Safe polymorphic type inference for a Dynamically Typed Language: Translating Scheme to ML[*]

Fritz Henglein and Jakob Rehof
DIKU, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East, Denmark
Email: {henglein,rehof}@diku.dk

April 18, 1995

Abstract

We describe a new method for polymorphic type inference for the dynamically typed language Scheme. The method infers both types and explicit *run-time type operations (coercions)* for a given program. It can be used to statically debug Scheme programs and to give a high-level translation to ML, in essence providing an "embedding" of Scheme into ML.

Our method combines the following desirable properties:

- It is *liberal* in that *no* legal Scheme programs are "rejected" outright by the type inferencer.
- It is modular in that definitions can be analyzed independent of the context of their use. The inferred type scheme for a definition is *safe* for all contexts. This is accomplished by admitting *type coercion parameters*, resulting in a form of polymorphic qualified type system.
- It infers both types and run-time type operations and places the latter opportunistically at any suitable program point, not only at data creation and destruction points.
- It is very efficient. Its monomorphic fragment can be implemented in almost-linear time, and the full polymorphic calculus appears to be comparable with ML type inference in practice.
- It is *conservative* over simple and ML-polymorphic typing in that the type inferencer inserts *no* run-time type operations for ML-typable programs (it may add some coercion parameters, however).
- It classifies subexpressions as either definitely type correct (requiring no run-time type checking), possibly type correct (requiring run-time type checking), or definitely type incorrect (definitely aborting).

Our method is based on the formal type theoretic framework of *dynamic typing*, generalized to include polymorphic type coercions, recursive

[*]To appear in Proc. ACM SIGPLAN Conf. on Functional Programming Languages and Computer Architecture (FPCA), La Jolla, California, June 26-28, 1995

types, and polymorphic *coercive* types. In this regard it is most closely related to the soft typing systems of Cartwright, Fagan, Wright and Aiken, Wimmers, Lakshman.

The main technical contributions reported in this work are two-fold:

1. *Synthesis of soft and dynamic typing:* It extends monomorphic dynamic typing to polymorphic coercions from and to discriminative sum types and recursive types. Thus it subsumes the typing power of Cartwright, Fagan and Wright's soft typing discipline and combines it with the ability of dynamic typing to place coercions anywhere at all, not only at data creation and destruction points. Despite admitting polymorphic coercions and their arbitrary placement, monomorphic inference of *minimal* completions is implementable in almost-linear time, preserving the best known time bounds from dynamic type inference.
2. *Safe modular type inference:* Definitions can be type checked safely and minimally, independent of any particular context, using coercion parameters. This solves an outstanding problem with the Cartwright, Fagan, Wright soft typing discipline. The number of coercion parameters is minimized by eliminating and identifying as many coercion parameters as possible before abstracting over them.

The result of type and coercion inference can be translated into a type correct ML program. Based on this we have developed a prototype Scheme-to-ML translator for a subset of IEEE Scheme. In this paper we give an introduction to the type theoretic framework of polymorphic dynamic typing, describe the phases of the type inference process, and give some examples of the resulting Scheme-to-ML translator.

1 Introduction

High-level translation between statically and dynamically typed programming languages is a notoriously tricky business, even for semantically and syntactically — seemingly — "compatible" language families such as Scheme and ML. There are several reasons for being interested in a high-level translation, instead of a low-level translation or direct compilation to machine code.

Systems written in several different programming languages have to communicate through the operating system with each other. This may be undesirable for several reasons:

- It adds dependency and vulnerability on a particular operating system. In particular it adds additional costs to software maintenance and porting.
- Communication through the operating system is low-level and thus error-prone as well as relatively inefficient.

Porting code from one language to another is a way of preserving the investment made in the ported part of the system. For porting purposes, a high-level translation is an absolute must. It should be geared primarily towards readability and maintainability, only secondarily towards reasonable efficiency. It need not even be complete so long as it handles a well-defined and sufficiently interesting subset of the ported language completely and correctly.

We are interested in embedding a dynamically typed language such as Scheme within a statically typed language. The principal advantages of this are two-fold: most run-time type operations are eliminated statically, and data can — at least in principle — be stored in a type-specific fashion. For example, a pair of small integers can be given an untagged/unboxed representation, or a list of sixteen elements, each of known type, can be allocated like a Pascal record. Most importantly, however, we can *statically debug* the Scheme code by identifying definite and possible type errors without executing the code.

2 Dynamic type inference with polymorphic type coercions

This section introduces the formal framework of our completion inference system with polymorphic completions. An in-depth coverage can be found in Rehof's Master's thesis [Reh95]. In this paper we develop our formal framework only for a core part of the language in order to keep the presentation manageable.

2.1 Polymorphic completions

In this section we introduce our framework of *polymorphic type coercions* and *completions*. Since we wish our coercions to model the run-time type checking and tagging operations of Scheme we start from the observation that Scheme's dynamic types are given by the *type tags* ranged over by *tc*: (See the language definition, [CR91])

$$\begin{array}{rcl} tc & ::= & \texttt{nil} \mid \texttt{boolean} \mid \texttt{symbol} \mid \texttt{char} \mid \texttt{vector} \\ & \mid & \texttt{pair} \mid \texttt{number} \mid \texttt{string} \mid \texttt{procedure} \end{array}$$

These types divide the data domain into nine disjoint sets. Of course, the full Scheme language contains yet other types, such as, *e.g.*, input and output ports, but here we restrict our attention to the core types above.

In the manner of [Hen92a, Hen92b, Hen94] we assume the following *primitive* type coercions: For every type tag *tc*, a primitive *tagging* coercion called `tc!` and a primitive *check-and-untag* coercion called `tc?`. In addition, we assume an *identity* coercion `id`. Intuitively, the tagging coercions are embedding functions which add the specific type tag *tc* to an object; the check-and-untag coercions project an object back: The operation `tc?` inspects its tagged argument, checks whether it has tag *tc* and, if so, strips it off and returns the underlying (untagged) object; if another tag is found then a *run-time type error* is generated. This typically means a program abort or escape to the interactive top-level loop, with some error message to indicate the reason for the abort. The identity coercion is just a no-op.

For example, executing `[boolean!] #t` evaluates to a tagged boolean object $\langle \text{bool}, \textit{true} \rangle$, where bool is a representation for the type `boolean`. (In implementations tags use the 2 or 3 low-order or high-order bits of a machine address word, with additional, more elaborate tagging in the pointed-to memory region.) Now, executing

```
[boolean?] ([boolean!] #t)
```

first evaluates `[boolean!] #t` to $\langle \text{bool}, \mathit{true} \rangle$ and then applies the check-and-untag operation `boolean?` to this value. Since the tag in $\langle \text{bool}, \mathit{true} \rangle$ is the expected one, the underlying untagged value, *true*, is returned. Note that this is also the result of evaluating `#t`. In other words, semantically we have

```
[boolean?] ([boolean!] #t) = #t
```

It should be clear, however, that the right-hand side is more *efficient* than the left-hand side. Applying another check-and-untag, such as `procedure?` to `[boolean!] #t` results in a run-time error.

We assume a Hindley-Milner style polymorphic *type language* enriched with *regular recursive* types and *discriminative sums*, in the style of *soft typing* [Fag90, CF91, Wri94, WC94]; in addition, we introduce the notion of *coercive types*:

$$\begin{array}{rcl} \tau & ::= & \alpha \mid tc^{(n)}(\tau_1, \ldots, \tau_n) \mid \mu\alpha.\tau \mid \sum_{tc} tc(\overline{\tau}) \\ \omega & ::= & \tau \mid (\tau \leadsto \tau) \Rightarrow \omega \\ \sigma & ::= & \omega \mid \forall\alpha.\sigma \end{array}$$

Here $tc^{(n)}$ ranges over type tags of arity n. We treat the `procedure` tag as a binary constructor, and we write $\tau \to \tau'$ for $\texttt{procedure}(\tau, \tau')$; also, we write $\tau * \tau'$ for $\texttt{pair}(\tau, \tau')$. Recursive type abstractions of the form $\mu\alpha.\tau$ are finite notations for possibly infinite, regular trees, as in [CC91].

Types of the form $\sum_{tc} tc(\overline{\tau})$ are discriminative sums, where the top level type constructors of the summands are required to be distinct (see [Reh95] and also [Fag90] for more details.) In this paper, we generally assume that tc ranges over the whole type constructor alphabet (denoted T) in a sum.[1] We use k as a parameter for the number of constructors. Moreover, we assume a fixed ordering of the type constructors in T such that the summands are always listed according to that order. We think of T as indexed by $I = \{1, \ldots, k\}$. Restrictions such as these result in a simplification which is important for practical purposes, as it is tantamount to turning the sum type construction into a "free" type constructor. The literature on soft typing (mentioned above) contains good discussions of these issues; see also [Reh95] for more detailed information. The notation $tc(\overline{\tau})$ denotes the application of a type constructor to a type vector $(\overline{\tau})$ where it is tacitly assumed that the length of the vector matches the arity of the type constructor.

Types ranged over by τ are *monotypes*, and types ranged over by σ are *type schemes*, as usual. However, in comparison with standard Hindley-Milner systems, we have further stratified the type language with an additional class, ranged over by ω, of *coercive types*, in order to accommodate *polymorphic coercion parameters*, as explained below. We sometimes use the abbreviation s to range over *coercion signatures*, which are expressions of the form $\tau \leadsto \tau'$.

It is worth noting that our coercive types (ω) are a special case of Jones' *qualified types* ([Jon92, Jon94]). It would be interesting to apply his general framework to the present application.

[1]This is due to the fact that Standard ML, the target of our translation, only has nonextensible sum types with name equivalence, called *datatypes*.

Coercions are assigned *polymorphic signatures* by the following axiom schemes:

$\mathtt{tc_j}! : tc_j(\tau_j) \leadsto \sum_{tc_i \in T} tc_i(\overline{\tau}_i)$
$\mathtt{tc_j}? : \sum_{tc_i \in T} tc_i(\overline{\tau}_i) \leadsto tc_j(\tau_j)$
$\mathtt{id} : \tau \leadsto \tau$

where $tc_j \in T$. (Recall that we assume that T denotes the set of *all* type constructors.)

For instance, the tagging coercion `procedure!` has signature of the form

$$(\tau \to \tau') \leadsto (\tau \to \tau') + \sum tc_i(\overline{\tau}_i)$$

such that the type constructor $\to$ is *not* one of the tc_i in $\sum tc_i(\overline{\tau}_i)$. The intention is that this coercion can be applied to any object of functional type, and such that the resulting object can be regarded as the element of a corresponding sum.

Thus, sum types can be thought of as the types for all tagged objects. In this respect sum types are like the universal type Dyn [Hen92a, Hen92b, Hen94]. Note, however, that the present framework is considerably more powerful. In the Dyn-based type system a tagging operation `tc!` is only applicable to objects of type $tc(\mathsf{Dyn}, \ldots, \mathsf{Dyn})$. This forces the components of an object to be tagged if the whole object is to be tagged, resulting in a *cascading* effect of tagging and untagging operations.

To illustrate, suppose we discover that some subexpression $\lambda x.M$ needs to be tagged (with `procedure!`) within some larger context. In a monomorphic discipline, this requires that we complete the abstraction at type $\mathsf{Dyn} \to \mathsf{Dyn}$, and this, in turn, can lead to the need for further tags and checks inside the body M, even though the abstraction, by itself (in isolation from the context), could be completed without any coercions. This problem is overcome by considering the coercions as *polymorphic constants*, where, say, the tag `procedure!` can be applied to an object of *arbitrary* functional type. Discriminativity, though, requires that two functions embedded into the same sum type must have identical types.

In addition to the primitive coercions above, we have *induced coercions*, as follows: For every tag tc of arity $n > 0$, and coercions `c1`, ..., `cn` there is an induced coercion $tc(\mathtt{c_1}, \ldots, \mathtt{c_n})$. In case tc is the functional tag, the signature of the corresponding induced coercion, written $\mathtt{c} \to \mathtt{d}$, is contra-variant in the signature of the first coercion argument:

$$\frac{\mathtt{c} : \tau \leadsto \tau' \qquad \mathtt{d} : \tau'' \leadsto \tau'''}{\mathtt{c} \to \mathtt{d} : (\tau' \to \tau'') \leadsto (\tau \to \tau''')}$$

In all other cases, the signatures of `c1, ..., cn` extend co-variantly to the signature of the induced coercion:

$$\frac{\mathtt{c1} : \tau_1 \leadsto \tau_1' \ldots \mathtt{cn} : \tau_n \leadsto \tau_n'}{tc(\mathtt{c1}, \ldots, \mathtt{cn}) : tc(\tau_1, \ldots, \tau_n) \leadsto tc(\tau_1', \ldots, \tau_n')}$$

Coercions may also be formed by *composition* ($\circ$):

$$\frac{\mathtt{c} : \tau \leadsto \tau' \qquad \mathtt{d} : \tau' \leadsto \tau''}{\mathtt{d} \circ \mathtt{c} : \tau \leadsto \tau''}.$$

Finally, we assume an infinite supply of *coercion parameters* (variables) ranged over by π, as a vehicle for coercion abstraction and application in polymorphic completions.

We develop our formal framework in this paper only for a core fragment of Scheme, called CoreScheme, defined as follows, where `M`, `N`, `P` range over CoreScheme expressions:

$$\begin{array}{lll} \mathtt{M} & ::= & \mathtt{x} \mid \mathtt{true} \mid \mathtt{false} \mid (\mathtt{if}\,\mathtt{M}\,\mathtt{N}\,\mathtt{P}) \mid \\ & & (\mathtt{lambda}\,(\mathtt{x})\,\mathtt{M}) \mid (\mathtt{M}\,\mathtt{N}) \mid (\mathtt{let}\,\mathtt{x}\,\mathtt{M}\,\mathtt{N}) \mid (\mathtt{define}\,\mathtt{x}\,\mathtt{M}) \end{array}$$

Here the `define` construct allows recursive definitions.

In this paper we construct *completions* of CoreScheme expressions. A completion of an expression `M` arises from insertion of coercions into `M`, in a manner which is disciplined by a type system, to be described. A CoreScheme expression without any coercions is sometimes referred to as a *pure* expression. To define the set of completions we first define formally the set of coercions introduced above, ranged over by c, d, as follows:

$$c \;\; ::= \;\; \pi \mid \mathtt{id} \mid \mathtt{tc!} \mid \mathtt{tc?} \mid \mathtt{tc}^{(n)}(c_1, \ldots, c_n) \mid c \circ d$$

The set of completions, then, is given by continuing the definition of CoreScheme expressions above with

$$\mathtt{M} \;\; ::= \;\; \ldots \mid [\mathtt{c}]\mathtt{M} \mid \Lambda\pi : s.\mathtt{M} \mid \mathtt{M}\{\mathtt{c}\}$$

defining, respectively, *coercion application, coercion abstraction* and *coercion instantiation.*

2.2 Type system

The type system is given in Figure 1. It defines the set of *well typed completions.* It is a standard Hindley-Milner polymorphic system extended, in the last five rules, with rules for recursive types, coercion application, coercion abstraction, coercion instantiation, and definitions, respectively. In the recursion rule (ninth rule from top), the relation $\tau \approx \tau'$ holds, if and only if the (possibly) infinite regular unfoldings of τ and τ' are identical. See [CC91] for further information. In the rule for definitions (last rule), the expression $C \Rightarrow \tau$ denotes type $s_1 \Rightarrow \ldots s_n \Rightarrow \tau$, where C is the set of coercion signatures $s_1, \ldots, s_n$.

Note that any pure expression which is well typed is a completion (of itself.) In the presence of recursive types, it is immediate that any term of the pure λ-calculus is a completion of itself.[2]

2.3 Safety, minimality and modularity

There may be several completions of the same term at the same type. Not all of these will be equally good, and a completion inference system must be judged by the quality of the completions it infers. *Safety* and *minimality* are the parameters on which we judge our completions.

[2]Prove by structural induction on λ-terms that every term has the type $\mu\alpha.\alpha \rightarrow \alpha$

$$\Gamma, \mathtt{x} : \sigma \vdash \mathtt{x} : \sigma$$

$$\Gamma \vdash \chi : \mathit{bool} \quad , \chi \in \{\mathtt{true}, \mathtt{false}\}$$

$$\frac{\Gamma, x : \tau \vdash \mathtt{M} : \tau'}{\Gamma \vdash (\mathtt{lambda}\,(\mathtt{x} : \tau)\,\mathtt{M}) : \tau \rightarrow \tau'}$$

$$\frac{\Gamma \vdash \mathtt{M} : \tau \rightarrow \tau' \quad \Gamma \vdash \mathtt{N} : \tau}{\Gamma \vdash (\mathtt{M}\,\mathtt{N}) : \tau'}$$

$$\frac{\Gamma \vdash \mathtt{M} : \mathit{bool} \quad \Gamma \vdash \mathtt{N} : \tau \quad \Gamma \vdash \mathtt{P} : \tau}{\Gamma \vdash (\mathtt{if}\,\mathtt{M}\,\mathtt{N}\,\mathtt{P}) : \tau}$$

$$\frac{\Gamma \vdash \mathtt{M} : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \mathtt{M} : \forall \alpha.\sigma}$$

$$\frac{\Gamma \vdash \mathtt{M} : \forall \alpha.\sigma}{\Gamma \vdash \mathtt{M} : \sigma\{\alpha := \tau\}}$$

$$\frac{\Gamma \vdash \mathtt{M} : \sigma \quad \Gamma, x : \sigma \vdash \mathtt{N} : \sigma'}{\Gamma \vdash (\mathtt{let}\,\mathtt{x}\,\mathtt{M}\,\mathtt{N}) : \sigma'}$$

$$\frac{\Gamma \vdash \mathtt{M} : \tau \quad \tau \approx \tau'}{\Gamma \vdash \mathtt{M} : \tau'}$$

$$\frac{\Gamma \vdash \mathtt{M} : \tau \quad \Gamma \vdash \mathtt{c} : \tau \rightsquigarrow \tau'}{\Gamma \vdash [\mathtt{c}]\mathtt{M} : \tau'}$$

$$\frac{\Gamma, \pi : s \vdash \mathtt{M} : \omega}{\Gamma \vdash \Lambda\pi : s.\mathtt{M} : s \Rightarrow \omega}$$

$$\frac{\Gamma \vdash \mathtt{M} : s \Rightarrow \omega \quad \Gamma \vdash \mathtt{c} : s}{\Gamma \vdash \mathtt{M}\{\mathtt{c}\} : \omega}$$

$$\frac{\{\pi_1 : s_1, \ldots, \pi_n : s_n\}, \Gamma, x : \tau \vdash \mathtt{M} : \tau \quad C = s_1 \ldots s_n}{\Gamma \vdash (\mathtt{define}\,\mathtt{x}\,\Lambda\pi_1 \ldots \pi_\mathtt{n}.\mathtt{M}) : C \Rightarrow \tau}$$

Figure 1: Inference rules for polymorphic dynamic typing

Note that `tc!` $\circ$ `tc?` = `id` does *not* hold, since standard ("naive") coercion evaluation may generate a run-time error during execution of `tc!`$\circ$`tc?`, whereas evaluating `id` does not. Satisfying this equation would require costly algebraic coercion simplification at run-time. Thus compositions of the former kind are liable to generate *avoidable* run-time type errors. Only equalities of the form `tc?`$\circ$`tc!` = `id` may be assumed to hold.[3] For example, the coercion `boolean?` $\circ$ `procedure!` $\circ$ `procedure?` $\circ$ `boolean!` will generate a run-time error, since a boolean tag is first attached to the argument, followed by a functional check. This check then generates an error. The coercion may consequently be regarded as *unsafe*, since the identity coercion `boolean?` $\circ$ `boolean!` = `id` might as well have been chosen. Hence, we shall always restrict our search for completions to those that are *safe*.

Definition 1 (Safety)
Let `M` and `M'` be two completions of the same pure expression at the same type. We write `M` $\sqsubseteq$ `M'` if it holds, for every context C into which `M` and `M'` can be type-correctly inserted, that `M` generates a run-time type error *only if* `M'` does. In case `M` is a completion at a *monomorphic* type τ, we say that `M` is *safe at* τ if it holds for every other completion `M'` of the same pure term at the same type, that `M` $\sqsubseteq$ `M'`.

An arbitrary completion `M` is said to be *adaptable* to a mono-type τ if there is a coercion c such that `[c]`$\tilde{\texttt{M}}$ has type τ where $\tilde{\texttt{M}}$ arises from `M` by a series of coercion abstractions and instantiations.

We say that a completion `M` at an arbitrary type is *polymorphically safe* if `M` is adaptable to a safe completion at *every* mono-type. □

Note that polymorphic safety is a very strong condition. It is suitable for a situation where we aim at *modularity*, demanding that completions must be produced which work in *all* possible contexts of use. Among the polymorphically safe completions we intuitively seek to *minimize* the amount of run-time type operations and the number of coercion parameters. Certain coercions can be statically determined to be superfluous; in particular we shall orient the equation `tc?` $\circ$ `tc!` = `id` from left to right, viewing the right-hand side as more *efficient* than the left hand side (although both are equally safe.) See [Hen94] for a formal notion of minimality.

To illustrate the rôle played by coercion parameters for both polymorphic safety and minimization, consider as a simple example the program

```
(define M (lambda (x) (if true x (cons 1 2))))
```

This program is a completion of itself at type

$$number * number \rightarrow number * number.$$

However, this completion is *not* polymorphically safe, since `M` will generate a run-time type error when adapted to another type; e.g., $number \rightarrow number$.

[3] In these decisions we follow [Hen92a, Hen94], to which the reader is referred for further information.

To wit, extending `M` to a completion for `C[M]`, where

$$C \equiv (+\ ([]\ \texttt{1})\ \texttt{2}),$$

requires coercions from *number* $*$ *number* to *number* and back:

```
C[(+ ([number?][cons!](M ([cons?][number!]1))) 2)],
```

which will generate a run-time type error. Let us consider the completion `M'` of `M` then:

$$\texttt{M}' \equiv (\texttt{define}\ \texttt{M}\ (\texttt{lambda}\ (\texttt{x})\ (\texttt{if}\ \texttt{true}\ \texttt{x}\ [\texttt{cons!}](\texttt{cons}\ \texttt{1}\ \texttt{2}))))$$

at type scheme

$$\sum[number * number] \rightarrow \sum[number * number]$$

(The sums shown are shorthands for summations in which the pair type occurs at the appropriate place and the other summands are the appropriate type constructors applied to fresh type variables.)

It can be adapted to `C` in such a way that the resulting expression does not generate an error:

```
(+ ([number?](M' [number!]1))  2)
```

This shows that M — our first completion — is not polymorphically safe.

To continue the example, we can easily find contexts where the tagging coercion of `M'` is undesirable; *e.g.*,

$$C' \equiv (\texttt{car}\ ([]\ (\texttt{cons}\ \texttt{3}\ \texttt{4})))$$

which forces the completion of `C'[M']` to be

```
(car ([pair?](M' ([pair!](cons 3 4)))))
```

In the context `C'` it would be better to use `M` instead of `M'` since `(car (M (cons 3 4)))` gives the same result, only without having to tag and untag any data.

Thus it is seen that minimization and safety are aims that must be kept in balance: from the point of view of minimization, `M` may be appealing, but unfortunately it is unsafe. Parameterization over coercions is an important means of ensuring polymorphic safety while still offering the possibility of utilizing context specific information, once the context of use is given.

In our example, using coercion parameters we can reach a compromise, completing `M` as

```
(define M'' (Λπ : number * number ⇝ α.
            (lambda (x)
                    (if #t x ([π](cons 1 2))))))
```

at coercive type scheme

$$\forall\alpha.number * number \rightsquigarrow \alpha \Rightarrow (\alpha \rightarrow \alpha)$$

This completion can then be instantiated with the identity coercion when inserted into `C'`:

```
(car (M''{id} (cons 3 4)))
```

When inserted into `C` we obtain a safe completion by passing `cons` instead of `id`:

```
(+ ([number?](M''{cons!} ([number!] 1))) 2)
```

The price we are paying is that coercions need to be passed at runtime. Clearly it is important to keep the number of coercion parameters to a minimum. In practice most function definitions have polymorphically safe and minimal completions with no or very few coercion parameters.

The cost of the remaining coercion parameters can be minimized by employing *partial evaluation* for complete programs and specializing coercion parameters away before compilation. This appears to be most promising in connection with "nonparametric" implementation technology for polymorphic functions [Tol94, HM95].

3 Translating Scheme to ML

In this section we outline the core of our Scheme-to-ML translation. The basic idea of the core translation is as follows. Tagging coercions are translated to injections into a polymorphic sum-type containing injections corresponding to every type tag of the Scheme type system:

```
datatype ('a1, 'a2, 'a3, 'a4, 'a5) Dynamic =
         T_NIL  |
         T_BOOL of bool |
         T_SYMBOL of string |
         T_CHAR of string |
         T_STRING of string |
         T_NUMBER of int |
         T_VECTOR of `a1 array |
         T_PAIR of `a2 * `a3 |
         T_PROCEDURE of `a4 ->`a5
```

This represents a discriminative sum type constructor in which all type constructors of the source language are present. The type variables `('a1, 'a2, 'a3, 'a4, 'a5)` reflect the number of different type argument positions in the sum type; these can be instantiated independently in different contexts.

Check-and-untag coercions can then be defined according to the schema

```
exception TypeError;

fun U_TAG x =
    case x of
      T_TAG y => y
    | _       => raise TypeError
```

for every primitive tagging coercion `TAG`. Induced coercions are defined by

```
fun id x = x

infix ->;
```

```
fun c->d = fn f => fn x => d(f x)

infix o;
fun c o d = fn x => c(d x)
```

Recursive types require that we use coercions of recursive datatypes. For instance, to translate the combinator

```
(lambda (x) (x x))
```

we translate the type recursion $\alpha = \alpha \rightarrow \beta$ via the declarations

```
datatype ('a, 'b) recty =
         INREC of ('a, 'b) recty -> 'b

fun outrec (INREC x) = x
```

where `outrec` gets the type

```
('a,'b) recty -> ('a,'b) recty -> 'b
```

Unfolding of the recursion equation can then be achieved by coercing with `outrec` as in the ML completion

```
fn x => (outrec x) x
```

at the type

```
('a,'b) recty -> 'b
```

The outline of the translation given here is somewhat oversimplified for the sake of brevity. Section 6 discusses how particular features of Scheme are addressed. Section 7 gives examples of inferred completions translated to Standard ML.

4 A simple example

Before delving into a detailed description of the type inference process let us consider a simple example to get an idea of what the type inference accomplishes and how it does it.

We think of type inference as a problem of solving for unknown coercion variables and type variables. After parsing the input program, every subexpression is annotated with a unique coercion variable having types satisfying the "equational" typing rules. For example, consider the function definition for `f`:

```
(define f
   (lambda (x)
      (lambda (y)
         (if y
             x
             (+ x 1)))))
```

After annotation with coercion variables of suitable types we get the annotated definition[4]

[4]This is simplified in that we assume `+` to be a given *form* with two argument expressions.

```
(define f[c1,c2,c3,c4,c5,c6,c7,c8]
  ([c1] (lambda (x)
    [c2] (lambda (y)
     ([c3] (if ([c4] y)
               ([c5] x)
                ([c6] (+ ([c7] x) ([c8] 1)))))))))
```

where the coercions variables `c1, ..., c8` have the following functionalities:

$$\begin{array}{lcllcl}
\texttt{c1} & : & (\alpha \to \epsilon) \rightsquigarrow \mu & \texttt{c5} & : & \alpha \rightsquigarrow \gamma \\
\texttt{c2} & : & (\beta \to \delta) \rightsquigarrow \epsilon & \texttt{c6} & : & \mathit{number} \rightsquigarrow \gamma \\
\texttt{c3} & : & \gamma \rightsquigarrow \delta & \texttt{c7} & : & \alpha \rightsquigarrow \mathit{number} \\
\texttt{c4} & : & \beta \rightsquigarrow \mathit{bool} & \texttt{c8} & : & \mathit{number} \rightsquigarrow \mathit{number}
\end{array}$$

This gives a type correct completion at type

$$\forall \alpha\beta\gamma\delta\epsilon\mu.(\alpha \to \epsilon) \Rightarrow (\beta \to \delta) \Rightarrow (\gamma \rightsquigarrow \delta) \Rightarrow (\beta \rightsquigarrow \mathit{bool}) \\ \Rightarrow (\alpha \rightsquigarrow \gamma) \Rightarrow (\mathit{number} \rightsquigarrow \mathit{number}) \Rightarrow \mu.$$

Since the number of these coercions is proportional to the size of an expression the number of coercion parameters obtained in this fashion quickly gets out of hand. With polymorphic let-expressions they even grow exponentially.

The purpose of the following analysis is to get rid of as many coercion parameters as possible by instantiating them to coercion constants, without, however, losing (polymorphic) safety *or* minimality of the resulting completion. For example, since `c8` has type signature *number* $\rightsquigarrow$ *number* we can replace `c8` by `id`. We can instantiate β to *bool* and thus replace `c4` by `id` because the boolean test requires a boolean value,[5] and we can check the value of y to be a boolean already when passing it to y. Finally, we can replace `c1`, `c2`, `c3` and `c5` by identity coercions since any tagging or checking at the corresponding subexpressions can be "pushed" into the context; that is, the context is responsible for coercing the arguments to `f` and its result to suit its requirements. (This may require applying induced coercions to `f`.)

The only remaining coercion variables are `c6`: *number* $\to \alpha$ and `c7`: $\alpha \to$ *number*, yielding the completion

```
(define f[c6,c7]
    (lambda (x)
       (lambda (y)
      (if y
          x
          ([c6] (+ ([c7] x) 1))))))
```

at type

$$\forall \alpha(\mathit{number} \rightsquigarrow \alpha) \Rightarrow (\alpha \rightsquigarrow \mathit{number}) \Rightarrow \alpha \to \mathit{bool} \to \alpha.$$

These last two coercions, however, we do not remove. It is tempting, but unsafe to set α to *number* and thus replace both `c6` and `c7` by the identity coercion. This is tantamount to insisting that x only be passed numbers. This

[5]This is *not* the case in ordinary Scheme! There *any* value is acceptable in in the test part of an if-expression.

is unsafe since an application of `f` to *any* value v for `x` succeeds and returns v if `y` is passed `#t`. We could also set α to `(...) Dynamic` and replace `c6` by `number!` and `c7` by `number?`. This gives a safe completion and eliminates both coercion parameters. Yet it commits the function to execute these type coercions *every* time the function body is executed, even when `x` is passed a number.

In a complete program context the coercion parameters can usually be removed by partial evaluation. This eliminates the need for passing coercions at run-time. At the same time the specialized versions have only a minimum of run-time type operations.

5 Solving coercion constraints

We describe the phases by which the type inference process analyzes a program. Several of these phases can be merged for practical efficiency.

5.1 Parsing and attribution

In the first stage the input is parsed into an attributed abstract syntax tree. Every subexpression `e` has a new coercion variable `c` whose type signature is initialized to $\alpha \rightsquigarrow \beta$, where α and β are unique. We say β is the *high type* of `c` (and, by extension, of `e`) and α its *low type*. The coercion variables range over *primitive* type coercions, including the operation that unconditionally generates a (run-time!) type error.

Lambda-bound variables have a single unique type variable as their attribute and let-bound variables get a (possibly coercion parameterized) type scheme, which is initialized to some (irrelevant) value.

The type variables may be be unified with other types or type variables in the later phases. Indeed, the unifications taking place constitute the inference process, because the coercions are determined uniquely by their type signatures.

5.2 Unification of types according to type rules

In the second stage the types in the coercion signatures are unified in accordance with the static typing rules. For example, if α is the high type of `e`, β the high type of `e'` and γ the low type of `(e e')`, then α is unified with the function type $\beta \rightarrow \gamma$. Thus after this step the coercion variable `c` attributing `e` has signature $\delta \rightsquigarrow (\beta \rightarrow \gamma)$, where δ is the low type of `e`. This gives some information about `c` but does not identify it. Knowing that `c` must be a primitive coercion there are four possibilities for what coercion `c` could be in the end:

1. if $\delta = (\beta \rightarrow \gamma)$ then it is the identity ("no-op") coercion `id`;
2. if $\delta = tc(\ldots)$ for any type constructor `tc` other than $\rightarrow$ then `c` is the *error coercion*, *i.e.* the coercion that generates a type error whenever applied to anything at all;
3. if $\delta = (\ldots)$ `Dynamic` then `c` is the tag checking coercion `U_PROCEDURE`;

4. finally, if δ is a type parameter (that is, a bound type variable within a type scheme), then `c` is a coercion parameter.

The result of the first stage is a completion that satisfies the type rules of Section 2. The following phases eliminate most of the coercion variables by "solving" for the type variables occurring in the coercion signatures and thus setting them either to the identity coercion or a specific primitive type coercion.

5.3 Construction of simple value flow graph

The type signatures of all the coercion variables in an attributed expression constitute its *coercion constraints*. They can be viewed as the edges of a *value flow graph*, where the types occurring in the constraints are the nodes of the graph. Knowing that constraints must be solved to be signatures of *primitive* coercions we can deduce that certain types must be equal. Let us consider a couple of typical examples.

- Consider a constraint of the form

  $$tc(\alpha_1, \ldots, \alpha_n) \rightsquigarrow tc(\beta_1, \ldots, \beta_n).$$

  There is only one primitive coercion with a signature that matches this constraint: the identity coercion. Thus we can unify $\alpha_1 = \beta_1, \ldots, \alpha_n = \beta_n$.

- Consider the two constraints

  $$tc(\alpha_1, \ldots, \alpha_n) \rightsquigarrow \gamma$$

  and

  $$tc(\beta_1, \ldots, \beta_n) \rightsquigarrow \gamma$$

  from two coercion variables `c1,c2`. Since they both coerce to the same type, `c1` and `c2` must be *equal*. Thus we can unify $\alpha_1 = \beta_1, \ldots, \alpha_n = \beta_n$.

- Consider the two constraints

  $$tc(\alpha_1, \ldots, \alpha_n) \rightsquigarrow \gamma$$

  and

  $$\gamma \rightsquigarrow tc(\beta_1, \ldots, \beta_n).$$

  Even though we cannot identify which particular coercions these constraints correspond to we can conclude that the component types must be equal. Thus we unify, as in the previous case, $\alpha_1 = \beta_1, \ldots, \alpha_n = \beta_n$.

Generalizing all these cases, let $\tau_1 = tc(\alpha_1, \ldots, \alpha_n)$ and $\tau_2 = tc(\beta_1, \ldots, \beta_n)$ be arbitrary nodes (types) in the constraint graph such that τ_1 and τ_2 can reach each other by following constraint graph edges *in either backwards or forwards direction*, then it must be the case that $\alpha_1 = \beta_1, \ldots, \alpha_n = \beta_n$. We call this the *simple value flow closure condition.*

The simple value flow graph (SVFG) induced by a set of constraints is the largest graph that satisfies the simple value flow closure condition and can be derived from the constraints, viewed as a graph, by contracting nodes (unifying types). Constructing the simple value flow graph is what we do in this, the third stage.

To get a more intuitive understanding of the SVFG we can think of the low type of a (sub)expression as its abstract value. (Several expressions may have the same abstract value.) The constraints (coercion signatures) are then edges that show how values "flow" through the program. If the low type of a constraint (the head of a flow edge) is of the form $tc(\alpha_1, \dots, \alpha_n)$, it indicates a subexpression where a value (such as a function, pair, number, etc.) is *constructed*. If the high type of a constraint (tail of a flow edge) is of the form `tc('a1,...,'ak)`, this indicates a subexpression where a value is *destructed*; such as when it is applied as a function, or is the argument of a projection operation for pairs. The simple value flow graph connects possible producers with consumers of values. It is called *simple* because it does not preserve flow directionality for the components of structured or higher-order types.

The SVFG is computed very efficiently, in almost-linear time relative to the size of the original graph, using an "instrumented" unification algorithm based on the union-find data structure with path compression and ranked union [Hen92c].

5.4 Cycle elimination

The simple value flow graph of an expression may contain cycles; that is, a set of type variable nodes each of which reaching and being reachable from any other node in the set. In the following (fifth) stage we shall set the value of a type variable based on the *sources* reaching it and the *sinks* it reaches in the SVFG. Since this information is identical for all type variables on a cycle, we collapse all variables in a cycle by unifying them with each other in this, the fourth stage. This is done using an efficient maximal strong components algorithm.

5.5 Sources and sinks of type variables

In the fifth stage we determine the types of type variables using information on all its *sources* and *sinks* in the acyclic SVFG. (A source is any node with no incoming flow edges. A sink is any node with no outgoing flow edges.)

The set $src(\alpha)$ for a node α are all those sources that reach α, and $snk(\alpha)$ contains all those sinks that can be reached from α.

The key insight in the following considerations is that $src(\alpha)$ describe the types of all values that can possibly reach α. Similarly, $snk(\alpha)$ describes the expected types of all the possible uses of the values of α.

Combining this information, we determine the type of a type variable α by checking which of the following cases applies first:

1. If $src(\alpha)$ contains only nonvariable types, each of them having the same type constructor, then this indicates that only values of that particular type can reach α. Thus we unify α with $src(\alpha)$.

2. If $\mathit{snk}(\alpha)$ contains only nonvariable types, each of them having the same type constructor, then this indicates that every value reaching α will be used — if at all — by operations for that particular type. Thus we unify α with $\mathit{snk}(\alpha)$. This may result in an "early" type check operation for a type that is performed long before an operation requiring an argument of that particular type is executed.

   This appears plausible — if the only use of a value after some program point is as a number, why not check that it is a number right away? — but it is not entirely safe in that the use may not always actually be executed. If an absolutely faithful translation preserving the original semantics is required we can use exclusively $\mathit{src}(\alpha)$, but not $\mathit{snk}(\alpha)$, to make a determination for α. Alternatively, it should be possible to devise a "is-definitely-used" analysis to verify when our rule can safely be applied.

3. If $\mathit{src}(\alpha)$ contains at most one nonvariable type (say $\mathit{tc}(\alpha_1, \ldots, \alpha_n)$ and at least one variable type then we have a situation where it *could* be that only values of type $\mathit{tc}(\alpha_1, \ldots, \alpha_n)$ reach α — or not — depending upon the values of the type variables(s). In this case we make α a type parameter and unify it with all the variable types in $\mathit{src}(\alpha)$ and $\mathit{snk}(\alpha)$.

   Analogously for $\mathit{snk}(\alpha)$

4. If both $\mathit{src}(\alpha)$ and $\mathit{snk}(\alpha)$ contain at least two types with distinct type constructors each, this signals that values of different types can reach the expression(s) `e` with low type α and that `e` may be used in operations requiring different types. This is a *plausible* situation [Tha90], as all the operations may work without type error at run-time. In this case the type tagging and checking will have to be done at run-time: we unify α with `(...) Dynamic` where all the nonvariable types of $\mathit{src}(\alpha)$ and $\mathit{snk}(\alpha)$ enter as summands.

Since $\mathit{snk}(\alpha)$ should give *all* the possible uses of α we have to adjust our notion of sinks. This is to make sure we detect the case where a program variable is used in the then-branch of a conditional, but not in the else-branch, or vice versa. We require of every program that, as in relevance logic, every bound variable has at least one applied occurrence and that both branches of a conditional contain the same set of free variables. This is accomplished by including explicit operations for "discarding" the value of a variable (as in linear logic) in a preprocessing step.

5.6 Coercion identification

In the previous stages we have solved for the type variables in the attributed syntax tree. In the sixth stage we determine the coercions by looking at their type signatures. For a coercion variable `c` with type signature $\tau_1 \leadsto \tau_2$ there are the following possibilities:

1. τ_1 and τ_2 are equal: in this case `c` is `id`;

2. τ_1 = `(...) Dynamic` and τ_2 = `tc(...)`: in this case `c` is the check operation for type constructor `tc`.

3. $\tau_1 = tc(\ldots)$ and τ_2 = `(...) Dynamic`: in this case `c` is the tag operation for type constructor `tc`.

4. $\tau_1 = tc(...)$ and τ_2 is a type parameter, or vice versa: in this case `c` is a coercion parameter.

5.7 Forming polymorphic coercive types

In the seventh stage we determine the types of let- and define-bound variables `x`: all type and coercion parameters occurring in the expression `x` is bound to are collected and abstracted over (generalized). To ensure that this is correct we check that

- the set of coercion parameters is empty, or
- e is a syntactic value, such as `(lambda (x) e)`.

If none of these two conditions applies, then we instantiate all coercion parameters to either tagging or checking coercions. This is to make sure that the call-by-value semantics of Scheme is preserved. Since top-level definitions must not contain imperative type variables we "monomorphize" all imperative type variables still occurring in the type by unifying them with type Dyn. Imperative type variables arise in connection with side-effecting operations such as `set!`, `set-car!`, `set-cdr!` and `call/cc`. The monomorphic universal type Dyn is definable as

```
datatype Dyn = REC of (Dyn,Dyn,Dyn,Dyn,Dyn) Dynamic.
```

5.8 Pretty-printing/translation

In the eighth and final stage the results of the analysis are pretty-printed as an annotated Scheme program (for static debugging purposes) and/or translated to Standard ML (for translation purposes). Translation to Standard ML is done by generating ML Kit abstract syntax from the coercion annotated Scheme abstract syntax.

6 Dealing with Scheme specifics

In this section we describe how we address the specific problems raised by translating full IEEE Scheme to ML.

6.1 Dynamic top-level bindings

Scheme employs dynamic top-level binding; that is, in a top-level procedure definition, the free variables are *not* resolved statically against the bindings that hold in the environment of the definition. Instead, the free variables are bound in the dynamic environment of the particular call to the function. In the example

```
(define append (lambda (l1 l2)
    (if (null? l1)
        l2
        (cons (car l1) (append (cdr l1) l2)))))
```

in Section 4 we assumed fixed types for `null?, cons, car, cdr` and treated the call to `append` as a recursive function call. This is indeed useful for static debugging purposes, where we pose the question: "What is the type of append, if we assume the standard bindings for the free variables?" For the purpose of translating this definition faithfully to ML, which has static top-level binding, we cannot simply assume that these variables have their standard bindings since they may be rebound dynamically. To account for this possibility we can abstract over all individual occurrences of the free variables in the body of a definition (this includes `append` in the example above!), in order to obtain a combinator; that is, we treat the definition of `append` above as the combinator definition

```
(define append (lambda (null?1 cons1 car1 append1 cdr1)
     (lambda (l1 l2)
       (if (null? l1)
           l2
           (cons (car l1) (append1 (cdr l1) l2))))
```

If a body contains several occurrences of a free variable then *each* is abstracted separately.

This is translated into the ML function

```
fun append (null1, ..., cdr1) (l1,l2) = ...
```

which takes two structured arguments, the first being the relevant part of the environment within which append is to be evaluated, the second being the actual arguments to append.

Top-level dynamic binding is useful mostly as a method for accomplishing *program editing* in an interactive, sequential top-level loop during program development. It is of rather questionable value in a finished system since, in effect, *all* identifiers, including the ones bound to standard procedures and those used at the top level in the construction of the system, can be rebound to completely different values *at any time* during which the system runs. Typically this is avoided, indeed prohibited, by providing a (nonstandard) mechanism for "freezing" the environment and packaging it up with a complete program. Thus a translation from Scheme programs to ML is probably best served by performing it under the assumption of *static* top-level binding. In this case none of the above parameterization over free variable occurrences in definitions is necessary and the translation will look considerably more natural.

6.2 Side effects and call/cc

The arguments (or rather: their low types) that are side effected in Scheme using `set!` or the standard functions for `set-car!, set-cdr!` are marked as side-effected. In the translation these types will become reference types ("box" types in Scheme/LISP lingo) with imperative type variables. Similarly, `call/cc` obtains Standard ML type `(('_a -> '_b) -> '_a) -> '_a` with imperative type variables `'_a, '_b`. Imperativity is preserved during type unification. Free imperative type variables in top-level definitions are avoided by instantiating them to type `Dyn`.

6.3 Boolean tests and exceptional values

In IEEE Scheme the value `#f` functions both as "false" in if-expressions and as exception value in lookup-operations such as assoc. Furthermore, *all* values different from `#f` function as "true" in an if-expression. The typing rules for if-expressions and lookup functions in Scheme have type `(...) Dynamic` in in the ML translation. This implies that, unfortunately, tagging and checking "propagate" in the neighborhood of if-expressions and applications of lookup functions for translation purposes only, even though their use is not attributable to possible problems with type safety.

6.4 Lists in Scheme and ML

Lists play a special role both in Scheme and ML. Scheme has a plethora of standard procedures for processing lists, and procedure arguments are passed as lists. Similarly, ML has a standard type for lists and lots of operations on them.

In the translation described above, lists become elements of type `(...) Dynamic` instead of being translated to (ML) lists. In order to obtain lists as the result of the translation we add another summand to `(...) Dynamic`:

```
datatype ('a1, 'a2, 'a3, 'a4, 'a5, 'a6) Dynamic =
        ...
       | T_LIST of 'a6 list
```

This provides the possibility of ambiguous representation of tagged values; for example, tagged `'()` is either represented by `T_NIL` or `T_LIST nil`. Correspondingly we change the check-and-untag coercions `U_NIL` and `U_PAIR` such that they do *not* automatically fail when applied to a value tagged with `T_LIST`.

We can now treat `'a list` as a "little" recursive sum type with type constructors `NIL` and `PAIR`. In the fifth stage of type inference (see Section 5.5) we recognize the special case when only `NIL` and `PAIR` are the type constructors from which a dynamic sum type would normally be built. In this case we map the type variable in question to a list type. Should it be necessary to coerce from a list type to a general dynamic type then we apply the `T_LIST` constructor to the list.

6.5 Type testing routines

Type testing predicates are essentially the "checking part" of our check-and-untag coercions. Any object can be safely tested for its Scheme type. For the purpose of translation, type testing routines have static type

$$('\mathtt{a},'\mathtt{b},'\mathtt{c},'\mathtt{d},'\mathtt{e})\mathtt{Dynamic}- > \mathtt{bool}$$

This supports an implementation of such predicates via type tags. For instance, we can translate the Scheme predicate `boolean?` as

```
fun is_boolean x =
    case x of T_BOOL _ => true | _ => false
```

Our type system does not model control flow information. Type testing predicates aggravate the loss of static type information since they are typically used to steer the control flow in a program in such a fashion that execution depends on which type tag an object has at run-time.

6.6 I/O routines and equality predicates

I/O routines and equality predicates (as well as classical garbage collection) are major causes of inefficiency in the implementation of polymorphically typed languages since they require extensive tagging. (This is also the reason why, hitherto, ML-like languages have been implemented by, in essence, translating them to Scheme, not the other way round, as we are describing here.)

In our translation I/O routines and equality predicates expect their inputs be of the monomorphic type Dyn. This is required in generality since Standard ML does not support nonparametric polymorphism. In that case we could have given the equality predicates equal?, eqv? and eq? the type scheme $\forall\alpha\beta.\alpha \times \beta \rightarrow \mathit{bool}$, and, similarly, for I/O routines.

6.7 Translation vs. static debugging

We have mentioned that our system is intended to support the double purpose of translation to ML and static debugging. We note, however, that those two enterprises may not always share common interests. From the point of view of static debugging, it is natural to adopt the principle that the inferred run-time type operations should satisfy *only* requirements of *run-time type safety.* On the other hand, viewing the process of completion as the backbone of a translation may lead to decisions of language *implementation* which violate the principle of inference for pure static debugging purposes just mentioned. An example of this is the implementation of Scheme's type testing predicates. As indicated earlier, we implement these predicates, in the translation, via tagging coercions. However, since such predicates are universally defined, they cannot possibly give rise to run-time type safety problems, and hence the "implementation driven" introduction of tags (and, possibly, concomitant checks elsewhere in the program) may be considered undesirable from a pure static debugging perspective. Under this view, we may rather prefer to use the (non-parametric polymorphic) typing of predicates P;

$$\mathtt{P} : \forall\alpha.\alpha \rightarrow \mathit{bool}$$

Here we abstract from the problem of how the predicates are *implemented*; they might be implemented via techniques of run-time type parameter passing (as have been discussed in several contexts recently, *e.g.*, [Tol94, HM95]), or they might in fact even be implemented via some form of tagging, and still this wouldn't show up in the completions of a "pure" system of static debugging.

7 Examples

In this section we consider SML-completions produced for some example definitions. We first show two very simple definitions, illustrating the use of coercion parameters.

```
(define f1 (lambda (x) (if #t x #f)))
(define f2 (lambda (x) (if #t x (car 1))))
```

The output of SML-completion inference is:

```
val rec f1 =
    fn CV8 => let val rec f1 =
                  fn (x, []) =>
                     if true then x else (CV8 false)
              in f1 end;
val rec f2 = fn (x, []) =>
      if true then x
      else (car (((fn x => raise TypeError) 1), []));
```

at the types

```
val f1 = fn : (bool -> 'a) -> 'a * 'b list -> 'a
val f2 = fn : 'a * 'b list -> 'a
```

respectively.

First we note a few things about the translation of non-core facilities used in our examples here. Coercion parameters are denoted `CVn` where n is a numerical index. Also, note that parameter lists of Scheme procedures are translated via right-associated pairs, ending in `[]`. This allows for simulation of Scheme's parameter list-pattern matching. Finally, note the use of the *error-coercion*

```
fn x => raise TypeError
```

which is used to translate error-generating coercions, such as, *e.g.*

```
[cons?][number!]1
```

The main point about the example above is that, in the case of `f1`, the analysis discovers that the type of x cannot safely be identified with *bool*, since x is not guaranteed to be consumed at that type. This forces the parameter application in the second branch of the conditional. However, in the case of `f2`, this consideration is rendered superfluous by the run-time type error at the application of `car`, which results in the absence of any type constraints on the result of that application.

Our next example shows the workings of primitive coercions. Consider the little program

```
(define zip (lambda (p1 p2)
            (cons (cons (car p1) (car p2))
                  (cons (cdr p1) (cdr p2)))))
(define g (lambda (x)
     (if x (zip x '(1 . 1)) (zip x '(#f . #f)))))
```

The inferred SML-completion is:

```
val rec zip =
fn (p1, (p2, [])) =>
 (cons ((cons ((car (p1, [])), ((car (p2, [])), []))),
       ((cons ((cdr (p1, [])), ((cdr (p2, [])), []))),
       [])));
val rec g =
fn (x, []) =>
   if ((check_BOOL) x)
   then (zip (((check_PAIR) x),
          ((((in_INT) 1), ((in_INT) 1)), [])))
   else (zip (((check_PAIR) x),
          ((((in_BOOL) false), ((in_BOOL) false)), [])));
```

at the types

```
val zip = fn : ('a * 'b) * (('c * 'd) * 'e list) ->
               ('a * 'c) * ('b * 'd)
val g = fn : ('a,'b,'c,'d) dyn * 'e list ->
      ('a * ('f,'g,'h,'i) dyn) * ('b * ('j,'k,'l,'m) dyn)
```

While the function `zip` is completed with no coercions at all, its use in function `g` leads to a spread of dynamic type coercions. There are two sources of this. First, the type of the parameter x must be coerced to a dynamic sum, because the conditional is assumed here to consume a boolean, and `zip` consumes pairs. Second, the two branches of the conditional construct pairs of elements of different types (integers and booleans), which again leads to the need for a dynamic sum type for the components of the pairs. The function `g` was purposely invented to illustrate the use of the dynamic sum; many programs are completed with substantially fewer coercions, of course.

We give a few examples to show, among other things, optimized translation of list types. In general, this is achieved by using "aggressive" typings of predicates together with representation shift coercions between lists and pairs. Using an aggressive non-parametric typing of the predicate `null?`, the `map` function

```
(define map (lambda (f l)
  (if (null? l) '() (cons (f (car l)) (map f (cdr l))))))
```

gets completed without any parameters, but with inferred representation shift coercions for lists and pairs,

```
val PAIR2LST = fn (x, y) => x::y
val LST2PAIR = fn (x::y) => (x, y) | _ => raise EmptyList
```

as follows:

```
val rec map =
fn (f, (l, [])) =>
   if (isnull (l, [])) then []
   else (PAIR2LST (cons
          ((f ((car (LST2PAIR l, [])), [])),
          ((map (f, ((cdr (LST2PAIR l, [])),
```

The type of the completed `map` is

```
('a * 'b list -> 'c) * ('a list * 'd list) -> 'c list
```

assuming the aggressive (non-dynamic) typing of the predicate `null?`, as

$$\texttt{null?} : \forall\alpha.\alpha\ \texttt{list} \rightarrow \mathit{bool}$$

This typing is also assumed in the next example.

The next example is the `append`-function, of two arguments. This function is peculiar in that it does not destruct its second argument, which can be anything; it consumes a list in the first argument. Since nothing can be assumed about the type of the second argument, the object produced by `append` must be coerced to the type of that argument; since this is unknown, the coercion must be parameterized, as is shown below:

```
val rec append =
fn CV10 =>
 let val rec append =
          fn (l1, (l2, [])) =>
             if (isnull (l1, [])) then l2
             else (CV10 (cons
        ((car (LST2PAIR l1, [])),
         ((append ((cdr (LST2PAIR l1, [])),
                  (l2, []))), []))))
 in append end;
```

The tautology checker

```
(define taut (lambda (prop)
    (if (equal? prop #t)
        #t
        (if (equal? prop #f)
            #f
           (if (taut (prop #t))
             (taut (prop #f))
               #f)))))
```

(compare [Fag90, WC94]) can be completed as shown below, under the assumption that `equal?` has type

$$\texttt{equal?} : \forall\alpha.\alpha * \alpha \rightarrow \mathit{bool}$$

```
val rec taut =
fn (CV1, CV2) =>
  let val rec taut = fn (prop, []) =>
       if (isequal (prop, ((CV1 true), [])))
       then true
       else
        if (isequal (prop, ((CV1 false), [])))
           then false
           else
            if (taut (((CV2 prop) (true, [])), []))
               then (taut (((CV2 prop) (false, [])), []))
               else false
  in taut end;
```

at the type

```
(bool -> ''a) * (''a -> bool * 'b list -> ''a) ->
''a * 'c list -> bool
```

A more admissive non-parametric polymorphic typing could be considered, namely

$$\texttt{equal?} : \forall\alpha\beta.\alpha * \beta \rightarrow \mathit{bool}$$

Using this typing would lead to elimination of the first coercion parameter in the completion shown above.

8 Related work

The literature on type analysis of dynamically typed programming languages is vast. Most of it employs intraprocedural data flow analysis to optimize the implementation of dynamically typed languages, such as by eliminating run-time type tests or resolving dynamic overloading or method dispatching. Some of it, notably the work of Shivers [Shi91b, Shi91a] employs (interprocedural, semantics-based) abstract interpretation for this purpose. As these works are aimed at the implementation level they are not based on *type systems* understood as formal tools for reasoning about the *source* programs. As such they neither give a clear specification (such as a type system) to a programmer of *what* they are doing, nor do they give much feedback to a user trying to debug his or her program.

Our approach is based on a formal type system for a dynamically typed language. As such it is most closely related in its goals to two recent systems of *soft typing*: Cartwright, Fagan and Wright at Rice University were the first to expound the virtues of what they termed *soft typing*. They have developed soft typing systems for Scheme based on Remy-encoded subtyping [Fag90, CF91, Wri94, WC94]. Aiken, Wimmers and Lakshman [AWL94] (see also [AW93]) have developed an alternative soft type system based on set constraint solution methods. The former system comprises Remy-encoded discriminative sum-types, recursive types and Hindley-Milner style polymorphism; the type language of the latter system is still more powerful, including all the set-theoretic operations of union, intersection, negation and subtyping together with so-called conditional types, which can express control-flow dependencies.

The present work differs from both these lines of work in several respects. Three major points are:

1. Our system models both tagging and check/untagging operations and deals with optimization of both kinds of run-time type operations as dual aspects, in the style of Henglein's *dynamic typing* [Hen92a, Hen92b, Hen94]. This is in distinction to the other systems which assume that all objects are tagged at run-time, and they only deal with optimization of run-time checking operations.

2. Our Scheme-to-ML translation, based on the integrated soft- and dynamic typing-style completion inference is new. Our translation is based in an essential way on the explicit modeling of both of the dual run-time type

operations. Furthermore, the translation can be used to produce type-specific efficient data representations.

3. The present work addresses the problem of inferring *polymorphically safe* completions, suitable for *modular* or *incremental* completion inference, via the use of *coercion parameters.* This problem appears not to have been studied before [6] Previous work on soft typing has relied on *global* analyses, where it is assumed that an entire program is given for analysis.

Of the systems mentioned, the soft type inference for Scheme of Cartwright, Fagan and Wright is closest in spirit to ours. Whereas their run-time checks are limited to be situated at primitive operations (including procedure application), our system has no such restriction, since type coercions can in principle be placed anywhere in a program, and in our present system their occurrence is in fact not limited to primitive application points. This flexibility is important in a system which addresses the modularity issue, since it helps decrease the number of run-time type operations required for a universally adaptable completion. Consider as a simple example the application function `apply`, given by

```
(define apply (lambda (f x) (f x)))
```

Our system infers the type $\forall\alpha\beta.((\alpha \to \beta) * \alpha) \to \beta$ and inserts no coercions at all into the function, deferring all run-time type operations to the context of its use. This is only possible because coercions can be placed at arbitrary points, such as, *e.g.*, argument positions. For example, `apply` can be inserted into the context $C \equiv$ ([] `true` `1`), yielding the completion

```
(apply [F?][B!]true \:1)
```

of the expression C[`apply`], whereas insertion of `apply` into $C' \equiv$ `([] (lambda (y) y) 1)` gives back just C'[`apply`] itself, using no coercions. In contrast, in a system in which run-time type checking operations adhere to primitive application points, one is faced with the choice of either having or not having a checked version of the application operation at `(f x)`. Having it is necessary in a universally applicable completion of `apply`, because a context such as C would require it, but its position inside the function is unfortunate for contexts such as C' which do not require the check.

9 Conclusion and future work

We have presented a type inference system for Scheme that is both liberal in that it does not outright reject any Scheme programs, and it is "aggressive" in that it infers safe polymorphic completions, yet seeks to minimize the number of primitive coercions and coercion parameters (preferring to minimize primitive coercions over coercion parameters) and finds a "clever" placement for coercions in the source program. The results of the analysis can be used both for static debugging of the source program and its translation to polymorphic languages in the ML family, such as Standard ML and CAML. We have developed a system

[6] In [Wri94] [WC94] the issue is mentioned as an important part of future work for their softly typed version of Scheme; see also below for further comparison.

that performs polymorphic type inference for the kernel of IEEE Scheme, and we are in the process of extending it to a substantial subset of IEEE Scheme. Part of this is a translator to Standard ML of New Jersey (SML/NJ). The reason for picking Standard ML of New Jersey is that the ML Kit is currently built on top of SML/NJ and that SML/NJ supports call/cc. Realistic alternatives to SML/NJ are CAML Light and Moscow ML.

The Scheme translator is intended as a component of the ML Kit, a Standard ML implementation kit developed at DIKU and the University of Edinburgh. We envisage that the new implementation techniques under development for Standard ML (region-based memory management and representation optimization of structured data) can be utilized productively also for dynamically typed languages such as Scheme. Obtaining truly efficient code is likely to require a low-level translation into a suitable intermediate language, however.

To support static debugging of Scheme new type error finding and tracing algorithms will have to be developed. For example, type error coercions passed to coercion parameterized functions have to be traced to the place in the body of the function definition that contains the corresponding coercion parameter in applied position.

References

[AW93] Alex Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), Copenhagen, Denmark*, pages 31–41. ACM Press, 1993.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proc. 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon.* ACM, ACM Press, Jan. 1994.

[CC91] F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation, Toronto, Ontario*, pages 278–292. ACM, ACM Press, June 1991.

[CR91] W. Clinger and J. Rees. Revised^4 report on the algorithmic language scheme. *ACM Lisp Pointers*, IV, July-September 1991.

[Fag90] Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages.* PhD thesis, Rice University, 1990.

[Hen92a] F. Henglein. Dynamic typing. In *Proc. European Symp. on Programming (ESOP), Rennes, France*, pages 233–253. Springer, Feb. 1992. Lecture Notes in Computer Science, Vol. 582.

[Hen92b] F. Henglein. Global tagging optimization by type inference. In *Proc. LISP and Functional Programming (LFP), San Francisco, California*, June 1992.

[Hen92c] Fritz Henglein. Simple closure analysis. DIKU Semantics Report D-193, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, March 1992.

[Hen94] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming (SCP)*, 22(3):197–230, 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages (POPL), San Francisco, California*, Jan. 1995.

[Jon92] Mark P. Jones. A theory of qualified types. In *Proc. European Symposium on Programming (ESOP), Rennes, France.* Springer-Verlag, 1992. Lecture Notes in Computer Science, Vol. 582.

[Jon94] Mark P. Jones. ML typing, explicit polymorphism, and qualified types. In *Proc. Conf. on Theoretical Aspects of Computer Science (TACS), Sendai, Japan*, pages 56–75. Springer-Verlag, 1994. Lecture Notes in Computer Science, Vol. 789.

[Reh95] Jakob Rehof. Polymorphic dynamic typing — aspects of proof theory and inference. Master's thesis, DIKU, University of Copenhagen, March 1995.

[Shi91a] O. Shivers. Data-flow analysis and type recovery in Scheme. In P. Lee, editor, *Topics in Advanced Language Implementation*, chapter 3, pages 47–88. MIT Press, 1991.

[Shi91b] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages* or Taming Lambda. PhD thesis, Carnegie Mellon University, May 1991.

[Tha90] S. Thatte. Quasi-static typing. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 367–381. ACM, Jan. 1990.

[Tol94] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. ACM SIGLPAN Symp. on LISP and Functional Programming (LFP), Orlando, Florida*, June 1994.

[WC94] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. In *Proc. ACM Symp. on LISP and Functional Programming (LFP), Orlando, Florida*, 1994.

[Wri94] Andrew Wright. *Practical Soft Typing.* PhD thesis, Rice University, 1994.