Type checking meta programs

Nikolaj Bjørner[*]
Kestrel Institute
bjorner@kestrel.edu

September 1, 1999

Abstract

We report on preliminary experiments with inferring types for meta programs: programs that manipulate programs. For this purpose we provide a two-level type system in a fragment of a higher-order system of dependent types. The system is formulated with automatic type inference in mind.

In particular, we give a type system for dependent types and a constraint generation procedure which generates semi-unification constraints from untyped terms that have a solution if and only if the terms have a type annotation in the type system. More interestingly, typability is preserved under reflection, i.e. when object level programs are reflected to the meta-level.

1 Introduction

We would like to have a way to infer that the operations meta programs perform on their objects preserve typability of the objects. Here, we develop type rules and constraint solving techniques for inferring types of such programs.

On the surface this may seem as an innocent exercise in extending for instance the Hindley-Milner typing discipline. However, since meta programs are often uniform in the sort of the object level expressions that they manipulate, we observe that a type system based on dependent types becomes relevant.

We provide a two-level type system that allows to annotate data according to a simply typed discipline. Programs that are allowed to inspect and change the data using pattern matching are on the other hand typed using a dependent type discipline containing the types from the data. A central concern here is to infer types at both levels automatically, which has led us to review notions such as polymorphic recursion and rank-bounded polymorphism where types can be inferred by solving a suitable set of constraints (which unfortunately is not always decidable in light of polymorphic recursion).

An example: Our leading example is that of a map operator that applies a function to every subterm of an object level program. In ML or Haskell we would define a datatype called `Term` to represent the object level programs together with the definition of `map`

```
datatype Term =
     App of Term * Term             map f (App(M,N)) = f(App(map f M,map f N))
   | Lam of var  * Term             map f (Lam(M,N)) = f(Lam(M,map f N))
   | Var of var                     map f (Var M)    = f(Var(M))
```

The Hindley-Milner discipline infers the type `map : (Term -> Term) -> Term -> Term` for `map`, but does not capture that for the subset of terms in `Term` that can be typed, `map` preserves the derivable type if `f` does so too.

[*]This material is based upon work supported by NASA under award No NAG 2-1227, Incorporation of Inference Servers using Logic Morphisms, and upon work supported by DARPA under Contract No. F30602-96-C-0282. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of DARPA or NASA.

Dependent types: Richer typing disciplines, such as LF, or $\lambda P2$ [AC98] do on the other hand allow to easily capture such dependent type information using judgments of the form:

$$\mathsf{op}\ \mathtt{map} : \Pi w : \mathsf{sort}\ .\ ((\Pi v : \mathsf{sort}.\mathsf{Term}[v] \to \mathsf{Term}[v])\ \to\ \mathsf{Term}[w] \to \mathsf{Term}[w]) \tag{1}$$

such that `map` has a dependent type. The abstraction operator Π is the usual constructor for dependent function spaces. The type constructor Term takes here an argument of type sort and generates a type. In LF it can be annotated as $\Pi v : \mathsf{sort}\ .\ tp$.

In fact, Twelf [PS98] employs sophisticated type inference based on higher-order pattern unification to infer missing information from declarations with implicit abstractions. Unfortunately the type of our `map` example requires inserting the dependent abstraction Π in a contravariant position. It is also an example of a recursive function that is polymorphic in the object sort. It therefore does not seem fall into the class of functions where the approach in Twelf infers the desired dependent type. On the other hand, Twelf benefits from higher-order pattern unification to infer types where standard first-order unification does not apply. An interesting extension of OCaml with dependent types is reported in [XP99]. A noteworthy application of these is extended static checking of array bounds checking.

The programming language Cayenne [Aug99] uses a general higher-order stratified dependent type system which allows to encode type preserving meta-programs, such as interpreters [1], directly. The compiler requires types to be fully annotated with types, yet type checking is undecidable as Cayenne allows fixpoint operators.

With our narrower scope of only wanting to infer types for meta programs over an object language that reflects the meta language closely we here attempt to enable type inference to infer the appropriate types for recursively defined functions such as `map`. We develop a type derivation calculus and use it to extract second-order semi-unification constraints and report on experiments with a prototype implementation that solves these constraints and returns principal types.

Other applications: We intend our type inference, when in a mature stage, to be applied to programs that manipulate typed object programs. In particular we are building several interfaces from the categorical specification and program refinement system Specware [SJ95] to theorem provers such as PVS, HOL98, SNARK, Gandalf, Setheo, and Spass (for proving more interesting properties of specifications than available in the type system given here) as well as to programming languages Lisp, C, Java, for extracting executable code from computable specifications. These interfaces are written in Specware's specification language itself. Notice that it is only necessary to encode the part of the target language, whether it from a theorem prover or programming language, which is in the image of our translations from Specware. We thereby avoid having to encode the full glory of rich type systems, such as that of PVS, in order to infer types for the translations. It accesses the language constructs using a recursive datatype much similar to the datatype `Term` (but somewhat richer) and uses successive transformations to normalize terms in order to interface these with theorem provers or programming languages. To interface with C, for instance, we employ transformations of the form and type:

Giving bound variables unique names:
$\mathit{uniqueNames} : \Pi v : \mathsf{sort}\ .\ \mathsf{NameSupply} \times \mathsf{Term}[v]\ \to\ \mathsf{NameSupply} \times \mathsf{Term}[v]$

Standardizing arities for targeting multi-argument function application:
$\mathit{normalizeArity} : \Pi v : \mathsf{sort}\ .\ \mathsf{ArityMap} \times \mathsf{Term}[v]\ \to\ \mathsf{Term}[v]$

Code optimization by simplification:
$\mathit{simplify} : \Pi v : \mathsf{sort}\ .\ \mathsf{Term}[v]\ \to\ \mathsf{Term}[v]$

Lambda lifting:
$\mathit{lambdaLiftTerm} : \Pi v : \mathsf{sort}\ .\ \mathsf{Term}[v]\ \to\ \mathsf{Term}[v]$

Insertion of explicit tail-recursive calls to compile into iteration:
$\mathit{tailRecursionEliminate} : \Pi v : \mathsf{sort}\ .\ \mathsf{Term}[v]\ \to\ \mathsf{Term}[v]$

[1] See `http://www.cs.chalmers.se/~augustss/cayenne`.

2 A two-level language and its typing discipline

We capture the essence of Specware's higher-order specification language using typed λ-calculus. It includes designated constructors App, Lam, Var that represent the object level representation of the meta-language. Object level types are here called *sorts* to distinguish these from the meta languages types. Finally, but most importantly a pattern matching construct allows to write specifications that manipulate terms based on their abstract syntax tree.

expression	M, N	$::=$	x	Meta Variable
		$\mid$	$M\,N$	Meta Application
		$\mid$	$\lambda x\,.\,N$	Meta Abstraction
		$\mid$	$\mu x\ .\ M$	Recursive definition
		$\mid$	$\mathsf{Var}\,x$	Object Variable
		$\mid$	$\mathsf{App}\ M\,N$	Object Application
		$\mid$	$\mathsf{Lam}\ x\,.\,M$	Object Abstraction
		$\mid$	$\lambda(\mathsf{App}\ x\,y\ \Rightarrow\ M_1,\ \mathsf{Lam}\ x\,.\,y\ \Rightarrow\ M_2,\ \mathsf{Var}\,x\ \Rightarrow\ M_3)$	Pattern match expression

2.1 Reduction semantics

We recall the rules for β-reduction that besides the standard β rule also includes the induced rules for pattern matching.

$$\begin{array}{l}
\lambda(\mathsf{App}\ x\,y\ \Rightarrow\ M_1,\ \mathsf{Lam}\ x\,.\,y\ \Rightarrow\ M_2,\ \mathsf{Var}\,x\ \Rightarrow\ M_3)\ (\mathsf{App}\ M\,N)\ \longrightarrow_\beta\ M_1[x := M, y := N]\\
\lambda(\mathsf{App}\ x\,y\ \Rightarrow\ M_1,\ \mathsf{Lam}\ x\,.\,y\ \Rightarrow\ M_2,\ \mathsf{Var}\,x\ \Rightarrow\ M_3)\ (\mathsf{Lam}\ z\,.\,M)\ \longrightarrow_\beta\ M_2[x := z, y := M]\\
\lambda(\mathsf{App}\ x\,y\ \Rightarrow\ M_1,\ \mathsf{Lam}\ x\,.\,y\ \Rightarrow\ M_2,\ \mathsf{Var}\,x\ \Rightarrow\ M_3)\ (\mathsf{Var}\,z)\ \longrightarrow_\beta\ M_3[x := z]\\
(\lambda x\ .\ M)N\ \longrightarrow_\beta\ M[x := N]\\
\mu x\ .\ M\ \longrightarrow_\beta\ \mu x\ .(M[x := \mu x\ .M])
\end{array}$$

2.2 A typing discipline

Object terms are annotated with terms taken from the type sort. Meta terms are assigned the standard types, that range over simple types τ, type schemes σ, and rank-2 bounded types ρ^+, where polymorphism ranges over sort variables. We write $\Pi\vec{v} : \mathsf{sort}\ .\ \tau$ as a shorthand for $\Pi v_1 : \mathsf{sort}, \ldots, \Pi v_n : \mathsf{sort}\ .\ \tau$ where $\vec{v} = v_1, \ldots, v_n$. We will be mainly using distinguished subsets ρ' and ρ of ρ^+ and use that (closed) terms that can be assigned ρ^+ types can also be assigned ρ types.

sorts	s	$::=$	b_i	Base sort
		$\mid$	$s\ \Rightarrow\ s'$	Function space sort
		$\mid$	$u \mid v \mid w$	Sort variable

types	τ	$::=$	b_i	Base type
		$\mid$	$\tau\ \rightarrow\ \tau'$	Function space type
		$\mid$	$\mathsf{Term}[s]$	Object term sort
		$\mid$	$\mathsf{Var}[s]$	Object variable sort

$$\begin{array}{llll}
\textit{rank}-1 & \sigma & ::= & \Pi\vec{v} : \mathsf{sort}\ .\ \tau\\
\textit{rank}-1\tfrac{1}{2} & \rho' & ::= & \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau\\
\textit{rank}-2 & \rho & ::= & \Pi\vec{v} : \mathsf{sort}\ .\ \rho'
\end{array}$$

$$\textit{full rank}{-}2\quad \rho^+\ ::=\ \sigma \mid \Pi\vec{v} : \mathsf{sort}\ .\ (\sigma \rightarrow \rho^+)$$

The language is for simplicity limited to only introduce type abstraction over elements from sort. There is no polymorphism on the meta level types nor within object level expressions. Polymorphism does on the other hand come into play when building expressions that manipulate object level expressions. The typing calculus for this combination is thus completely analogous to those used for rank-2 typing as well as Mycroft's calculus for polymorphic recursion.

To type terms formed from App, Lam, and Var we introduce the constructors:

$$\Gamma_0\ :\ \left\{\begin{array}{l}
\mathsf{App} : \Pi v, w : \mathsf{sort}\ .\ \mathsf{Term}[v\ \Rightarrow\ w]\ \rightarrow\ \mathsf{Term}[v]\ \rightarrow\ \mathsf{Term}[w]\\
\mathsf{Lam} : \Pi v, w : \mathsf{sort}\ .\ \mathsf{Var}[v]\ \rightarrow\ \mathsf{Term}[w]\ \rightarrow\ \mathsf{Term}[v\ \Rightarrow\ w]\\
\mathsf{Var} : \Pi v : \mathsf{sort}\ .\ \mathsf{Var}[v]\ \rightarrow\ \mathsf{Term}[v]
\end{array}\right. \qquad (2)$$

Otherwise a term M has a type annotation in the full rank-2 fragment if there is a type ρ^+ and derivation of $\Gamma_0 \vdash M : \rho^+$ using the inference rules from Figures 1 and 2. For later reference we write $\Gamma_0 \vdash_0 M : \rho^+$ to indicate that there is a typing derivation for ρ^+ in this calculus.

We have conciously provided an explicit rule for fixpoints instead of introducing a distinguished family of constants $\mathbf{fix} : (\rho^+ \to \rho^+) \to \rho^+$ because this would have taken us out of the rank-2 fragment.

$$\texttt{VAR}_1 \quad \Gamma, x : \rho^+ \vdash x : \rho^+ \qquad\qquad \texttt{FIX}_1 \quad \frac{\Gamma, x : \rho^+ \vdash M : \rho^+}{\Gamma \vdash \mu x.M : \rho^+}$$

$$\texttt{APP}_1 \quad \frac{\Gamma \vdash M : \sigma \to \rho^+ \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \rho^+} \qquad\qquad \texttt{GEN} \quad \frac{\Gamma \vdash M : \rho^+}{\Gamma \vdash M : \Pi v : \mathsf{sort}\ .\ \rho^+} \quad v \notin FV(\Gamma)$$

$$\texttt{ABS}_1 \quad \frac{\Gamma, x : \sigma \vdash M : \rho^+}{\Gamma \vdash \lambda x.M : \sigma \to \rho^+} \qquad\qquad \texttt{INST}_1 \quad \frac{\Gamma \vdash M : \Pi v : \mathsf{sort}\ .\ \rho^+}{\Gamma \vdash M : \rho^+[s/v]}$$

Figure 1: Full rank-2 typing rules

The typing rule for pattern matching can be expressed in three variants, depending on whether the sort of the matched term is of sort b_i, v or $s_1 \Rightarrow s_2$. We give the most general case, where the matched sort is v, the other cases are instantiations of this case.

$$\frac{\begin{array}{ll}\Gamma, x : \mathsf{Term}[u \Rightarrow v],\ y : \mathsf{Term}[u] \ \vdash\ M_1 : \rho & u \notin FV(\Gamma) \\ \Gamma, x : \mathsf{Var}[u],\ y : \mathsf{Term}[w] \ \vdash\ M_2 : \rho[v \mapsto u \Rightarrow w] & u, w \notin FV(\Gamma) \\ \Gamma, x : \mathsf{Var}[v] \ \vdash\ M_3 : \rho & \end{array}}{\Gamma \vdash \lambda(\mathsf{App}\ x\, y \Rightarrow M_1,\ \mathsf{Lam}\ x\, .\, y \Rightarrow M_2,\ \mathsf{Var}\, x \Rightarrow M_3) : \mathsf{Term}[v] \to \rho}$$

Figure 2: Pattern matching typing rules

2.3 Derivability with reduced rank-2 types

To prepare the ground for type inference we will establish that we can replace the typing rules from Figure 1 by those from Figure 3 that only derive types in a subset of the rank-2 fragment and most importantly do not contain `INST` as a separate rule.

$$\texttt{VAR} \quad \Gamma, x : \Pi\vec{v} : \mathsf{sort}\ .\ \rho' \vdash x : \rho'[\vec{s}/\vec{v}]$$

$$\texttt{FIX} \quad \frac{\Gamma, x : \Pi\vec{v} : \mathsf{sort}\ .\ \rho' \vdash M : \rho'}{\Gamma \vdash \mu x.M : \rho'[\vec{s}/\vec{v}]}$$

$$\texttt{APP} \quad \frac{\Gamma \vdash M : \sigma \to \rho' \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \rho'} \qquad\qquad \texttt{GEN} \quad \frac{\Gamma \vdash M : \rho}{\Gamma \vdash M : \Pi v : \mathsf{sort}\ .\ \rho} \quad v \notin FV(\Gamma)$$

$$\texttt{ABS} \quad \frac{\Gamma, x : \sigma \vdash M : \rho'}{\Gamma \vdash \lambda x.M : \sigma \to \rho'}$$

Figure 3: Restricted rank-2 typing rules

For this calculus we write $\Gamma_0 \vdash M : \rho$ to indicate that there is a typing derivation of ρ for the term M using the rules from Figures 3 and 2.

Define the operation $\rho^\bullet$ as the normal form under the rewrite:

$$\sigma \rightarrow \Pi v : \mathsf{sort}.\rho \qquad \longrightarrow \qquad \Pi v : \mathsf{sort}.(\sigma \rightarrow \rho)$$

bound and free variables have been renamed apart to avoid the problems of variable capture. By $\Gamma^\bullet$ understand the context where each type annotation $x : \rho$ is lifted to $x : \rho^\bullet$. Clearly, if ρ^+ is of the full rank-2, then $\rho^{+\bullet}$ is of the restricted rank-2.

It is easy to establish invertibility

Lemma 2.1 (Invertibility) *If $\Gamma^\bullet \vdash M : \rho^{+\bullet}$ is derivable and $\rho^{+\bullet} = \Pi\vec{v} : \mathsf{sort} \ . \ \rho'$, then $\Gamma^\bullet \vdash M : \rho'$ is derivable too.*

By this we have a simplified version of a theorem from [KT92]

Theorem 2.2 (Calculus minimization)

$$\Gamma \vdash_1 M : \rho^+ \qquad \textit{iff} \qquad \Gamma^\bullet \vdash M : \rho^{+\bullet}$$

This establishes adequacy of our type system based on $\vdash$ instead of $\vdash_1$ to infer rank-2 bounded types.

2.4 Subject reduction

In order to establish that this calculus preserves types under β-normalization we make use of the lemmas:

Lemma 2.3 *If $\Gamma \vdash M : \rho$ then $\Gamma[s/v] \vdash M : \rho[s/v]$.*

Lemma 2.4 *If $\Gamma \vdash M : \Pi v : \mathsf{sort}.\rho$ then $\Gamma \vdash M : \rho[s/v]$.*

Lemma 2.5 *If $\Gamma \vdash M : \rho$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash M : \rho$.*

Lemma 2.6 *If $\Gamma, x_1 : \rho_1, \ldots, x_n : \rho_n \vdash M : \rho$, and each of $\Gamma \vdash N_i : \rho_i$, then $\Gamma \vdash M[x_i := N_i] : \rho$.*

In lemmas 2.3 and 2.4 we assume α-renaming of bound sort variables to avoid capture. The proofs of these lemmas are straight-forward, though we notice that the type rules for pattern matching have been formulated carefully such that they admit this sort specialization. In Lemma 2.6 we assume α-renaming of bound variables in terms to also avoid capture.

Theorem 2.7 (Subject reduction) *If $M \longrightarrow_\beta N$ and $\Gamma \vdash M : \sigma$ then $\Gamma \vdash N : \sigma$.*

As a corollary we obtain that if $\Gamma \vdash N : \mathsf{Term}[s]$ and $\Gamma \vdash M(N) : \mathsf{Term}[s']$ then any reduction of $M(N)$ preserves the derivability of sort $\mathsf{Term}[s']$ (Naturally, the type of M is $\mathsf{Term}[s] \ \rightarrow \ \mathsf{Term}[s']$).

2.5 Weak reification

As the object level expressions are themselves programs we are here interested to ensure that the object level programs have a typing derivation. The typing system we are using happens to be sufficient to to type the object level programs, when reflected at the meta-level, so we seek a property of the form:

$$\text{If } \Gamma \vdash M : \mathsf{Term}[s] \text{ then } \mathit{reify}(\Gamma) \vdash \mathit{reify}(M) : \mathit{reify}(s)$$

where we define reification in the obvious way:

$$\begin{array}{lcllcl}
\mathit{reify}(\mathsf{Lam}\ x \,.\, M) & = & \lambda x \ . \ \mathit{reify}(M) & \mathit{reify}(b_i) & = & b_i \\
\mathit{reify}(\mathsf{App}\ M\, N) & = & \mathit{reify}(M)\ \mathit{reify}(N) & \mathit{reify}(s \ \Rightarrow \ s') & = & \mathit{reify}(s) \rightarrow \mathit{reify}(s') \\
\mathit{reify}(\mathsf{Var}\, x) & = & x & \mathit{reify}(v) & = & b_v
\end{array}$$

and reification of a context Γ is obtained by reifying each type annotation $x : \mathsf{Term}[s]$ to $x : \mathit{reify}(s)$, respectively $x : \mathsf{Var}[s]$ to $x : \mathit{reify}(s)$.

We state the available theorem in more detail:

Theorem 2.8 (Weak Reification) *Assume M is built from only* Lam*,* App*, and* Var*. Then, if* $\Gamma \vdash M : \mathsf{Term}[s]$ *is derivable, then*

$$\mathit{reify}(\Gamma) \vdash \mathit{reify}(M) : \mathit{reify}(s)$$

is also derivable. Furthermore, if $\Gamma \vdash x : \mathsf{Var}[s]$ *is derivable, then* $(x : \mathsf{Var}[s]) \in \Gamma$

We call the theorem *Weak Reification* because the presented system comes with some deficiencies.

Variable capture Abstraction at the object level does not correspond directly to the scope conventions at the meta level. In particular, the effect of $\mathit{reify}(\mathsf{Lam}\ x\,.\,M)$ causes x to be bound while we make essential use of that it is free before reification.

Open code Consider the expression:

$$\lambda(\mathsf{Lam}\ z\,.\,u\ \Rightarrow\ u, u\ \Rightarrow\ \mathsf{App}\ u\,x)\ (\mathsf{Lam}\ y\,.\,\mathsf{Var}\,y)$$

It type checks with $\Gamma:\quad x : \mathsf{Term}[s],\ y : \mathsf{Var}[s]$, but after reduction we are left with $\mathsf{Var}\,y$, which is an open term. Hence, our type system does not distinguish between open and closed code. It may be another desirable feature of a type system to support this distinction.

3 Type inference

This section describes a method for inferring the necessary constraints that when solved give principal type schemes for meta programs. The constraint system generates a second-order semi-unification problem that is solved using a higher-order semi-unification algorithm.

3.1 Semi unification

A semi-unification problem is a set

$$\vec{s}_1 \le \vec{t}_1,\quad \vec{s}_2 \le \vec{t}_2,\quad \ldots\quad, \vec{s}_n \le \vec{t}_n$$

of pairs of term lists of the same length. A solution to is a collection of substitutions $R_1, \ldots, R_n, S$ such that $R_1(S(\vec{s}_1)) = S(\vec{t}_1)$, ..., $R_n(S(\vec{s}_n)) = S(\vec{t}_n)$. The shorthand $\vec{s} = \vec{t}$ is used for the two constraints $\vec{s} \le \vec{t}, \vec{t} \le \vec{s}$. The problem of determining if there is a solution is undecidable [KTU93] and is log-space inter-reducible to type inference for polymorphic recursion [Hen91].

We have extended a naive semi-unification search algorithm with higher-order unification [Hue76] to get a higher-order unification algorithm specialized to semi unification.

In addition to well-known reduction rules from higher-order unification and semi-unification we use a rule of the form:

$$\{x,\ (x\ t_1 \ldots t_n) \le x, u\} \cup C \quad\longrightarrow\quad \{(x\ y_1 \ldots y_n) = u,\ \ x,\ t_1,\ \ldots, t_n \le x, y_1, \ldots, y_n\} \cup C$$

The precise properties (completeness) of our unification algorithm variant are under investigation.

Alternatively to building semi-unification on top of higher-order unification one can naturally reformulate semi-unification as a higher-order unification problem by λ bindig the unifiable variables and replacing their occurrences by substitution variables applied to all of the bound variables. For example we change $f(X, g(Y))$ to $\lambda(x,y).f(\theta_1(x,y), g(\theta_2(x,y)))$, where θ_1 and θ_2 are the new unifiable variables. If we were to model application of two substitutions θ and ψ we would for example have

$$\lambda(x,y).f(\theta_1(\psi_1(x,y),\psi_2(x,y)), g(\theta_2(\psi_1(x,y),\psi_2(x,y)))). \tag{3}$$

So if $f(X, g(Y)) \le f(h(X), g(X))$ is a semi-unification constriant we reduce it to

$$(3) \le \lambda(x,y) f(h(\theta_1(x,y)), g(\theta_1(x,y))),$$

and can extract a most general solution

$$\psi_1 = \lambda(x,y)\ .\ h(x);\quad \psi_2 = \theta_1 = \lambda(x,y)\ .\ x;\quad \theta_2 = \lambda(x,y)\ .\ y.$$

3.2 Syntax directed accumulation of constraints

The inference algorithm [Wel93] and constraint system [Hen91] guide us to collect the necessary constraints for checking our terms. The extraction of constraints is performed using the function

$$\mathcal{C}[\Gamma; \vec{\tau} \vdash M] \implies \sigma, C$$

that takes a type context Γ, a set of type variables $\vec{\tau}$, and a term M. It produces a type σ and a set of semi unification constraints. We appeal to the typing rules in defining the effect of $\mathcal{C}$ on each possible subterm. To generate constraints that allow to infer rank-2 types we label lambda abstractions with numbers 1, 2, or 3 in terms using the call $\mathcal{L}(M, 0, 2)$ to the labeling function (see also [Wel93]):

$$\begin{array}{lcl}
\mathcal{L}(M\ N, k, d) & = & \mathcal{L}(M, k+1, d)\ \mathcal{L}(N, 0, 3) \\
\mathcal{L}(\lambda x\ .\ M, k, d) & = & \begin{cases} \lambda^d x.\mathcal{L}(M, k, d) & k = 0 \\ \lambda^1 x.\mathcal{L}(M, k-1, d) & k > 0 \end{cases} \\
\mathcal{L}(\mu x.M, k, d) & = & \mu x.\mathcal{L}(M, k, d) \\
\mathcal{L}(x, k, d) & = & x
\end{array}$$

$$\begin{array}{l}
\mathcal{L}(\lambda(\mathsf{App}\ x\, y\ \Rightarrow\ M_1,\ \mathsf{Lam}\ x\,.\,y\ \Rightarrow\ M_2,\ \mathsf{Var}\, x\ \Rightarrow\ M_3), k, d) = \\
\begin{cases} \lambda^d(\mathsf{App}\ x\, y\ \Rightarrow\ \mathcal{L}(M_1, k, d),\ \mathsf{Lam}\ x\,.\,y\ \Rightarrow\ \mathcal{L}(M_2, k, d),\ \mathsf{Var}\, x\ \Rightarrow\ \mathcal{L}(M_3, k, d)) & k = 0 \\ \lambda^1(\mathsf{App}\ x\, y\ \Rightarrow\ \mathcal{L}(M_1, k-1, d),\ \mathsf{Lam}\ x\,.\,y\ \Rightarrow\ \mathcal{L}(M_2, k-1, d),\ \mathsf{Var}\, x\ \Rightarrow\ \mathcal{L}(M_3, k-1, d)) & k > 0 \end{cases}
\end{array}$$

The labeling has the property of pairing off abstractions with their bodies and indicate which abstractions can be polymorphically unconstrained and which not. In summary the abstractions labeled by 1 have companions, those labeled by 2 are unconstrained, and those labeled by 3 must be treated as monomorphic to generate constraints that admit rank-2 annotations. (It can be shown that the normalization functions θ_1, θ_2, θ_3, and θ_4 from [Wel93] preserve the labeling induced by $\mathcal{L}$ so we can in fact ignore the normalization provided there.)

$$\mathcal{C}[\Gamma, x : \tau; \vec{\tau} \vdash x] \implies \alpha, \{(\tau, \vec{\tau}) \leq (\alpha, \vec{\tau})\}$$

$$\frac{\mathcal{C}[\Gamma, x : \alpha; \vec{\tau} \vdash M] \implies \tau, C}{\mathcal{C}[\Gamma; \vec{\tau} \vdash \lambda^{1,2} x\,.\,M] \implies (\alpha\ \rightarrow\ \tau), C} \qquad \frac{\mathcal{C}[\Gamma, x : \alpha;\ \alpha\vec{\tau} \vdash M] \implies \tau, C}{\mathcal{C}[\Gamma; \vec{\tau} \vdash \lambda^3 x\,.\,M] \implies (\alpha\ \rightarrow\ \tau), C}$$

$$\frac{\mathcal{C}[\Gamma; \vec{\tau} \vdash M] \implies \tau_1, C_1 \qquad \mathcal{C}[\Gamma; \vec{\tau} \vdash N] \implies \tau_2, C_2}{\mathcal{C}[\Gamma; \vec{\tau} \vdash M\ N] \implies \alpha,\ C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \rightarrow \alpha\}} \qquad \frac{\mathcal{C}[\Gamma, x : \alpha; \vec{\tau} \vdash M] \implies \tau, C_1 \qquad C = C_1 \cup \{\alpha = \tau,\ (\alpha, \vec{\tau}) \leq (\beta, \vec{\tau})\}}{\mathcal{C}[\Gamma; \vec{\tau} \vdash \mu x.M] \implies \beta,\ C}$$

$$\frac{\begin{array}{c}
\mathcal{C}[\Gamma, x : \mathsf{Term}[b_i\ \Rightarrow\ v_p], y : \mathsf{Term}[b_i];\ [v_p,]\rho, \vec{\tau} \vdash M_1] \implies \tau_1, C_1 \qquad b_i \text{ is fresh} \\
\mathcal{C}[\Gamma, x : \mathsf{Var}[v_p];\ [v_p,]\rho, \vec{\tau} \vdash M_2] \implies \tau_2, C_2 \\
\mathcal{C}[\Gamma, x : \mathsf{Var}[u], y : \mathsf{Term}[w];\ [v_p,]\rho, \vec{\tau} \vdash M_3] \implies \tau_3, C_3 \qquad u,\ w \text{ are fresh} \\
C = C_1 \cup C_2 \cup C_3 \cup \{\tau_1\ =\ \tau_2\ =\ \mathsf{Term}[\rho(v_p)], \mathsf{Term}[\rho(u\ \Rightarrow\ w)] = \tau_3\} \\
[v_p] = \mathbf{if}\ \ d = 3\ \mathbf{then}\ \ v_p\ \mathbf{else}\ \epsilon
\end{array}}{\mathcal{C}[\Gamma; \vec{\tau} \vdash \lambda^d(\mathsf{App}\ x\, y\ \Rightarrow\ M_1,\ \mathsf{Lam}\ x\,.\,y\ \Rightarrow\ M_2,\ \mathsf{Var}\, x\ \Rightarrow\ M_3)] \implies \mathsf{Term}[v_p] \rightarrow \mathsf{Term}[\rho(v_p)], C}$$

The variables α, β (which range over types), u, v, w, v_p, (which range over sorts), and ρ (which ranges over functions from sorts to sorts) introduced in the rules are all fresh.

For a closed term M we generate constraints C by applying $\mathcal{C}[\Gamma_0, \epsilon \vdash M]$, where the sort abstractions have been deleted from Γ_0 (defined in (2)).

3.3 Constraint solving

We solve the accumulated constraints C for the term M by iterating the following:

Initial solution Find a most general semi-unifier $\overline{R}$, S for C.

Sort specialization When $S(v_p) = s_1 \Rightarrow s_2$ for some v_p, then add the constraints $u = s_1$, $w = s_2$ to C. This ensures that if the domain of the pattern match expression instantiates to a function space, then the case for Lam is compatible with this type. In other words, it reduces the constraints to those associated with the $s_1 \Rightarrow s_2$ version of the typing rule for pattern match expressions.

Domain restriction If there is a variable v (or α) not in $FV(S(\vec{t_i}))$ for any of the right-hand sides of the semi-unification problem, such that the least specific generalizer θ of $R_1(v), \ldots, R_n(v)$ is not a variable, then set $S := \theta \circ S$.

For the remaining variables v_p, such that $S(v_p) = v_p$ check that $S(u) = u$, and $S(w) = w$. This ensures that any instantiation of v_p to a functionspace is compatible with the typing constraints on the subcase for abstraction.

The domain restriction step refines a principal solution to one where a maximally compositional typing is produced. We describe this situation in more detail. If the constraint system associates the variables $\alpha_1, \ldots, v_i, \ldots, \alpha_n$ with the unconstrained labeled abstractions (λ^2), then the resulting type of M is of the form $S(\alpha_1 \to \ldots \to \mathsf{Term}[\ldots v_i \ldots] \to \ldots \to \alpha_n \to \rho')$. Furthermore, since the constraint system does not place any of the α_i in an equation (because they are introduced for bound variables without companions), we have $S(\alpha_i) = \alpha_i$ for a principal S. It is possible to instantiate each of the α_i by $\forall\beta \,.\, \beta$, but this makes the typing of M minimally compositional: there are no well-typed terms with type $\forall\beta \,.\, \beta$, so M cannot be typed separately from its application points. The domain restriction step ensures that the α_i are specialized according to the uses of x_i in the body of M[2].

4 The example revisited

Recall the `map` function from the introduction

```
map f (App(M,N)) = f(App(map f M,map f N))
map f (Lam(M,N)) = f(Lam(M,map f N))
map f (Var M)    = f(Var(M))
```

It can be defined as a closed term using the recursion operator μ:

$$\mu map.\lambda f \,.\, \lambda(\mathsf{App}\; x\, y \Rightarrow f(\mathsf{App}\;(map\; f\; x)\,(map\; f\; y)),\; \mathsf{Lam}\; x \,.\, y \Rightarrow f(\mathsf{Lam}\; x \,.\, (map\; f y)),\; \mathsf{Var}\; x \Rightarrow f(\mathsf{Var}\; x))$$

Recall also the desired principal type

$$map : \Pi w : \mathsf{sort} \,.\, ((\Pi v : \mathsf{sort}.\mathsf{Term}[v] \to \mathsf{Term}[v]) \to \mathsf{Term}[w] \to \mathsf{Term}[w])$$

We supply $\mathcal{C}$ with the initial argument $\Gamma_0, \epsilon \vdash \mathcal{L}(M, 0, 2)$ and apply the rules for $\mathcal{C}$ bottom-up. The result of labeling M is:

$$\mu map.\lambda^2 f \,.\, \lambda^2(\mathsf{App}\; x\, y \Rightarrow \ldots) \tag{4}$$

The first rule requires to generate the constraints for polymorphic recursion. We therefore introduce the sort map_τ for the bound variable map. Similarly we associate the sort f_τ with the second bound variable f. The third rule to apply uses the pattern matching construct and introduces variables $v_1, \ldots v_3, v, \rho$. More invocations of $\mathcal{C}$ are made for each sub-case. In summary, the rules for $\mathcal{C}$ determine the constraints:

[2] As a technical aside, we would furthermore have to extend our system with polymorphism over types to allow types of the form $\forall\beta \,.\, \beta$, so for the more restricted type system we have to reject solutions that require polymorphism over types.

$$\begin{array}{lcl}
map_\tau & = & f_\tau \to \mathsf{Term}[v] \to \mathsf{Term}[\rho\, v] \\
\mathsf{Term}[\rho\, v] & = & \tau_1 \\
\mathsf{Term}[\rho\, v] & = & \tau_2 \\
\mathsf{Term}[\rho\, (u \Rightarrow w)] & = & \tau_3 \\
 & & \\
\rho, map_\tau & \leq & \rho, f_\tau \to \mathsf{Term}[b_i \Rightarrow v] \to \mathsf{Term}[v_1 \Rightarrow v_2] \\
\rho, map_\tau & \leq & \rho, f_\tau \to \mathsf{Term}[b_i] \to \mathsf{Term}[v_1] \\
\rho, f_\tau & \leq & \rho, \mathsf{Term}[v_2] \to \tau_1 \\
 & & \\
\rho, f_\tau & \leq & \rho, \mathsf{Term}[v] \to \tau_2 \\
 & & \\
\rho, map_\tau & \leq & \rho, f_\tau \to \mathsf{Term}[w] \to \mathsf{Term}[v_3] \\
\rho, f_\tau & \leq & \rho, \mathsf{Term}[u \Rightarrow v_3] \to \tau_3
\end{array}$$

We notice that $\forall\alpha.\alpha$ is a solution for f_τ, but also $\Pi s : \mathsf{sort}.\mathsf{Term}[s] \to \mathsf{Term}[s]$ which is the least specific generalization obtained from the inequalities for f_τ.

In summary we (and our prototype semi-unification implementation) get the principal solution:

$$\begin{array}{c}
v_1 = b_i; \quad v_2 = v; \quad v_3 = w; \\
\rho = \lambda x.x; \quad \tau_1 = \mathsf{Term}[v]; \quad \tau_2 = \mathsf{Term}[v]; \quad \tau_3 = \mathsf{Term}[u \Rightarrow w]; \\
map_\tau = f_\tau \to \mathsf{Term}[v] \to \mathsf{Term}[v]
\end{array}$$

by first solving the semi-unification problem, and then extracting the type

$$\Pi v : \mathsf{sort}\ .\ \mathsf{Term}[v] \to \mathsf{Term}[v]$$

for f_τ as we are taking the least specific generalization of the types $\mathsf{Term}[v] \to \mathsf{Term}[v]$, $\mathsf{Term}[v] \to \mathsf{Term}[v]$, and $\mathsf{Term}[u \Rightarrow w] \to \mathsf{Term}[u \Rightarrow w]$.

4.1 Beyond a single example

To give a taste of additional features that are required to infer types for slightly more elaborate programs we summarize a simple example that has appeared in an optimization transformation in our C generator.

The optimization steps consists in transforming code of the form:

$$\mathbf{let}\ x = (M_1, M_2)\ \mathbf{in}\ M_3[\mathsf{First}(x), \mathsf{Second}(x), x]$$

to a partially reduced version

$$\mathbf{let}\ y_1 = M_1;\ y_2 = M_2; x = (y_1, y_2)\ \mathbf{in}\ M_3[y_1, y_2, x]$$

where we have introduced the additional constructors together with the corresponding pattern-maching rules:

$$\begin{array}{l}
\mathsf{Pair} : \Pi v_1, v_2 : \mathsf{sort}\ .\ \mathsf{Term}[v_1] \to \mathsf{Term}[v_2] \to \mathsf{Term}[v_1 \times v_2] \\
\mathsf{First} : \Pi v_1, v_2 : \mathsf{sort}\ .\ \mathsf{Term}[v_1 \times v_2] \to \mathsf{Term}[v_1] \\
\mathsf{Second} : \Pi v_1, v_2 : \mathsf{sort}\ .\ \mathsf{Term}[v_1 \times v_2] \to \mathsf{Term}[v_2]
\end{array}$$

into the language.

We can now write our transformation function in a slightly sugared extension of our core language. First, the auxiliary

$\mathsf{mkLet}(x, M, N) = \mathsf{App}\ (\mathsf{Lam}\ x\ .\ N)\ M$

from which we easily infer the typing

$\mathsf{mkLet} : \Pi v_1, v_2 : \mathsf{sort}\ .\ \mathsf{Var}[v_1] \times \mathsf{Term}[v_1] \times \mathsf{Term}[v_2] \ \rightarrow\ \mathsf{Term}[v_2]$.

To reduce pairs from one occurrence we give the function reducePairs, it assumes the existence of a function fresh : $\Pi v : \mathsf{sort}\ .\ \mathsf{Term}[v] \ \rightarrow\ \mathsf{Var}[v]$ to generate fresh variables. More critically, it is necessary to either extend pattern-matching with equational constraints or allow type inference to take conditions into account when typing the forks of an if-then-else statement. We adapt the former approach and introduce the construct freeze which causes the compiler to replace a pattern $\lambda(\mathsf{Var}\,(\mathsf{freeze}(t)) \Rightarrow M_1, \ldots)$ by a *when* clause (as found in for instance Ocaml): $\lambda(\mathsf{Var}\, y\ \mathit{when}\ (t = y)\ \Rightarrow M_1, \ldots)$. The sort of the branch containing freeze is naturally now constrained to be that of t, which is essential for the type inference to succeed.

We are now ready for the definition of reducePairs.

```
reducePairs(App (Lam $x$ . $M_3$) (Pair($M_1, M_2$))) =
    let $y_1$ = fresh($M_1$)
        $y_2$ = fresh($M_2$)
        replace = $\lambda$ (First(Var (freeze($x$)))) $\Rightarrow y_1$ | Second(Var (freeze($x$))) $\Rightarrow y_2$ | $t \Rightarrow t$)
        $M_3'$ = map replace $M_3$
    in
        mkLet($y_1, M_1$,
            mkLet($y_2, M_2$,
                mkLet($x$, Pair($y_1, y_2$), $M_3'$)))

reducePairs($t$) = $t$
```

It is possible to now generate type constraints and infer the type reducePairs : $\Pi v : \mathsf{sort}\ .\ \mathsf{Term}[v] \ \rightarrow$ $\mathsf{Term}[v]$. The let-bound variable replace, which by the labeling algorithm $\mathcal{L}$ is labeled by 1, is given the type: $\Pi v : \mathsf{sort}\ .\ \mathsf{Term}[v] \ \rightarrow\ \mathsf{Term}[v]$, which naturally matches they way it is used in map.

5 Conclusions

We have presented a calculus and type inference system together with a little experimental evidence that the sought typability problem is within reach of even simple solvers for higher-order semi-unification problems.

For the presented approach we have kept the meta-language and object language monomorphic on their own domain of types and sorts. Besides preventing typability of let-polymorphism it also prevents us from checking type correctness for programs that manipulate programs that manipulate programs. For instance, if we optimize our meta-programs using themselves we would like to ensure that the resulting programs also preserve typability of the object programs that they manipulate. This can be checked by running the type inference on the generated output, but we have not provided a way for dealing with this statically.

In some of our translation examples the meta programs do also manipulate and build object level sorts. This takes us directly out of the proposed type system and leads in general to highly intractable type inference problems. We are nevertheless hopeful that it will be possible to find reasonable restrictions where this added functionality is isolated in a controllable way.

In general we may think of multi-level type systems for staged meta computation as described in the forthcoming [Tah99]. The more general type system described here may on the other hand enable writing multi-stage programs that also access the abstract syntax of stages at a lower level. It will however be necessary to strengthen the type system to distinguish open and closed code for a sound integration with multi-staged programming.

Acknowledgements: We thank Walid Taha, who pointed out the distinction between open and closed code, David Espinosa, Doug Smith, and César Muñoz for stimulating discussions, and Richard Waldinger for equally stimulating coffee.

References

[AC98] R.M. Amadio and P.L. Curien. *Domains and Lambda-Calculi*, vol. 46 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1998.

[Aug99] L. Augustsson. Cayenne — a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, vol. 34(1) of *ACM SIGPLAN Notices*, pages 239–250. ACM, June 1999.

[Hen91] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, January 1991.

[Hue76] G. Huet. Résolution d'équations dans des langages d'ordre 1, 2, ... ω. Thèse de doctorat, Université de Paris VII, Paris, France, 1976.

[KT92] A.J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and Computation*, 98(2):228–257, June 1992.

[KTU93] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993.

[PS98] F. Pfenning and C. Schürmann. Twelf User's Guide, 1.2 edition. Technical Report CMU-CS-98-173, Carnegie Mellon University, September 1998.

[SJ95] Y.V. Srinivas and R. Jüllig. Specware: Formal support for composing software. *Lecture Notes in Computer Science*, 947:399–422, 1995.

[Tah99] W. Taha. *Multi-Stage Programming: Its theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999.

[Wel93] J. Wells. A direct algorithm for type inference in the rank 2 fragment of the second-order lambda-calculus. Technical Report 93-017, Boston University, November 1993.

[XP99] H. Xi and F. Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM, January 1999.