easily extended to (sets of) constraints

$$\begin{array}{ll} R_{\mathrm{M}}(x_1 \approx x_2) & = R_{\mathrm{M}}(x_1) \approx R_{\mathrm{M}}(x_2) \\ R_{\mathrm{M}}(C) & = \{R_{\mathrm{M}}(x_1 \approx x_2) \mid x_1 \approx x_2 \in C\} \end{array}$$

We define the function $R_{\mathrm{C}} : \mathcal{P}(\mathcal{C}) \to \mathcal{P}(\mathcal{C}) + \{\mathsf{fail}\}$ by the following reduction rules

$$\begin{array}{rcl} x \approx x & \longrightarrow & \emptyset \\ \alpha \approx T & \longrightarrow & \{\alpha \approx T\} \\ T \approx \alpha & \longrightarrow & \{T \approx \alpha\} \\ max\ \alpha \approx T & \longrightarrow & \{max\ \alpha \approx T\} \\ T \approx max\ \alpha & \longrightarrow & \{T \approx max\ \alpha\} \\ \tau[\overline{T}] \approx \tau[\overline{T}'] & \longrightarrow & \{T_i \approx T_i' \mid 1 \le i \le n\} \\ T_1 \to T_2 \approx T_1' \to T_2' & \longrightarrow & \{T_1 \approx T_1', T_2 \approx T_2'\} \\ T|r \approx T'|r' & \longrightarrow & \{T \approx T', r \approx r'\} \\ o[\overline{T}] \approx o[\overline{T}'] & \longrightarrow & \{T_i \approx T_i' \mid 1 \le i \le n\} \\ e_1\ e_2 \approx e_1'\ e_2' & \longrightarrow & \{e_1 \approx e_1', e_2 \approx e_2'\} \\ \lambda v{:}T.\, e \approx \lambda v{:}T'.\, e' & \longrightarrow & \{T \approx T', e \approx e'\} \\ x_1 \approx x_2 & \longrightarrow & \mathsf{fail} \end{array}$$

Each rule, except the last one, has the form $x_1 \approx x_2 \longrightarrow C'$, where C' is a set of constraints: its meaning is that, given a set of constraints C, if $x_1 \approx x_2 \in C$ we replace $x_1 \approx x_2$ with C', i.e. we reduce C to $(C - \{x_1 \approx x_2\}) \cup C'$. The order of the rules is significant, i.e. each rule applies only if all the previous rules do not. In particular, the last rule applies when none of the previous rules apply, and it reduces any set of constraints C that contains a constraint $x_1 \approx x_2$ that does not fit the patterns of the preceding rules to fail. Given a set of constraints C, $R_{\mathrm{C}}(C)$ is the final result of exhaustively applying all the reduction rules to the constraints in C, until no more change takes place (for example, if fail is reached).

The first rule eliminates trivially satisfied constraints. The next four rules serve to preserve constraints that have a metyvar (or the *max* of a metyvar) on some side, because they cannot be reduced. All the other rules except the last one decompose constraints into smaller ones. The last rule is used when there is some unsatisfiable constraint.

Given any set of constraints C, it is easy to see that $R_{\mathrm{C}}(R_{\mathrm{M}}(C))$ is either fail or a set of constraints C' whose constraints all have (the *max* of) a metyvar on some side. Note that constraints on expressions $e_1 \approx e_2$ arise only temporarily from the decomposition of constraints on restriction types, but eventually disappear during the exhaustive application of the reduction rules.

A set of constraints C_0 is solved via the following (pseudo) program, which makes use of two (program) variables: $C \in \mathcal{P}_\omega(\mathcal{C}) + \{\mathsf{fail}\}$, which holds the set of constraints left to solve (or fail); and $\sigma : \mathcal{M} \xrightarrow{\mathrm{f}} \mathit{Type}_{\mathrm{X}}$, which holds the solution of the constraints that is under construction.

1. Initialize $C := R_{\mathrm{C}}(R_{\mathrm{M}}(C_0))$ and $\sigma := \emptyset$.
2. If $C = \emptyset$ or $C = \mathsf{fail}$, terminate.
3. If C does not contain any constraint of the form $\alpha \approx T$ or $T \approx \alpha$, go to Step 7. Otherwise, choose a constraint of that form and let α and T be its constituents.
4. If $\alpha \in \mathcal{TV}(T)$ then set $C := \mathsf{fail}$ and terminate.
5. Substitute α with T in the current constraints and then reduce the resulting constraints, i.e. update $C := R_{\mathrm{C}}(R_{\mathrm{M}}(C[\alpha/T]))$.
6. Substitute α with T in all the types of the current solution σ and extend the current solution with the association of T to α, i.e. update $\sigma := \{\langle \alpha', T'[\alpha/T] \rangle \mid \langle \alpha', T' \rangle \in \sigma\} \cup \{\langle \alpha, T \rangle\}$. Go back to Step 2.
7. Choose a constraint $x_1 \approx x_2 \in C$ of the form $\mathit{max}\ \alpha \approx T$ or $T \approx \mathit{max}\ \alpha$. If $\neg \mathit{Maximal}(T)$ then set $C := \mathsf{fail}$ and terminate.
8. Replace $\mathit{max}\ \alpha$ with α in the chosen constraint, i.e. update $C := (C - \{x_1 \approx x_2\}) \cup \{\alpha \approx T\}$. Go back to Step 2.

The program consists of a loop that always terminates (at Step 2, Step 4, or Step 7), with $C = \emptyset$ or $C = \mathsf{fail}$. In the latter case, no solution has been found. In the former case, σ holds a solution.

There is some non-determinism at Step 3 and Step 7, in the choice of the α and T that form the constraint that is processed. However, the program as a whole should be deterministic [[[THIS NEEDS TO BE CHECKED AND PROVED]]].

A loop invariant is that $R_{\mathrm{C}}(R_{\mathrm{M}}(C)) = C$, i.e. the current constraints are always fully reduced. This means that all the constraints have (the *max* of) a metyvar on some side. Thus, if we go to Step 7 from Step 3, at Step 7 there must be at least one constraint with the *max* of a metyvar on some side (because $C \neq \emptyset$ is an explicit loop invariant).

The test at Step 4 checks that the constraint is not recursive in α. If it is, it cannot be solved. Recall that, in this document, we assume the absence of recursive types and the expansion of type equalities prior to type inference.

The update at Step 5 eliminates the metyvar α from C by substitution. The constraint $\alpha \approx T$ or $T \approx \alpha$ that has been chosen at Step 3 is also automatically eliminated by the reduction operated by R_{C}, because the subtitution turns the constraint into $T \approx T$.

The update at Step 6 updates the current solution in two ways. First, it eliminates α from the types associated to the previously solved metyvars, by substituting α with T in those types. Thus, at the end of program the solution will be fully "solved". Second, it extends the solution with α associated to T.

The test at Step 7 checks that a maximal type $max\ \alpha$ is not being required to be equal to a non-maximal type. Note that T cannot be a metyvar, because otherwise we would not have reached Step 7 from Step 3. Therefore, if $\mathit{Maximal}(T)$ does not hold, we know that T is definitely not maximal, and so the constraint cannot be satisfied.

The constraint replacement operated at Step 8 may "lose" solutions. The constraint $max\ \alpha \approx T$ has, as solutions, T and all the subtypes of T. By replacing $max\ \alpha$ with α, we limit α to be T and not any subtype of T. [[[WE NEED TO INVESTIGATE WHETHER THIS IS A DRAWBACK IN PRACTICE OR NOT AND/OR WHETHER THERE IS A WAY TO DEAL WITH $max\ \alpha$ DIFFERENTLY]]]

5.4 Putting it all together

The type inference problem stated in §4 is solved as follows.

Let cx be a well-formed context, and T the type whose well-formedness we are trying to establish:

- Let C_0 be the result of constraint generation (starting from the empty set of constraints), i.e. $\emptyset \quad cx \quad T \quad \leadsto \quad C_0$.
- Let C and σ be the results of the program in §5.3 (which takes C_0 as its initial input).
- If $C = \mathsf{fail}$, type inference fails because no solution can be found.
- Otherwise, if $\mathcal{D}(\sigma) \not\supseteq \mathcal{TV}(T) \cap \mathcal{M}$ or if $\alpha' \in \mathcal{TV}(\sigma(\alpha))$ for some $\alpha \in \mathcal{D}(\sigma)$, then type inference fails because the assignment of types to metyvars is not unique: we could assign any type to $\alpha \in (\mathcal{TV}(T) \cap \mathcal{M}) - \mathcal{D}(\sigma)$ or to $\alpha' \in \mathcal{TV}(\sigma(\alpha))$.
- Otherwise, $\mathcal{D}(\sigma) \supseteq \mathcal{TV}(T) \cap \mathcal{M}$ and there is no $\alpha' \in \mathcal{TV}(\sigma(\alpha))$ for any $\alpha \in \mathcal{D}(\sigma)$, i.e. all the metyvars have types assigned to them. The solution to the type inference problem is σ. Note that $\mathcal{D}(\sigma)$ may include metyvars not in T, introduced during the constraint generation process.

Let cx be a well-formed context, and e the type whose well-typedness we are trying to establish:

- Let C_0 and T be the results of constraint generation (starting from the empty set of constraints), i.e. $\emptyset \quad cx \quad e \quad \leadsto \quad T \quad C_0$.
- Let C and σ be the results of the program in §5.3 (which takes C_0 as its initial input).
- If $C = \mathsf{fail}$, type inference fails because no solution can be found.
- Otherwise, if $\mathcal{D}(\sigma) \not\supseteq \mathcal{TV}(e) \cap \mathcal{M}$ or if $\alpha' \in \mathcal{TV}(\sigma(\alpha))$ for some $\alpha \in \mathcal{D}(\sigma)$, then type inference fails because the assignment of types to metyvars is not unique: we could assign any type to $\alpha \in (\mathcal{TV}(e) \cap \mathcal{M}) - \mathcal{D}(\sigma)$ or to $\alpha' \in \mathcal{TV}(\sigma(\alpha))$.
- Otherwise, $\mathcal{D}(\sigma) \supseteq \mathcal{TV}(e) \cap \mathcal{M}$ and there is no $\alpha' \in \mathcal{TV}(\sigma(\alpha))$ for any $\alpha \in \mathcal{D}(\sigma)$, i.e. all the metyvars have types assigned to them. The solution to the type inference problem is σ. Note that $\mathcal{D}(\sigma)$ may include metyvars not in e, introduced during the constraint generation process. The inferred type for e is $T[\sigma]$.

6 Correctness

We prove that the algorithm in §5 indeed solves the type inference problem stated in §4. The proofs of the theorems are in §A.

6.1 Properties of maximal types

We start by proving some simple and intuitive properties that involve types with *max*. We use the notation $max^n\ T$, with $n > 0$, for $max\ \ldots\ max\ T$ (where *max* is applied n times). We use the notation $\ldots|\ldots$ to denote a restriction type whose constituents are irrelevant (in particular, the inequality $T \neq \ldots|\ldots$ means that T is not a restriction type).

Theorem 6.1 $R_{\mathrm{M}}(max\ \alpha) \neq R_{\mathrm{M}}(\alpha)$

Theorem 6.2 $R_{\mathrm{M}}(max^n\ T) = R_{\mathrm{M}}(max\ T)$

Theorem 6.3 $R_{\mathrm{M}}(max\ T) \neq \ldots|\ldots$

Theorem 6.4 $R_{\mathrm{M}}(max\ (T|r)) \neq R_{\mathrm{M}}(T|r)$

Theorem 6.5 $Maximal(T)\ \Leftrightarrow\ R_{\mathrm{M}}(max\ T) = R_{\mathrm{M}}(T)$

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.

A Proofs

A.1 Proof of Theorem 6.1

$R_{\mathrm{M}}(max\ \alpha)$
$\quad = \{\text{definition of } R_{\mathrm{M}}\}$
$max\ \alpha$
$\quad \neq \{\text{obvious}\}$
α
$\quad = \{\text{definition of } R_{\mathrm{M}}\}$
$R_{\mathrm{M}}(\alpha)$

A.2 Proof of Theorem 6.2

By induction on T:

- $R_{\mathrm{M}}(max^n\ \alpha)$
  $\quad = \{\text{reduction rule, } n-1 \text{ times}\}$
  $R_{\mathrm{M}}(max\ \alpha)$
- $R_{\mathrm{M}}(max^n\ \mathsf{Bool})$
  $\quad = \{\text{reduction rule, } n-1 \text{ times}\}$
  $R_{\mathrm{M}}(max\ \mathsf{Bool})$
- $R_{\mathrm{M}}(max^n\ \beta)$
  $\quad = \{\text{reduction rule, } n-1 \text{ times}\}$
  $R_{\mathrm{M}}(max\ \beta)$
- $R_{\mathrm{M}}(max^n\ \tau[\overline{T}])$
  $\quad = \{\text{reduction rule, } n-1 \text{ times}\}$
  $R_{\mathrm{M}}(max\ \tau[\overline{T}])$

- $R_{\mathrm{M}}(max^n\ (T_1 \to T_2))$
  $=$ {reduction rule, n times}
  $R_{\mathrm{M}}(T_1 \to max^n\ T_2)$
  $=$ {obvious property of R_{M}}
  $R_{\mathrm{M}}(T_1) \to R_{\mathrm{M}}(max^n\ T_2)$
  $=$ {induction hypothesis}
  $R_{\mathrm{M}}(T_1) \to R_{\mathrm{M}}(max\ T_2)$
  $=$ {obvious property of R_{M}}
  $R_{\mathrm{M}}(T_1 \to max\ T_2)$
  $=$ {reduction rule}
  $R_{\mathrm{M}}(max\ (T_1 \to T_2))$

- $R_{\mathrm{M}}(max^n\ (T|r))$
  $=$ {reduction rule}
  $R_{\mathrm{M}}(max^n\ T)$
  $=$ {induction hypothesis}
  $R_{\mathrm{M}}(max\ T)$
  $=$ {reduction rule}
  $R_{\mathrm{M}}(max\ (T|r))$

- $R_{\mathrm{M}}(max^n\ max\ T)$
  $=$ {obvious}
  $R_{\mathrm{M}}(max^{n+1}\ T)$
  $=$ {induction hypothesis}
  $R_{\mathrm{M}}(max\ T)$
  $=$ {induction hypothesis}
  $R_{\mathrm{M}}(max\ max\ T)$

A.3 Proof of Theorem 6.3

By induction on T:

- $R_{\mathrm{M}}(max\ \alpha)$
  $=$ {definition of R_{M}}
  $max\ \alpha$
  $\neq$ {obvious}
  $\ldots|\ldots$

- $R_{\mathrm{M}}(max\ \mathsf{Bool})$
  $=$ {definition of R_{M}}
  Bool
  $\neq$ {obvious}
  $\ldots|\ldots$

- $R_{\mathrm{M}}(max\ \tau[\overline{T}])$
  $=$ {reduction rule}
  $R_{\mathrm{M}}(\tau[\overline{T}])$
  $=$ {obvious property of R_{M}}
  $\tau[R_{\mathrm{M}}(\overline{T})]$
  $\neq$ {obvious}
  $\ldots|\ldots$

- $R_{\mathrm{M}}(max\ (T_1 \to T_2))$
  $=$ {reduction rule}
  $R_{\mathrm{M}}(T_1 \to max\ T_2)$
  $=$ {obvious property of R_{M}}

$R_{\mathrm{M}}(T_1) \to R_{\mathrm{M}}(max\ T_2)$
$\quad \neq$ {obvious}
$\ldots | \ldots$

- $R_{\mathrm{M}}(max\ (T|r))$
  $\quad =$ {reduction rule}
  $R_{\mathrm{M}}(max\ T)$
  $\quad \neq$ {induction hypothesis}
  $\ldots | \ldots$

- $R_{\mathrm{M}}(max\ max\ T)$
  $\quad =$ {Theorem 6.2}
  $R_{\mathrm{M}}(max\ T)$
  $\quad =$ {induction hypothesis}
  $\neq \ldots | \ldots$

A.4 Proof of Theorem 6.4

$R_{\mathrm{M}}(max\ (T|r))$
$\quad \neq$ {Theorem 6.3}
$R_{\mathrm{M}}(T)|R_{\mathrm{M}}(r)$
$\quad =$ {obvious property of R_{M}}
$R_{\mathrm{M}}(T|r)$

A.5 Proof of Theorem 6.5

By induction on T:

- $Maximal(\alpha)$
  $\quad \Leftrightarrow$ {definition of *Maximal*}
  false
  $\quad \Leftrightarrow$ {Theorem 6.1}
  $R_{\mathrm{M}}(max\ \alpha) = R_{\mathrm{M}}(\alpha)$

- $Maximal(\mathsf{Bool})$
  $\quad \Leftrightarrow$ {definition of *Maximal*}
  true
  $\quad \Leftrightarrow$ {reduction rule}
  $R_{\mathrm{M}}(max\ \mathsf{Bool}) = R_{\mathrm{M}}(\mathsf{Bool})$

- $Maximal(\tau[\overline{T}])$
  $\quad \Leftrightarrow$ {definition of *Maximal*}
  true
  $\quad \Leftrightarrow$ {reduction rule}
  $R_{\mathrm{M}}(max\ \tau[\overline{T}]) = R_{\mathrm{M}}(\tau[\overline{T}])$

- $Maximal(T_1 \to T_2)$
  $\quad \Leftrightarrow$ {definition of *Maximal*}
  $Maximal(T_2)$
  $\quad \Leftrightarrow$ {induction hypothesis}
  $R_{\mathrm{M}}(max\ T_2) = R_{\mathrm{M}}(T_2)$
  $\quad \Leftrightarrow$ {congruence}
  $R_{\mathrm{M}}(T_1) \to R_{\mathrm{M}}(max\ T_2) = R_{\mathrm{M}}(T_1) \to R_{\mathrm{M}}(T_2)$
  $\quad \Leftrightarrow$ {obvious property of R_{M}}
  $R_{\mathrm{M}}(T_1 \to max\ T_2) = R_{\mathrm{M}}(T_1 \to T_2)$
  $\quad \Leftrightarrow$ {reduction rule}
  $R_{\mathrm{M}}(max\ (T_1 \to T_2)) = R_{\mathrm{M}}(T_1 \to T_2)$

- $Maximal(T|r)$
  $\Leftrightarrow$ {definition of $Maximal$}
  false
  $\Leftrightarrow$ {Theorem 6.4}
  $R_{\mathrm{M}}(max\ (T|r)) = R_{\mathrm{M}}(T|r)$
- $Maximal(max\ T)$
  $\Leftrightarrow$ {definition of $Maximal$}
  true
  $\Leftrightarrow$ {Theorem 6.2}
  $R_{\mathrm{M}}(max\ max\ T) = R_{\mathrm{M}}(max\ T)$