

An Algorithm for Inferring Quasi-Static Types

Alberto Oliart*

December 14, 1994

Abstract

This report presents an algorithm, and its implementation, for doing type inference in the context of Quasi-Static Typing (QST) [14]. The package infers types a la “QST” for the simply typed λ -calculus.

Introduction

There has been an increasing interest in dealing with dynamic vs. static typing discipline issues in recent years. This interest comes from people working from diverse points of view. Some of them trying to incorporate some flexibility to the typing system of statically typed languages, as is the case with Partial Typing, in which heterogeneous data structures are included in the typing system. Others try to make the execution of programs written in a dynamically typed language more efficient by eliminating some of the dynamic type checks with the obtention of some typing information from the program.

In practice, in untyped programming languages like Lisp, only primitive values and predefined functions have types. For example, the function *car* has type $list \rightarrow S\text{-expression}$, that is, it takes as input a *list* and returns an *S-expression* as a result, but the function *first*, defined as (**defun** *first* (x) (**car** x)) has no type. The expression (*first* 3) will eventually produce a runtime type error, since **car** expects a list, and this error will be raised when *car* is applied to 3, and not before, as would be done in a language like ML.

We will call a typing system dynamic if the typechecking of operands is done at run time, that is, if the typechecking of operands is done exactly before the operation is executed. If the typing system does the checking at compile time it is a static typing system. Examples of dynamically typed programming languages are Lisp and Smalltalk.

Dynamic typing discipline offers a high degree of flexibility that static typing discipline cannot give, this means that there are some program phrases that can be executed in a dynamic type checking discipline that can not be written in a language with a static type checking discipline. An example of such a program is $(\lambda f.((fK)(fI)))I$ where $K = \lambda x.\lambda y.x$ and $I = \lambda x.x$. This term is

*Partly supported by CONACyT grant No. 54716. Address: 111 Cummington St. Boston, MA 02215 USA.

rejected by the ML typing system because the types of K and I are not unifiable. If we “execute” this term, we get $K I$ which reduces to $\lambda y. I$, which has type $\forall\alpha.\forall\beta.\alpha \rightarrow (\beta \rightarrow \beta)$. This program translated to Lisp runs without problems. This example shows another problem with statically typed languages. Sometimes there are programs that contain no typing error, but will be rejected by the typing system.

On the other hand, statically typed languages offer early error detection, and they give the opportunity for code optimization. Typically, a Lisp compiler has to “insert” code to do the necessary typechecking during run time, making the execution of programs more inefficient than it would be if the program had been written in a statically typed language, like ML.

One problem that sometime arises with statically typed languages is that programs that use some kind of external data cannot be typechecked at compile time, as are programs that read external files or distributed programs that make use of remote procedure calls. In these cases, some degree of dynamic typing is needed. As an example taken from [1] consider a program that reads a bitmap and displays it. If the bitmap is stored in an external file, the program has to read the contents of the file. If the contents is the representation of a bitmap, then there is no problem. If it is not, then there can be two ways of solving this, one is to treat the contents of the file as if it was a bitmap, and the other is to check that the contents of the file is a representation of a bitmap, and in case it is not raise an exception. So some dynamic typing is needed even in statically typed languages.

Another disadvantage of statically typed programming languages is the impossibility to have non homogeneous data objects. As an example, it is possible to handle the list $[1, true, \text{“string”}]$ in Lisp, but not in ML. This also motivates the addition of dynamics to static languages [13].

Our aim is to find a typing system that allows most programs to run but stops all programs it can guarantee will have type mismatches at run-time.

Note that given a static typing system A , a program that statically type checks in A must dynamically type check, this is, the set of programs statically typable by A , ST_A , is a subset of the set of programs that are dynamically type correct, DN , in symbols $ST_A \subseteq DN$. It is undecidable whether a program is dynamically type correct.

Re-stating our goal, we want a typing system A such that ST_A is as close as possible to DN , and such that lets all programs P , for which A cannot prove that P does not type check, run.

We present a type inference algorithm that infers types in the context of “Quasi Static Typing”, which was introduced by Thatte in [14]. The idea is to have a system that combines static and dynamic types, not accepting some expressions that will lead to run time type error, while accepting others that may lead to error at run time.

Quasi Static Typing (QST)

QST is a combination of Partial Types (see [13]) and automatic insertion of implicit positive (tagging) and negative (checking) coercions, see [14]. The system divides programs into three categories, well typed, ill typed and ambivalent.

The typing system has two phases, that can be integrated into one “pass”. The first one inserts implicit coercions where there is the possibility of a type mismatch, and the other does “plausibility checking”. Ill typed programs are those that the system can prove will lead to run time type error, and therefore “rejected” by the typing system. Programs that pass the second phase are either well typed or ambivalent and are allowed to run. Well typed programs will never produce a run time type error, whereas ambivalent programs may or may not end in a run time type error.

The types assigned by QST are Partial Types, which were originally introduced in [13] with the intention of type checking heterogeneous objects. They include the type Ω , which is a type assignable to all objects except the object called *wrong* that denotes run time type error, and a subtype relation. Partial types are defined by:

$$\tau ::= int \mid bool \mid \Omega \mid \tau \rightarrow \tau$$

The subtype relation, denoted \leq is defined as follows:

$$\begin{aligned} & \forall \tau, \tau \leq \Omega \\ & \tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2 \text{ iff } \tau'_1 \leq \tau_1 \text{ and } \tau_2 \leq \tau'_2 \end{aligned}$$

The object language is defined by:

$$M ::= c \mid x \mid M M \mid \lambda x : \tau. M \mid (M, M)$$

where x are variables and c integer or boolean constants.

The typing system “transforms” an expression given in the object language into an expression of an “internal” language. This internal language consists of the object language together with coercions. For each pair of types τ_1 and τ_2 such that $\tau_1 \leq \tau_2$ there are coercions $\uparrow_{\tau_1}^{\tau_2}$ and $\downarrow_{\tau_1}^{\tau_2}$. These coercions are inserted where there is the possibility of a type mismatch during run time, according to the typing rules below. In each typing rule the notation $TE \vdash e \Rightarrow e_{new} : \tau$ reads “under the assumptions TE the object expression e can be transformed into the internal expression e_{new} with type τ ”.

$$\begin{array}{c} \frac{}{TE \vdash x \Rightarrow x : TE(x)} \\ \\ \frac{TE[x \leftarrow \tau] \vdash M \Rightarrow M_1 : \sigma}{TE \vdash \lambda x : \tau. M \Rightarrow \lambda x : \tau. M_1 : \tau \rightarrow \sigma} \\ \\ \frac{TE \vdash M \Rightarrow M_1 : \sigma \rightarrow \mu \quad TE \vdash N \Rightarrow N_1 : \tau \quad \tau \geq \sigma}{TE \vdash MN \Rightarrow M_1(N_1 \downarrow_{\sigma}^{\tau}) : \mu} \end{array} \qquad \begin{array}{c} \frac{TE \vdash M \Rightarrow M_1 : \tau \quad \tau \leq \sigma}{TE \vdash M \Rightarrow (M_1 \uparrow_{\tau}^{\sigma}) : \sigma} \\ \\ \frac{TE \vdash M \Rightarrow M_1 : \sigma \quad TE \vdash N \Rightarrow N_1 : \tau}{TE \vdash (M, N) \Rightarrow (M_1, N_1) : \sigma \times \tau} \\ \\ \frac{TE \vdash M \Rightarrow M_1 : \Omega \quad TE \vdash N \Rightarrow N_1 : \tau}{TE \vdash MN \Rightarrow (M_1 \downarrow_{\tau \rightarrow \Omega}^{\Omega}) N_1 : \Omega} \end{array}$$

The second phase of the QST typing system tries to type expressions that could lead to run time type error. This phase, called plausibility checking, is described in [14] as a confluent terminating set of rewrite rules. These rules are given below. The object *wrong* means run time type error.

$$\begin{array}{c}
e \downarrow_{\tau}^{\tau} \rightsquigarrow e \\
\\
e \downarrow_{\sigma}^{\tau} \downarrow_{\mu}^{\sigma} \rightsquigarrow e \downarrow_{\mu}^{\tau} \\
\\
\frac{\mu = \tau \sqcap \nu}{e \uparrow_{\tau}^{\sigma} \downarrow_{\nu}^{\sigma} \rightsquigarrow e \downarrow_{\mu}^{\tau} \uparrow_{\mu}^{\nu}}
\end{array}
\qquad
\begin{array}{c}
e \uparrow_{\tau}^{\tau} \rightsquigarrow e \\
\\
e \uparrow_{\mu}^{\sigma} \uparrow_{\sigma}^{\tau} \rightsquigarrow e \uparrow_{\mu}^{\tau} \\
\\
\frac{\exists \mu. \mu = \tau \sqcap \nu}{e \uparrow_{\tau}^{\sigma} \downarrow_{\nu}^{\sigma} \rightsquigarrow \text{wrong}}
\end{array}$$

Example 1 $\lambda x : \Omega. x x$

As an example of how this system works consider the expression $\lambda x : \Omega. x x$. The typing of this expression is :

$$\frac{\frac{x \in \text{Dom}([x \leftarrow \Omega])}{[x \leftarrow \Omega] \vdash x \Rightarrow x : \Omega} \quad \frac{x \in \text{Dom}([x \leftarrow \Omega])}{[x \leftarrow \Omega] \vdash x \Rightarrow x : \Omega}}{[x \leftarrow \Omega] \vdash x x \Rightarrow (x \downarrow_{\Omega \rightarrow \Omega}^{\Omega}) x : \Omega}}{\emptyset \vdash \lambda x : \Omega. x x \Rightarrow \lambda x : \Omega. (x \downarrow_{\Omega \rightarrow \Omega}^{\Omega}) x : \Omega \rightarrow \Omega}$$

If we rewrite the expression in the following way, $\lambda x : \Omega \rightarrow \Omega. x x$, then the resulting internal expression is $\lambda x : \Omega \rightarrow \Omega. x x \uparrow_{\Omega \rightarrow \Omega}^{\Omega}$.

Example 2 $(\lambda x : \Omega \rightarrow \Omega. x x)(\lambda x : \Omega. x x)$

The resulting internal expression is: $(\lambda x : \Omega \rightarrow \Omega. (x x \uparrow_{\Omega \rightarrow \Omega}^{\Omega}))(\lambda x : \Omega. (x \downarrow_{\Omega \rightarrow \Omega}^{\Omega} x))$.

In [14], Thatte gives an algorithm called *Type* that tries to integrates in one pass the typing rules and the plausibility checking phase. Given an expression in the object language, this algorithm returns an expression in the internal language and a type for it. The algorithm, as it appears in [14] is :

$$\begin{array}{l}
\textit{Type}(TE, e) = \text{case } e \text{ of} \\
\quad x : TE(x), x \\
\quad \lambda x : \tau. e_{body} : \text{let } \tau_b, e_b = \textit{Type}(TE[x \leftarrow \tau], e_{body}) \\
\quad \quad \text{in } \tau \rightarrow \lambda x : \tau. e_b \\
\quad e_{fun} e_{arg} : \text{let } \tau_f, e_f = \textit{Type}(TE, e_{fun}) \text{ and } t_a, e_a = \textit{Type}(TE, e_{arg}) \\
\quad \quad \text{in if } \tau_f = \tau_{fa} \rightarrow \tau_{fr} \text{ then} \\
\quad \quad \quad \text{let } e_{na} = \textit{Simplify}(e_a \uparrow_{\tau_a}^{\Omega} \downarrow_{\tau_{fa}}^{\Omega}) \text{ in } \tau_{fr}, (e_f e_{na}) \\
\quad \quad \quad \text{else if } \tau_f = \Omega \text{ then } \Omega, (e_f \downarrow_{\tau_a \rightarrow \Omega}^{\Omega} e_a) \\
\quad \quad \quad \text{else } \textit{fail}
\end{array}$$

Contrary to the claim in [14], page 375, this algorithm does not type all expressions that can be typed by the typing rules, as shown by the following example. Consider the following expression:

$$(\lambda x : int.succ\ x)\ true$$

where *succ* is the successor function with type $int \rightarrow int$. One possible “typing” under the typing rules is:

$$((\lambda x : int.succ\ x) \uparrow_{int \rightarrow int}^{\Omega} \downarrow_{\Omega \rightarrow int}^{\Omega})\ true \uparrow_{bool}^{\Omega},$$

while the algorithm type rejects the expression as ill typed.

The *Type* algorithm is equivalent to the following typing rules :

$$TE \vdash x \Rightarrow x : TE(x) \qquad \frac{TE[x \leftarrow \tau_1] \vdash M \Rightarrow M_1 : \tau_2}{TE \vdash \lambda x : \tau_1.M \Rightarrow \lambda x : \tau_1.M_1 : \tau_1 \rightarrow \tau_2}$$

$$\frac{TE \vdash M \Rightarrow M_1 : \tau_1 \rightarrow \tau_2, TE \vdash N \Rightarrow N_1 : \tau_3, \tau_4 = g.l.b(\tau_1, \tau_2)}{TE \vdash MN \Rightarrow M_1(N_1 \downarrow_{\tau_4}^{\tau_3} \uparrow_{\tau_4}^{\tau_1}) : \tau_2}$$

$$\frac{TE \vdash M \Rightarrow M_1 : \Omega, TE \vdash N \Rightarrow N_1 : \tau}{TE \vdash MN \Rightarrow (M_1 \downarrow_{\tau \rightarrow \Omega}^{\Omega})N_1 : \Omega}$$

where $g.l.b(\tau_1, \tau_2)$ denotes the greatest lower bound of τ_1 and τ_2 .

If M is an expression, TE is a type environment and τ is a type, then we write

$$TE \vdash_{Type} M \Rightarrow M_1 : \tau$$

to say that τ is the type assigned to M , which is converted to M_1 by the above typing rules.

Type Inference

The object language is basically the same except that λ -bound variables need not be given a type. We have also eliminated pairing because it adds nothing to the discussion. The idea is to integrate plausibility checking into a typing system that does type inference, extending the work in [14]. The language is defined as follows:

$$M ::= x \mid c \mid \lambda x.M \mid \lambda x : \tau.M \mid M\ M$$

Types are also extended to include variables :

$$\tau ::= \alpha \mid int \mid bool \mid \tau \rightarrow \tau$$

The definition of the subtype relation is the same as in QST.

There is one problem. In [14] there are two typing systems, one is defined by the typing rules and the other by the *Type* algorithm. The choice is maybe a matter of taste. The set of expressions accepted by the *Type* algorithm are a proper subset of the expressions accepted by the typing rules plus plausibility checking (PC). Because the algorithm rejects more terms that are ill typed than the rules plus PC, we choose to extend the system defined by the *Type* algorithm.

To derive a type for an expression, the typing rules use an “environment” in the same way ML does, and also a set of constraints. The set of constraints is also modified by some of the typing rules. A particular expression is “typable” in the system if the derived set of constraints is solvable. A set of constraints can have more than one solution, and a particular solution determines the type given to the expression. We are looking for the solution that gives a type as precise and “general” as possible. A solution is a substitution σ that assigns ground types to type variables and such that, if C is the set of constraints, $C\sigma$ contains valid inequalities only. We call this system QSTIN. The typing rules are:

$$\frac{}{TE, C \vdash x : TE(x)} \qquad \frac{TE[x \leftarrow t_1], C \vdash M : t_2}{TE, C \vdash \lambda x.M : t_1 \rightarrow t_2}$$

$$\frac{TE, C_1 \vdash M : t_1 \rightarrow t_2, TE, C_2 \vdash N : t_3}{TE, C_1 \cup C_2 \cup \{t_4 \leq t_3, t_4 \leq t_1\} \vdash MN : t_2}$$

$$\frac{TE, C_1 \vdash M : t_1, TE, C_2 \vdash N : t_2, t_1 \neq t_3 \rightarrow t_4}{TE, C_1 \cup C_2 \cup \{t_3 \rightarrow t_4 \leq t_1, t_5 \leq t_3, t_5 \leq t_2\} \vdash MN : t_4}$$

We illustrate how these rules work with two examples:

Example 3 $((\lambda x.x)3)4$

Consider the following derivation for the expression $((\lambda x.x)3)4$:

$$\frac{\frac{\frac{\{x : \alpha_x\} \vdash x : \alpha_x}{\vdash \lambda x.x : \alpha_x \rightarrow \alpha_x} \quad \frac{}{\vdash 3 : int}}{\{t_1 \leq int, t_1 \leq \alpha_x\} \vdash (\lambda x.x)3 : \alpha_x} \quad \frac{}{\vdash 4 : int}}{\{t_1 \leq int, t_1 \leq \alpha_x, t_2 \rightarrow t_3 \leq \alpha_x, t_4 \leq t_2, t_4 \leq int\} \vdash ((\lambda x.x)3)4 : t_3}$$

The set of constraints for the given expression is :

$$C = \{t_1 \leq \alpha_x, t_1 \leq int, t_2 \rightarrow t_3 \leq \alpha_x, t_4 \leq t_2, t_4 \leq int\}$$

which has, among others, the following solution σ :

$$\sigma(\alpha_x) = \Omega, \sigma(t_1) = int, \sigma(t_2) = int, \sigma(t_3) = \Omega, \sigma(t_4) = int$$

Using this typing together with Thatte's *Type* algorithm we get the following expression:

$$(((\lambda x : \Omega.x)3) \downarrow_{int \rightarrow \Omega}^{\Omega})4 : \Omega$$

Example 4 $(\lambda x.x x)(\lambda y.y y)$

The set of constraints for the expression $(\lambda x.x x)(\lambda y.y y)$ is:

$$t_1 \rightarrow t_2 \leq \alpha_x, t_3 \leq \alpha_x, t_3 \leq t_1, t'_1 \rightarrow t'_2 \leq \alpha_y, t_3 \leq \alpha_y, t_3 \leq t'_1, t_4 \leq \alpha_x, t_4 \leq \alpha_y \rightarrow t'_2$$

The above constraints have as a solution the following :

$$\begin{aligned} t_1 = \Omega \rightarrow t_2, t'_1 = \Omega \rightarrow t'_2, \alpha_x = (\Omega \rightarrow t_2) \rightarrow t_2, \alpha_y = (\Omega \rightarrow t'_2) \rightarrow t'_2 \\ t'_2 = t_2, t_3 = \Omega \rightarrow t_2, t'_3 = \Omega \rightarrow t'_2, t_4 = (\Omega \rightarrow t_2) \rightarrow t_2 \end{aligned}$$

Definition 1 If M is a term in QST, then $strip(M)$ is defined as follows :

$$\begin{aligned} strip(x) &= x \\ strip(\lambda x : \tau.M) &= \lambda x.strip(M) \\ strip(MN) &= strip(M)strip(N) \end{aligned}$$

Proposition 1 Let M be a QST expression. If $TE \vdash_{Type} M \Rightarrow M_1 : \tau$ then $TE, C \vdash_{QSTIN} M : \tau$, and C has a solution σ .

Proof:

By induction on the size of M .

If $M = x$ then we have that $TE, \emptyset \vdash_{QSTIN} x$, and the identity is a solution for \emptyset .

If $M = \lambda x : \tau_1.M_1 : \tau_1 \rightarrow \tau_2$ then we have that, by the typig rules for *Type*, $TE[x \leftarrow \tau_1] \vdash_{Type} M \Rightarrow M_1 : \tau_2$. By I.H. we then have that $TE[x \leftarrow \tau_1], C \vdash_{QSTIN} M : \tau_2$, and C has a solution σ . Using the rule for introduction of \rightarrow we then have that $TE, C \vdash_{QSTIN} \lambda x.M : \tau_1 \rightarrow \tau_2$, and σ is a solution for C , which is what we want.

If $M = M_1 M_2$ then we have to check to possibilities that correspond to the two typing rules for application.

(1) We have the following $TE \vdash_{Type} M_1 \Rightarrow M'_1 : \tau_1 \rightarrow \tau_2$ and $TE \vdash_{Type} M_2 \Rightarrow M'_2 : \tau_3$ and $\tau_4 = g.l.b(\tau_1, \tau_3)$.

By I.H. we then have that $TE, C_1 \vdash_{QSTIN} M_1 : \tau_1 \rightarrow \tau_2$, $TE, C_2 \vdash_{QSTIN} M_2 : \tau_3$, and also that C_1 and C_2 have solutions σ_1 and σ_2 respectively. Applying the corresponding QSTIN application rule we then get

$$TE, C_1 \cup C_2 \cup \{\tau'_4 \leq \tau_1, \tau'_4 \leq \tau_3\} \vdash_{QSTIN} M_1 M_2 : \tau_2$$

Observe that $Dom(\sigma_1) \cap Dom(\sigma_2) = \emptyset$. This is because the only type variables that occur in C_1 and C_2 are the ones introduced by the typing rules, and these are all fresh variables.

Let σ be defined as follows :

$$\sigma(\tau'_4) = \tau_4$$

$$\sigma(\alpha) = \sigma_1(\alpha) \text{ iff } \alpha \in Dom(\sigma_1)$$

$$\sigma(\alpha) = \sigma_2(\alpha) \text{ iff } \alpha \in Dom(\sigma_2)$$

It is easy to see that σ is a solution for $C_1 \cup C_2 \cup \{\tau'_4 \leq \tau_1, \tau'_4 \leq \tau_3\}$.

(2) $TE \vdash_{Type} M_1 \Rightarrow M'_1 : \Omega$ $TE \vdash_{Type} M_2 \Rightarrow M'_2 : \tau$. By the I.H. we have that $TE, C_1 \vdash_{QSTIN} M_1 : \Omega$, $TE, C_2 \vdash_{QSTIN} M_2 : \tau$, and σ_1 and σ_2 are solutions for C_1 and C_2 respectively. We also have that $Dom(\sigma_1) \cap Dom(\sigma_2) = \emptyset$. Using the corresponding application rule we then have

$$TE, C_1 \cup C_2 \cup \{\tau_1 \rightarrow \tau_2 \leq \Omega, \tau_3 \leq \tau_1, \tau_3 \leq \tau\} \vdash_{QSTIN} M_1 M_2 : \tau_2.$$

Let σ be defined as follows:

$$\sigma(\tau_2) = \Omega$$

$$\sigma(\tau_3) = \sigma(\tau_1) = \tau$$

$$\sigma(\alpha) = \sigma_1(\alpha) \text{ iff } \alpha \in Dom(\sigma_1)$$

$$\sigma(\alpha) = \sigma_2(\alpha) \text{ iff } \alpha \in Dom(\sigma_2)$$

It is clear that σ is a solution to the set of constraints. □

Definition 2 If σ is a substitution we define σ_{ext} as follows :

$$\begin{aligned} \sigma_{ext}(\alpha) &= \sigma(\alpha) \text{ iff } \sigma(\alpha) \notin V, \text{ the set of type variables} \\ \sigma_{ext}(\alpha) &= \Omega \text{ otherwise} \end{aligned}$$

Definition 3 If σ is a substitution and T is a type environment, then we define T_σ as follows :

$$\begin{aligned} Dom(T_\sigma) &= Dom(T) \\ T_\sigma(\alpha) &= \sigma_{ext}(T(\alpha)) \end{aligned}$$

Proposition 2 *If M is a QSTIN expression and $T, C \vdash_{QSTIN} M : \tau$ and C has solution σ then there is a QST expression N , an internal QST expression N_1 , and a QST type τ_1 such that $strip(M) = strip(N)$, $T_\sigma \vdash_{Type} N \Rightarrow N_1 : \tau_1$ and $\tau_1 = \sigma_{ext}(\tau)$.*

Proof:

The proof is by induction on the size of M .

If $M = x$ then making $N = x = N_1$ we have that $T_\sigma \vdash_{Type} N \Rightarrow N_1 : T_\sigma(x)$, which is what we want.

If $M = \lambda x.M_1$ then we have that $T[x \leftarrow \tau_1], C \vdash_{QSTIN} M_1$. By I.H. we have then that there is a N_1 such that $strip(N) = strip(N_1)$ and a type τ'_1 such that $T_\sigma \vdash_{Type} N_1 \Rightarrow N'_1 : \tau'_1$ where $\tau'_1 = \sigma_{ext}(\tau_1)$. Applying the \rightarrow -intro rule for Type we get the desired result.

If $M = M_1 M_2$ then we have to check two cases :

(1) $T, C_1 \vdash_{QSTIN} M_1 : \tau_1 \rightarrow \tau_2$, $T, C_2 \vdash_{QSTIN} M_2 : \tau_3$ and $C = C_1 \cup C_2 \cup \{\tau_4 \leq \tau_1, \tau_4 \leq \tau_3\}$ with solution σ , which is also a solution for C_1 and C_2 .

By I.H. we have that $T_\sigma \vdash_{Type} N_1 \Rightarrow N'_1 : \tau'_1 \rightarrow \tau'_2$, and $T_\sigma \vdash_{QSTIN} N_2 \Rightarrow N'_2 : \tau'_3$ for some $N_1, N'_1, N_2, N'_2, \tau'_1, \tau'_2, \tau'_3$, with $\tau'_1 = \sigma_{ext}(\tau_1)$, and $\tau'_3 = \sigma_{ext}(\tau_3)$.

We also have that $\sigma(\tau_4) \leq \tau'_1$, and $\sigma(\tau_4) \leq \tau'_3$, and therefore there exists a $\tau_5 = g.l.b(\tau'_1, \tau'_3)$, and therefore we can apply the corresponding QSTIN rule and get $T_\sigma \vdash_{Type} N_1 N_2 \Rightarrow N'_1 (N'_2 \downarrow_{\tau_5}^{\tau'_3} \uparrow_{\tau_5}^{\tau'_1}) : \sigma_{ext}(\tau_2)$.

(2) $T, C_1 \vdash_{QSTIN} M_1 : \tau_1$, $T, C_2 \vdash_{QSTIN} M_2 : \tau_2$ with $C = C_1 \cup C_2 \cup \{\tau_3 \rightarrow \tau_4 \leq \tau_1, \tau_5 \leq \tau_3, \tau_5 \leq \tau_2\}$.

By I.H. we have that $T_\sigma \vdash_{Type} N_1 \Rightarrow N'_1 : \tau'_1$ and $T_\sigma \vdash_{Type} N_2 \Rightarrow N'_2 : \tau'_2$.

If we have that $\sigma(\tau_1) = \tau_{1l} \rightarrow \tau_{1r}$ for some τ_{1l} and τ_{1r} then this is similar to case (1).

If $\sigma(\tau_1) = \Omega$ then we have that we can apply the corresponding application rule of Type and we get $T_\sigma \vdash_{Type} N_1 N_2 \Rightarrow (N'_1 \downarrow_{\tau_2}^{\Omega} \rightarrow \Omega)$, and this concludes our proof.

□

As mentioned above an expression is “typable” using the rules above if the set of constraints C obtained has a solution. The algorithm we use to solve the set of constraints is described next:

Let C be a set of constraints. The idea is to transform C into a set of constraints C_1 such that σ is a solution of C_1 iff it is a solution for C . With $C\sigma$ we denote the application of σ to C . The transformation and the building of σ are as follows:

1. For all constraints of the form $t \leq int$ where t is a type variable make $\sigma(t) = int$, and make $C = (C - \{t \leq int\})\sigma$. If an inconsistent inequality is created stop signaling a type error.
2. For all constraints of the form $t \leq bool$ where t is a type variable make $\sigma(t) = bool$, and make

- $C = (C - \{t \leq \text{bool}\})\sigma$. If an inconsistent inequality is created stop signaling a type error.
3. For all constraints of the form $\Omega \leq t$ where t is a type variable make $\sigma(t) = \Omega$ and make $C = (C - \{\Omega \leq t\})\sigma$. If an inconsistent inequality is created stop signaling a type error.
 4. For all constraints of the form $t \leq \tau_1 \rightarrow \tau_2$ make $\sigma(t) = t_l \rightarrow t_r$, where t_l and t_r are fresh variables, and make $C = ((C - \{t \leq \tau_1 \rightarrow \tau_2\}) \cup \{\tau_1 \leq t_l, t_r \leq \tau_2\})\sigma$. If no inconsistencies have been introduced goto 1, otherwise stop signaling a type error.
 5. For all constraints of the form $t_1 \rightarrow t_2 \leq \tau_1 \rightarrow \tau_2$ make $C = (C - \{t_1 \rightarrow t_2 \leq \tau_1 \rightarrow \tau_2\}) \cup \{\tau_1 \leq t_1, t_2 \leq \tau_2\}$. If no inconsistencies were introduced then goto 1, otherwise stop signaling a type mismatch.
 6. For all inequalities of the form $t \leq \Omega$ make $C = C - \{t \leq \Omega\}$

At the end of this process C has been converted into a set of constraints of the form $\tau_1 \leq \tau_2$, where τ_1 and τ_2 are type variables, and of the form $\tau_1 \rightarrow \tau_2 \leq \tau_3$, where τ_3 is a type variable. We also have a substitution σ , which will be extended to a solution of C as follows:

If t_1 is a type variable and there is an inequality $t_2 \leq t_1$ and t_2 is not an arrow type, then make $\sigma(t_1) = t_2$. If t_2 is an arrow type then, if t_1 occurs in t_2 make $\sigma(t_1) = \Omega$, otherwise make $\sigma(t_1) = t_2$.

The above rules do not have a most general type property that some typing systems, like ML, enjoy. This means that given an expression, there is no type from which all other possible types for that expression can be derived in some fashion. This is because of the use of type Ω . This creates problems for *let*-polymorphism. As an example consider the following expression (in the expression we make use of lists, not included in the original language, we do this because this example has been used elsewhere to illustrate the problem, see [6] and [14]):

$$M = \lambda x. \text{cons}(1, x)$$

The type of M is $\text{int-list} \rightarrow \text{int-list}$. But if we apply M to a boolean, say $M \text{ true}$, then M could be modified as follows by the above system:

$$\lambda x : \Omega\text{-list}. \text{cons}(1 \uparrow_{\text{int}}^{\Omega}, x)$$

with type $\Omega\text{-list} \rightarrow \Omega\text{-list}$. One could conclude that the type $\forall \alpha. \alpha\text{-list} \rightarrow \alpha\text{-list}$ is a valid type for M , but the type $\text{bool-list} \rightarrow \text{bool-list}$ is not a valid type for M .

References

- [1] Martin Abadi, Luca Cardelli, Benjamin Pierce, Gordon Plotkin. *Dynamic Typing in a Statically Typed Language* ACM Transactions on Programming Languages and Systems Vol. 13, No. 2, April 1991
- [2] Luca Cardelli and Peter Wegner *On Understanding Types, Data Abstractions, and Polymorphism* Computing Surveys, Vol 17, No. 4, Dec. 1985
- [3] R. Cartwright and M. Fagan *Soft Typing* Proc. ACM SIGPLAN '91 ACM Press 1991.
- [4] L. Damas, R. Milner *Principal Type Schemes for Functional Programs* 9th ACM Symp. on Principles of Programming Languages 1982
- [5] C. Gomard *Partial Type Inference for Untyped Functional Languages (extended abstract)* Proc.. Lisp and Functional Programming 1990.
- [6] Fritz Henglein *Dynamic Typing* To appear in European Symposium on Programming 1992.
- [7] Fritz Henglein *Global Tagging optimizations by Type Checking* To appear in Lisp and Functional Programming 1992.
- [8] Dexter Kozen, Jens Palsberg, Michael I. Schwartzbach *Efficient Inference of Partial Types* to appear
- [9] Hans Leiss *Combining Recursive and Dynamic Types* Some Proceedings, 1993.
- [10] Xavier Leroy, Michel Mauny *Dynamics in ML* 5Th ACM Conference on Functional Programming Languages and Computer Architecture. Lecture Notes In Computer Science. Springer Verlag 1991.
- [11] Patrick M. O'Keefe, Mitchell Wand *Type Inference for Partial Types is Decidable* Proceedings of Fifth European Symposium on Programming, LNCS 582 1992
- [12] Plotkin, G. *Call by Name, Call by Value, and the λ -calculus* Theoretical Computer Science, 1 1975
- [13] Satish Thatte *Type inference with partial types* Proc. Int. Coll. on Automata, Languages and Programming, 1988
- [14] Satish Thatte *Quasy-static Typing* Proc. ACM Symp. on Principles of Programming Languages, 1988
- [15] David. A. Watt *Programming Language Concepts and Paradigms* Prentice Hall International, 1990

A Appendix

The algorithm presented in this report has been implemented. In this appendix we show some examples of the types obtained using the program. The output of the program has been slightly modified for readability.

The syntax used by the “type checker” is similar to the used by Standard ML of New Jersey. A λ -expression of the form $\lambda x.M$ translates to $fnx \Rightarrow M$, and an expression of the form $M N$ are translated to $M @ N$.

The output of the program consists of the following: A set of constraints, a set of equalities that show the solutions to the constraints, and the expression with the given type. The character W represents the type Ω . The type variables are of the form $TVNN$, where NN is an integer.

Example 5 $\lambda x.x$

```
qst> fn x => x;
{ TV1 = TV1 , }
( FN x : TV1 => x ) : ( TV1 -> TV1 )
```

Example 6 $\lambda x.x x$

```
qst> fn x => x @ x;
( TV2 -> TV3 ) <= TV1, TV4 <= TV2, TV4 <= TV1
{ TV1 = W, TV2 = TV4, TV3 = TV3, TV4 = TV4 }
( FN x : W => (x @ x) ) : ( W -> TV3 )
```

Example 7 $(\lambda x.x x)(\lambda y.y y)$

```
qst> (fn x => x @ x) @ (fn y => y @ y);
( TV2 -> TV3 ) <= TV1, TV4 <= TV2, TV4 <= TV1, ( TV6 -> TV7 ) <= TV5,
TV8 <= TV6, TV8 <= TV5
{ TV1 = W, TV2 = TV4, TV3 = TV3, TV4 = TV4, TV5 = W, TV6 = TV8, TV7 = TV7,
TV8 = TV8 }
(( FN x : W => (x @ x) ) @ ( FN y : W => (y @ y) )) : TV3
```

Example 8 $\lambda f.((f(\lambda x.\lambda y.x))(f(\lambda z.z)))$

```
qst> (fn f => ((f @ (fn x => fn y => x)) @ (f @ (fn z => z))));

( TV4 -> TV5 ) <= TV1, TV2 <= TV6, TV3 <= TV7, TV8 <= TV2,
( TV6 -> ( TV7 -> TV8 ) ) <= TV4, ( TV10 -> TV11 ) <= TV1,
( TV12 -> TV13 ) <= TV5, TV14 <= TV12

{ TV1 = W, TV2 = W, TV3 = TV7, TV4 = ( W -> ( TV7 -> TV8 ) ),
TV5 = ( TV14 -> TV13 ), TV6 = W, TV7 = TV7, TV8 = TV8, TV9 = TV9, TV10 = TV10,
TV11 = W, TV12 = TV14, TV13 = TV13, TV14 = TV14 }

( FN f : W => ((f @ ( FN x : W => ( FN y : TV7 => x ) ))@
(f @ ( FN z : TV9 => z ))) ): ( W -> TV13 )
```

Example 9 $\lambda f.(\lambda x.\lambda z.(f(x x))z)(\lambda x.\lambda z.(f(x x))z)$

```
qst> (fn f => (fn x => fn z => (f @ (x @ x)) @ z) @ (fn x => fn z => (f @ (x @
x)) @ z));

( TV4 -> TV5 ) <= TV2, TV6 <= TV4, TV6 <= TV2, ( TV7 -> TV8 ) <= TV1,
TV9 <= TV7, TV9 <= TV5, ( TV10 -> TV11 ) <= TV8, TV12 <= TV10, TV12 <= TV3,
( TV15 -> TV16 ) <= TV13, TV17 <= TV15, TV17 <= TV13, ( TV18 -> TV19 ) <= TV1

{ TV1 = W, TV2 = W, TV3 = TV12, TV4 = TV6, TV5 = TV9, TV6 = TV6, TV7 = TV9,
TV8 = ( TV12 -> TV11 ), TV9 = TV9, TV10 = TV12, TV11 = TV11, TV12 = TV12,
TV13 = W, TV14 = TV14, TV15 = TV17, TV16 = TV16, TV17 = TV17, TV18 = TV18,
TV19 = W}

( FN f : W => (( FN x : W => ( FN z : TV12 => ((f @ (x @ x))@ z ) ) )@
( FN x : W => ( FN z : TV14 => ((f @ (x @ x))@ z ) ) ) ): ( W -> ( TV12 ->
TV11 ) )
```

Example 10 $(\lambda x.x \text{ true})4$

```
qst> (fn x => x @ true) @ 4;

( TV2 -> TV3 ) <= TV1, bool <= TV2, int <= TV1 ,

{ TV1 = W, TV2 = bool, TV3 = TV3 }

(( FN x : W => (x @ true ) )@ 4 ): TV3
```

Example 11 $((\lambda x.\lambda y.x y)4)3$

```
qst> ((fn x => fn y => x @ y) @ 4) @ 3;
```

```
( TV3 -> TV4 ) <= TV1, TV5 <= TV3, TV5 <= TV2, int <= TV1, int <= TV2
```

```
{ TV1 = W, TV2 = W, TV3 = W, TV4 = TV4, TV5 = W }
```

```
(( ( FN x : W => ( FN y : W => ( x @ y ) ) ) @ 4 ) @ 3 ): TV4
```