Towards Partial Evaluation of Full Scheme

Peter Thiemann*

Abstract

We present a binding-time analysis for Scheme which enables an offline partial evaluator to successfully treat Scheme's reflective features eval, apply, and the control operator call/cc. Additionally, our analysis empowers the specializer to select the most efficient representation for each object. This removes some limitations of previous specializers regarding the use of higher-order functions. The theoretical development is backed by an implementation.

Keywords: partial evaluation of reflective language features, meta-computation

The programming language Scheme [20] is an ideal vehicle for meta-computation, specifically for partial evaluation. Part of the appropriateness of Scheme for meta-computation tasks derives from its reflective features: eval [27], which reflects the external representation of, say, a procedure to a functional value, apply, which reflects lists to argument lists, and call/cc[1], which makes the continuation of the current expression available as a procedure. The continuation can be viewed as a reified object from the meta-interpreter (supposedly written in CPS).

Much work in partial evaluation has been done in the context of Scheme [3, 6, 7, 5, 8]. However, its reflective features have been neglected so far. In order to overcome this drawback, we present a binding-time analysis and a specialization algorithm for a large functional subset of Scheme. The subset includes eval, apply, and call/cc. Additionally, our new analysis supports variadic functions and includes a representation analysis which treats external and built-in operators more liberal than previous analyses. The latter analysis improves the efficiency of the specializer by enabling it to choose most efficient representations for functions and other partially static data. To the best of our knowledge, these features are unique to our work.

Overview In the following section we briefly review offline partial evaluation. We assume some familiarity with partial evaluation [21] and with type-based binding-time analysis (*e.g.*, [15, 16, 18, 7]). In Sec. 2, we present two applications—parser generation and exception handling—which benefit from a partial evaluator capable of dealing with eval, apply, and call/cc. Section 3 informally explains the specializer's treatment of these reflective operators. In Sec. 4 we present the formal development of our binding-time analysis in a type-based framework. We briefly discuss automatic type reconstruction for the underlying type system in Sec. 5 outline the specialization algorithm in Sec. 6. Finally, Section 7 considers related work.

1 Partial Evaluation

Partial evaluation is a program specialization technique which is based on aggressive constant propagation. The aim of specialization is improving efficiency: Given a *source program* and the *static* (known) part of its input, construct a *residual program* which accepts the remaining, *dynamic* part of the input and computes the same answer as the source program applied to the entire input, but more efficiently.

We deal with the offline variant of partial evaluation where—in a first pass—a binding-time analysis annotates each expression with its binding time: either "executable" (*static*) or "suspended" (*dynamic*). The second pass, the specializer, simply follows the annotations, reducing the static expressions and rebuilding the dynamic ones. In order to ensure termination of partial evaluation, the specializer *memoizes* certain functions (called *sp-functions*). Whenever the specializer calls an sp-function it specializes its body with respect to the current arguments and generates a function call. In addition, the static part of the current arguments is *memoized*, so that the specialized function is reused the next time the same sp-function is called with arguments that have an identical static part. Specialization is polyvariant: If the sp-function is called with arguments differing in the static part the, specializer generates a new specialized function.

2 Application

We present two applications—parser generation and non-local exits—that can be elegantly expressed using eval and call/cc, respectively. Being able to handle them is advantageous, for both conceptual and efficiency reasons. However, previous offline specializers do not address these features.

2.1 Parser Generation

Recently, we have investigated the generation of efficient LR parsers by partial evaluation [29]. In a continuation of that work we have built a realistic parser generator. The partial evaluator specializes an interpretive LR parser with respect to an attribute grammar and the number of lookahead characters. In an attribute grammar, each context-free production is augmented with an attribution. An (only-S) attribution is simply a piece of Scheme code that maps the attribute values of the right side nonterminals of a production to the attribute value of the left side nonterminal. In order to perform attribute evaluation during parsing, the attribution of each production must be transferred from the

*Address: Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany. E-mail: thiemann@informatik.uni-tuebingen.de

[1] Scheme calls it call-with-current-continuation.

interpretive parser to the generated parser. The interpretive parser can employ a Scheme interpreter interp to evaluate the attributions:

```
(interp (cons (attribution->definition attribution)
              program)
        (take rhs-length attribute-stack))
```

Unfortunately, this approach requires writing an interpreter. Furthermore, parser generation becomes grossly inefficient as the specializer must specialize/interpret the interpreter which actually runs the code.

A much better choice consists in using eval to transform the text of the attribution into the actual function:

```
(apply (eval attribution (interaction-environment))
       (take rhs-length attribute-stack))
```

Since `attribution` will be static, we would like to remove eval from the residual code and simply paste in the static value of `attribution`, considered as code. For example, for the attribution `(lambda ($1 $2 $3) (* $1 $3))` we our specializer generates (real output of the specializer):

```
(let ((var-9031 ((lambda ($1 $2 $3) (* $1 $3))
                 (car var-9030)
                 (car var-9029)
                 (car clone-9018))))
  ...)))
```

from the apply/eval code fragment above.

Apart from binding-time issues, the main problem of specializing eval is a representation problem: If eval reflects a function, the representation of the function may not coincide with the specializer's representation of a function. Also, a reflected function should not manipulate code.

2.2 Non-local Exits

A typical application of control operators is to provide non-local exits, similar to exceptions. The following program which computes the product of the elements of a list and immediately returns 0 if any element of the list is 0 serves as an example.

```
(define (product xs)
  (call-with-current-continuation
   (lambda (c)
     (let loop ((xs xs))
       (cond
        ((null? xs) 1)
        ((zero? (car xs)) (c 0))
        (else (* (car xs) (loop (cdr xs)))))))))
```

For this code with `xs` dynamic, binding-time analysis (slightly modified wrt. [7]) yields the following two-level code (see below for a formal definition of two-level terms):

```
(define (product xs)
  (call-with-current-continuation
   (lambda (c : d -> d)
     (let loop ((xs xs))
       (_cond
        ((_null? xs) (_lift 1))
        ((_zero? (_car xs)) (@ c (_lift 0)))
        (else (_* (_car xs) (loop (_cdr xs)))))))))
```

Apparently, the `call-with-current-continuation` can be executed statically, at specialization time, with a continuation `c`, which maps dynamic to dynamic (code to code). However, the effect is disastrous: The specializer starts constructing a variant `product-0` of `product`, captures the continuation, and continues generating a specialized function `loop-0` for the loop because of the dynamic conditional `(_cond ...)` [6]. Then it constructs the body of `loop-0`. As the specializer reaches the first branch of the second conditional (`(zero? (car xs))`) the captured continuation is applied to `0`. Hence, specialization of `loop-0` is indefinitely suspended and the generated code

```
(define (product-0 xs) 0)
```

is clearly wrong.

There are three basic problems in the interaction of `call/cc` with the specializer.

1. A captured continuation can only be invoked at specialization time if the ignored part of the continuation of the specializer coincides with a part of the continuation of the interpreter.

2. No specialization/memoization with respect to a captured continuation should occur as the representation for functions in the specializer may not agree with the representation of the underlying Scheme system.

3. In general, it is unsafe for continuations to be applied to code as the resulting residual code may contain variables which are used outside of their defining scope. However, dynamically closed code (without free variables with binding time dynamic) is fine.

3 Informal Development

In order to provide some intuition about which problems need to be addressed, we embark on an informal discussion of the requirements of a proper treatment of the reflective features. The representation problem is the key problem: The representation of data in the specializer differs from the representation in the underlying Scheme system (*i.e.*, to perform memoization).

3.1 Eval

There are three different ways in which eval might occur, depending on the binding times of the argument and the context. Consider the expression $C[(\mathsf{eval}\ E)]$[2] where C is an arbitrary context.

1. If the binding times of the context $C[\ ]$ and E agree, eval must be performed by the specializer at the appropriate time.

2. If the value of E is available earlier than the context demands it, we can omit the eval altogether and just paste the computed value of E in the dynamic context.

3. The context cannot possess an earlier binding time than E, because it cannot receive the value of E before it is computed.

[2] From now on we assume that the required second argument of eval is always (interaction-environment).

In the interesting case 2 we have to avoid a name capture problem: (eval E) mandates that all identifiers occurring free in E must have a top-level binding. If we inline the value of E in the residual program, these free identifiers might be captured. However, all identifiers in a residual program are "fresh" identifiers produced by a symbol generator, so this problem does not arise in practice.

In summary, the binding time of the expression must be less than (earlier, more static) or equal to that of the context. If the binding times differ, the specializer can treat it similarly to quoted values. It simply inserts the static value (a piece of program text) in the dynamic residual code. Furthermore, we must insist that values produced by eval must be uniform in their binding time and that no memoization is performed on them because of the representation problem.

3.2 Apply

Since eval can produce functions with unknown arity, we need a way to deal with them. Specifically, we must be able to apply such a function to a variable number of arguments. Unfortunately, most offline partial evaluators for Scheme [6, 8] do not treat variadic functions at all.

If we consider argument lists as partially static data structures [25] it is surprisingly simple to handle variadic functions. Hence the possible cases for specialization of C[(apply F E)]:

1. If $C[\ ]$, E, and F are all available at the same binding time, apply is either executed immediately or the code (apply F' E') is generated.
2. If $C[\ ]$ and F are dynamic and E is an argument list of known length with dynamic elements $X_1, \ldots, X_n$ the specializer can omit the application of apply and can directly generate the residual $(F\ X_1 \ldots X_n)$.

3.3 Call/cc

With the mechanics for handling eval in place, no special tricks are needed to specialize call/cc. It is only necessary to provide a binding-time specification which ensures that no memoization point appears between abstraction and use of the continuation (*e.g.*, the discarded part of the continuation of the specializer must coincide with the continuation of the evaluator) and that no code fragments containing free dynamic variables are ever transmitted through the continuation. Our binding-time type system ensures both: for (call/cc (lambda (c) ...)) to be static we require the lambda expression to be completely static and not memoized and enforce the same condition on the continuation variable c. It does not matter in what context (static or dynamic) call/cc occurs: A call/cc can be executed statically in a dynamic context (*e.g.* a dynamic conditional) unless the continuation escapes.

3.4 The Representation Problem

In the above discussion we have come across a problem related to the specializer's representation of functions and other partially static data (data with a static skeleton and dynamic components). In order to perform memoization, the specializer needs an external representation, for example as closures. On the other hand, the use of eval and call/cc demands that the specializer uses the standard representation and also that reflected functions and continuations never handle dynamic data.

We capture these demands on the representation of functions by refinement of the function type. eval and call/cc are restricted to use only *completely static functions.* Such functions use the standard representation and do not touch dynamic data. On the other hand, there are *memoizable functions* which arise as arguments of sp-functions. They are represented as explicit closures. If these two requirements collide, the specializer has to suspend the function to run time. The representation at run time does not matter for the specializer. In the presence of the above three alternatives we obtain a fourth one for free: functions that are not memoized can assume the standard representation, too.

3.5 Further Results

The need to specify a refined binding-time type system yields some additional results:

1. Offline partial evaluators typically treat operators in a duovariant manner. Either they operate on static base values or they operate on dynamic values. In consequence, many useful higher-order functions (such as map, filter, or reduce) cannot be provided externally as operators but have to be interpreted by the specializer. This is inefficient, it causes annoying and artificial binding-times problems for the unwary, and it forces programmers to hack around the limitations.

   Our type system allows for the specification of completely static, unmemoized function types which may be passed to and returned from operators. The type system also deals with the inherent representation problem.

2. For similar reasons, partial evaluators usually do not cater to static higher-order input values and constants of functional type. We can deal with them, provided their completely static type is given.

3. Our binding-time type system computes which functions and data are completely static and which may be memoized. The specializer can take advantage and employ the more expensive encoding as memoized data only when necessary. For example, using the encoding shown in our recent work [30] calling a memoized function entails two function calls, whereas an ordinary function needs just one call.

4. We are not aware of any other offline specializer that deals with variadic functions. Our approach is simple and flexible. The binding-time annotations of programs that do not use variadic functions do not get worse than with a binding-time analysis which only considers fixed-arity functions.

   Sometimes, our binding-time analysis marks functions as static even if they may cause arity mismatches. However, these will only occur during specialization if the original program already had these problems.

4 Formal Development

4.1 Syntax

For our presentation we use higher-order recursion equations (see Fig. 1) in Scheme notation [20]. A definition can define a function of fixed arity or any value, for example a variadic function. An expression is either a variable V, a constant K, a conditional, an application of an external or built-in operator O, a procedure call, an abstraction of a function of fixed arity, a function application to a known number of arguments, an abstraction of a variadic function, a function application to an unknown number of arguments, eval, or call/cc. Among the operators we assume the constructors vcons and vnil, the selectors vcar and vcdr, and the constructor tests vnil? and vcons? to process argument lists. They correspond exactly to the usual list operators. Either a flow analysis could introduce them or we can interpret all standard list operators as argument list operators.

4.2 Type Language

The type language, defined by

$$\begin{array}{rcl}\tau & ::= & S \mid D \mid \tau \to \tau \mid \tau \to_S \tau \mid \tau \to_M \tau \mid \\ & & \mathit{vcons}\ \tau\ \tau \mid \mu\alpha.\tau \mid \alpha\end{array}$$

provides the type of static base values S and the type of dynamic values D. Function space construction $\tau \to_X \tau$ comes in three variants indicated by $X \in \{S, M, \emptyset\}$: S denotes a completely static function, M a memoized function, and $\emptyset$ a function which is neither completely static nor memoized. We usually drop the subscript $\emptyset$. $\mathit{vcons}\ \tau\ \tau$ describes constructed data, in our case argument lists. Finally, we have recursive types $\mu\alpha.\tau$ and type variables α in order to have precise types for recursively constructed data. There will be no rules that explicitly introduce recursive types. Instead we adopt infinite regular type terms in the type language and adopt the $\mu\alpha.\tau$ notation for representing infinite types. Type variables α only occur bound by the explicit μ construction, they never occur in infinite type terms.

Types are ordered by the strict inequalities shown in Fig. 2. All type constructors are monotonic (covariant)

$$\begin{array}{rcl} S & \prec & D \\ \mathit{vcons}\ \tau\ \tau' & \prec & D \\ \tau \to \tau' \prec \tau \to_S \tau' & \prec & D \\ \tau \to \tau' \prec \tau \to_M \tau' & \prec & D \end{array}$$

Figure 2: Ordering on types

in all arguments with respect to the ordering.

The predicates *static* and *memo* on types are defined in Figures 3 and 4. The predicate *static* captures the type of completely static values. A completely static value does not refer to any dynamic value whatsoever. We will see later on how the typing rules constrain the derivation of completely static types.

The predicate *memo* characterizes the type of values which may be memoized. Its use specifies a demand,

$$\begin{array}{l} \mathit{static}(S) \\ \mathit{static}(\tau_1) \wedge \mathit{static}(\tau_2) \Rightarrow \mathit{static}(\tau_1 \to_S \tau_2) \\ \mathit{static}(\tau_1) \wedge \mathit{static}(\tau_2) \Rightarrow \mathit{static}(\mathit{vcons}\ \tau_1\ \tau_2) \\ (\mathit{static}(\alpha) \Rightarrow \mathit{static}(\tau)) \Rightarrow \mathit{static}(\mu\alpha.\tau) \end{array}$$

Figure 3: Completely static types

$$\begin{array}{l} \mathit{memo}(S) \qquad \mathit{memo}(D) \\ \mathit{memo}(\tau_1 \to_M \tau_2) \\ \mathit{memo}(\tau_1) \wedge \mathit{memo}(\tau_2) \Rightarrow \mathit{memo}(\mathit{vcons}\ \tau_1\ \tau_2) \\ (\mathit{memo}(\alpha) \Rightarrow \mathit{memo}(\tau)) \Rightarrow \mathit{memo}(\mu\alpha.\tau) \end{array}$$

Figure 4: Memoizable types

namely that we want to specialize a function with respect to some value. Such a value must have a memoizable type.

4.3 Two-Level Syntax

A two-level syntax specifies the binding-time properties of each syntactic construct [26]. Two-level terms (see Fig. 5) have the dynamic constructs underlined. The specializer statically reduces every construct which is not underlined.

Some constructs come in more than two variants, *i.e.*, function application, abstraction, eval, and apply. For function application and abstraction we must distinguish two different forms of static functions: The standard (lambda ...) constructs a plain static function which cannot be memoized. The alternative ($\overline{\texttt{lambda}}$ ...) constructs a memoizable function which the specializer represents differently (see Sec. 3.5). The companion (@ ...) and ($\overline{\texttt{@}}$ ...) perform application of plain functions and memoizable functions, respectively. The same holds for variadic abstractions and ($\overline{\texttt{apply}}$ ...).

In addition to the underlined and non-underlined versions of eval and apply (with the standard meaning), there exist doubly underlined versions where $\underline{\underline{\texttt{eval}}}$ lifts a static piece of code into a dynamic context and $\underline{\underline{\texttt{apply}}}$ applies a dynamic function to a (static) list of dynamic elements as explained in Sec. 3.1 and 3.2.

Together with the two-level terms we introduce a type system which captures a congruence criterion [21]. In a well-typed two-level term , each static construct only refers to static data. Therefore, all static constructs can be statically reduced and all remaining constructs are dynamic.

4.4 Typing rules

In Figures 6 to 11 we summarize the typing rules for our two-level calculus. They define judgements of the form $A \vdash 2E : \tau$ where $A = \{V_1 : \tau_1, \ldots, V_n : \tau_n\}$ is a set of type assumptions, $2E$ is a two-level term, and τ is its type under assumptions A. We write A^S to assert that $\mathit{static}(\tau_i)$ holds ($1 \leq i \leq n$) and A^M to assert that $\mathit{memo}(\tau_i)$ holds ($1 \leq i \leq n$). $[\tau_1, \ldots, \tau_n]$ abbreviates $\mathit{vcons}\ \tau_1\ (\mathit{vcons}\ \tau_2\ \ldots (\mathit{vcons}\ \tau_n\ \tau') \ldots)$ for some τ'.

$$
\begin{array}{lll}
\multicolumn{3}{l}{V \in \mathsf{Variable}, P \in \mathsf{Procedure}, O \in \mathsf{Operator}, K \in \mathsf{Constant}} \\
\multicolumn{3}{l}{\Pi \in \mathsf{Program}, D \in \mathsf{Definition}, E \in \mathsf{Expression}} \\
\Pi & ::= & D^{+} \\
D & ::= & (\mathsf{define}\ (P\ V^{*})\ E) \mid (\mathsf{define}\ P\ E) \\
E & ::= & V \mid K \mid (\mathsf{if}\ E\ E\ E) \mid (O\ E^{*}) \mid (P\ E^{*}) \mid (\mathsf{lambda}\ (V^{*})\ E) \mid (E\ E^{*}) \mid \\
 & & (\mathsf{lambda}\ V\ E) \mid (\mathsf{apply}\ E\ E) \mid (\mathsf{eval}\ E) \mid (\mathsf{call/cc}\ E)
\end{array}
$$

Figure 1: Syntax of the subject language

$$
\begin{array}{lll}
\multicolumn{3}{l}{2\Pi \in \mathsf{2Program}, 2D \in \mathsf{2Definition}, 2E \in \mathsf{2Expression}} \\
2\Pi & ::= & 2D^{+} \\
2D & ::= & (\mathsf{define}\ (P\ V^{*})\ 2E) \mid (\mathsf{define}\ P\ 2E) \\
2E & ::= & V \mid K \mid (\mathsf{if}\ 2E\ 2E\ 2E) \mid (O\ 2E^{*}) \mid (P\ 2E^{*}) \mid (\mathsf{lambda}\ (V^{*})\ 2E) \mid (\mathsf{@}\ 2E\ 2E^{*}) \mid \\
 & & (\mathsf{lambda}\ V\ 2E) \mid (\mathsf{apply}\ 2E\ 2E) \mid (\mathsf{eval}\ 2E) \mid (\mathsf{call/cc}\ 2E) \mid \\
 & & (\mathsf{lift}\ 2E) \mid (\underline{\mathsf{if}}\ 2E\ 2E\ 2E) \mid (\underline{O}\ 2E^{*}) \mid \\
 & & (\underline{\mathsf{lambda}}\ (V^{*})\ 2E) \mid (\underline{\mathsf{@}}\ 2E\ 2E^{*}) \mid (\overline{\mathsf{lambda}}\ (V^{*})\ 2E) \mid (\overline{\mathsf{@}}\ 2E\ 2E^{*}) \mid \\
 & & (\underline{\mathsf{lambda}}\ V\ 2E) \mid (\underline{\mathsf{apply}}\ 2E\ 2E) \mid (\underline{\underline{\mathsf{apply}}}\ 2E\ 2E) \mid (\overline{\mathsf{lambda}}\ V\ 2E) \mid (\overline{\mathsf{apply}}\ 2E\ 2E) \mid \\
 & & (\underline{\mathsf{eval}}\ 2E) \mid (\underline{\underline{\mathsf{eval}}}\ 2E) \mid (\underline{\mathsf{call/cc}}\ 2E)
\end{array}
$$

Figure 5: Two-level syntax of the subject language

The first group of typing rules (Fig. 6) deals with the first-order fragment of the language. The rules [VAR] and [WEAK] together provide the functionality of the standard axiom $A\{V : \tau\} \vdash V : \tau$. We explicitly include the weakening rule here, as it can drop assumptions on variables that have no influence on the (type of the) expression. In a later set of rules (see Fig. 7 and 8) we will constrain the types in the assumption so as to guarantee properties of the free variables of lambda abstractions. The rules for lifting of static base values in a dynamic context [LIFT], for the conditional [COND, COND-D], and for top-level function calls [CALL, CALL-D] are standard for continuation-based specialization [4, 22], save for the fact that [CALL-D] (the memoizing call) demands that the types of all its arguments are memoizable. However, the treatment of constants and operators is more liberal than usual: The standard rules for them [7]

$$
\begin{array}{ll}
[\text{CONST-STD}] & A \vdash K : S \\[4pt]
[\text{OP-STD}] & \dfrac{A \vdash 2E_1 : S \ldots A \vdash 2E_n : S}{A \vdash (O\ 2E_1 \ldots 2E_n) : S}
\end{array}
$$

constrain constants as well as the arguments and the result of an operator to first-order static values to prevent the representation mismatch mentioned above. Our rules [CONST] and [OP] are more permissive as arguments and result of the operator can have arbitrary types, as long as they are completely static. Otherwise, the operator must be suspended using [OP-D].

The next set of rules (Fig. 7) deals with functions of fixed arity. Here, our ability to constrain exactly the types of the free variables comes into play. Rule [ABS-M] guarantees that the closure of every memoized function only captures variables whose type is memoizable. The types of the arguments of such a function are unconstrained. Similarly, [ABS-S] introduces completely static functions which may only refer to completely static values, while [ABS] introduces standard functions which may process arbitrary values. Rule [ABS-D] is the standard rule for dynamic abstraction. Each of these rules has a companion application rule which types applications of memoized, static, and dynamic functions, respectively.

The same reasoning applies to variadic functions (Fig. 8). Here, we must additionally insist that the argument variables be of type "argument list." Furthermore, we have the [VAPP-D2] rule which deals with the case where the function is dynamic, but the length of the argument list is known (see 3.2).

The final set of rules deals with eval and call/cc. The rule [EVAL] deals with the static eval: The text of the expression must be a static base value and the result must be completely static (not necessarily constrained to base values). In this case, the specializer executes eval. Rule [EVAL-LIFT] deals with static expression texts in dynamic context (quite similar to the standard rule [LIFT]), the corresponding $\underline{\underline{\mathsf{eval}}}$ inserts the code result of the expression into the residual program. Finally, rule [EVAL-D] deals with dynamic expression texts in dynamic contexts. Executing $\underline{\mathsf{eval}}$ generates a call to eval in the residual program.

The construct $\underline{\underline{\mathsf{eval}}}$ is superficially similar to lift as it also coerces a specialization-time value to code. However, lift produces a piece of code which produces the specialization-time value when executed; it generates the quoted value. In contrast, $\underline{\underline{\mathsf{eval}}}$ simply inserts its argument in the residual program without quoting it. Composing $\underline{\mathsf{eval}}$ with lift has the same effect, but with $\underline{\underline{\mathsf{eval}}}$ it is possible to omit eval from the residual program altogether.

In order to safely process call/cc statically, the argument must be a completely static function of type $(\tau \to_S \tau') \to_S \tau$ as demanded by rule [CWCC]. This way, it cannot handle code and it cannot be memoized. Otherwise it must be suspended to the residual program as dictated by rule [CWCC-D].

For completeness, we also exhibit the typing rules for the static operations on argument lists in Fig. 10

$$[\text{VAR}] \quad \{V : \tau\} \vdash V : \tau \qquad [\text{WEAK}] \quad \frac{A \vdash 2E : \tau}{A\{V : \tau'\} \vdash 2E : \tau}$$

$$[\text{CONST}] \quad \frac{\mathit{static}(\tau_K)}{A \vdash K : \tau_K} \qquad [\text{LIFT}] \quad \frac{A \vdash 2E : S}{A \vdash (\mathsf{lift}\ 2E) : D}$$

$$[\text{COND}] \quad \frac{A \vdash 2E_1 : S \quad A \vdash 2E_2 : \tau \quad A \vdash 2E_3 : \tau}{A \vdash (\mathsf{if}\ 2E_1\ 2E_2\ 2E_3) : \tau}$$

$$[\text{COND-D}] \quad \frac{A \vdash 2E_1 : D \quad A \vdash 2E_2 : \tau \quad A \vdash 2E_3 : \tau}{A \vdash (\underline{\mathsf{if}}\ 2E_1\ 2E_2\ 2E_3) : \tau}$$

$$[\text{OP}] \quad \frac{A \vdash 2E_1 : \tau_1 \ldots A \vdash 2E_n : \tau_n \quad \mathit{static}(\tau) \quad \mathit{static}(\tau_1) \ldots \mathit{static}(\tau_n)}{A \vdash (O\ 2E_1 \ldots 2E_n) : \tau}$$

$$[\text{OP-D}] \quad \frac{A \vdash 2E_1 : D \ldots A \vdash 2E_n : D}{A \vdash (\underline{O}\ 2E_1 \ldots 2E_n) : D}$$

$$[\text{CALL}] \quad \frac{A \vdash P : [\tau_1, \ldots, \tau_n] \to \tau \quad A \vdash 2E_1 : \tau_1 \ldots A \vdash 2E_n : \tau_n}{A \vdash (P\ 2E_1 \ldots 2E_n) : \tau}$$

$$[\text{CALL-D}] \quad \frac{A \vdash P : [\tau_1, \ldots, \tau_n] \to D \quad A \vdash 2E_1 : \tau_1 \ldots A \vdash 2E_n : \tau_n \quad \mathit{memo}(\tau_1) \ldots \mathit{memo}(\tau_n)}{A \vdash (\underline{P}\ 2E_1 \ldots 2E_n) : D}$$

Figure 6: Typing Rules for the first-order fragment

$$[\text{ABS-M}] \quad \frac{A^M\{V_1 : \tau_1, \ldots, V_n : \tau_n\} \vdash 2E : \tau}{A \vdash (\overline{\mathsf{lambda}}\ (V_1 \ldots V_n)\ 2E) : [\tau_1, \ldots, \tau_n] \to_M \tau}$$

$$[\text{ABS-S}] \quad \frac{A^S\{V_1 : \tau_1, \ldots, V_n : \tau_n\} \vdash 2E : \tau \quad \mathit{static}(\tau_1) \ldots \mathit{static}(\tau_n) \quad \mathit{static}(\tau)}{A \vdash (\mathsf{lambda}\ (V_1 \ldots V_n)\ 2E) : [\tau_1, \ldots, \tau_n] \to_S \tau}$$

$$[\text{ABS}] \quad \frac{A\{V_1 : \tau_1, \ldots, V_n : \tau_n\} \vdash 2E : \tau}{A \vdash (\mathsf{lambda}\ (V_1 \ldots V_n)\ 2E) : [\tau_1, \ldots, \tau_n] \to \tau}$$

$$[\text{ABS-D}] \quad \frac{A\{V_1 : D, \ldots, V_n : D\} \vdash 2E : D}{A \vdash (\underline{\mathsf{lambda}}\ (V_1 \ldots V_n)\ 2E) : D}$$

$$[\text{APP-M}] \quad \frac{A \vdash 2E : [\tau_1, \ldots, \tau_n] \to_M \tau \quad A \vdash 2E_1 : \tau_1 \ldots A \vdash 2E_n : \tau_n}{A \vdash (\overline{@}\ 2E\ 2E_1 \ldots 2E_n) : \tau}$$

$$[\text{APP-S}] \quad \frac{A \vdash 2E : [\tau_1, \ldots, \tau_n] \to_X \tau \quad A \vdash 2E_1 : \tau_1 \ldots A \vdash 2E_n : \tau_n \quad X \in \{\emptyset, S\}}{A \vdash (@\ 2E\ 2E_1 \ldots 2E_n) : \tau}$$

$$[\text{APP-D}] \quad \frac{A \vdash 2E : D \quad A \vdash 2E_1 : D \ldots A \vdash 2E_n : D}{A \vdash (\underline{@}\ 2E\ 2E_1 \ldots 2E_n) : D}$$

Figure 7: Typing Rules for functions with fixed arity

$$[\text{VABS-M}] \quad \frac{A^M\{V : [\tau_1, \ldots, \tau_n]\} \vdash 2E : \tau}{A \vdash (\overline{\mathsf{lambda}}\ V\ 2E) : [\tau_1, \ldots, \tau_n] \to_M \tau}$$

$$[\text{VABS-S}] \quad \frac{A^S\{V : [\tau_1, \ldots, \tau_n]\} \vdash 2E : \tau \quad \mathit{static}(\tau) \quad \mathit{static}(\tau_1) \ldots \mathit{static}(\tau_n)}{A \vdash (\mathsf{lambda}\ V\ 2E) : [\tau_1, \ldots, \tau_n] \to_S \tau}$$

$$[\text{VABS}] \quad \frac{A\{V : [\tau_1, \ldots, \tau_n]\} \vdash 2E : \tau}{A \vdash (\mathsf{lambda}\ V\ 2E) : [\tau_1, \ldots, \tau_n] \to \tau}$$

$$[\text{VABS-D}] \quad \frac{A\{V : D\} \vdash 2E : D}{A \vdash (\underline{\mathsf{lambda}}\ V\ 2E) : D}$$

$$[\text{VAPP-M}] \quad \frac{A \vdash 2E_1 : \tau' \to_M \tau \quad A \vdash 2E_2 : \tau'}{A \vdash (\overline{\mathsf{apply}}\ 2E_1\ 2E_2) : \tau}$$

$$[\text{VAPP-S}] \quad \frac{A \vdash 2E_1 : \tau' \to_X \tau \quad A \vdash 2E_2 : \tau' \quad X \in \{\emptyset, S\}}{A \vdash (\mathsf{apply}\ 2E_1\ 2E_2) : \tau}$$

$$[\text{VAPP-D}] \quad \frac{A \vdash 2E_1 : D \quad A \vdash 2E_2 : D}{A \vdash (\underline{\mathsf{apply}}\ 2E_1\ 2E_2) : D}$$

$$[\text{VAPP-D2}] \quad \frac{A \vdash 2E_1 : \overline{D} \quad A \vdash 2E_2 : [D, \ldots, D]}{A \vdash (\underline{\underline{\mathsf{apply}}}\ 2E_1\ 2E_2) : D}$$

Figure 8: Typing Rules for variadic functions

$$
\text{[EVAL]} \quad \frac{A \vdash 2E : S \quad \mathit{static}(\tau)}{A \vdash (\mathsf{eval}\ 2E) : \tau}
$$

$$
\text{[EVAL-LIFT]} \quad \frac{A \vdash 2E : S}{A \vdash (\underline{\underline{\mathsf{eval}}}\ 2E) : D}
$$

$$
\text{[EVAL-D]} \quad \frac{A \vdash \overline{2E} : D}{A \vdash (\underline{\mathsf{eval}}\ 2E) : D}
$$

$$
\text{[CWCC]} \quad \frac{A \vdash 2E : (\tau \rightarrow_S \tau') \rightarrow_S \tau \quad \mathit{static}(\tau) \quad \mathit{static}(\tau')}{A \vdash (\mathsf{call/cc}\ 2E) : \tau}
$$

$$
\text{[CWCC-D]} \quad \frac{A \vdash 2E : D}{A \vdash (\underline{\mathsf{call/cc}}\ 2E) : D}
$$

Figure 9: Typing Rules for eval and call/cc

$$
\text{[VCONS]} \quad \frac{A \vdash 2E_1 : \tau_1 \quad A \vdash 2E_2 : \tau_2}{A \vdash (\mathsf{vcons}\ 2E_1\ 2E_2) : \mathit{vcons}\ \tau_1\ \tau_2}
$$

$$
\text{[VNIL]} \quad A \vdash (\mathsf{vnil}) : \mathit{vcons}\ \tau_1\ \tau_2
$$

$$
\text{[VCAR]} \quad \frac{A \vdash 2E : \mathit{vcons}\ \tau_1\ \tau_2}{A \vdash (\mathsf{vcar}\ 2E) : \tau_1}
$$

$$
\text{[VCDR]} \quad \frac{A \vdash 2E : \mathit{vcons}\ \tau_1\ \tau_2}{A \vdash (\mathsf{vcdr}\ 2E) : \tau_2}
$$

$$
\text{[VCONS?]} \quad \frac{A \vdash 2E : \mathit{vcons}\ \tau_1\ \tau_2}{A \vdash (\mathsf{vcons?}\ 2E) : S}
$$

$$
\text{[VNIL?]} \quad \frac{A \vdash 2E : \mathit{vcons}\ \tau_1\ \tau_2}{A \vdash (\mathsf{vnil?}\ 2E) : S}
$$

Figure 10: Typing Rules for operations on variable argument lists

$$
\text{[DEF]} \quad \frac{A\{P : [\tau_1, \ldots, \tau_n] \rightarrow \tau, V_1 : \tau_1, \ldots, V_n : \tau_n\} \vdash 2E : \tau}{A\{P : [\tau_1, \ldots, \tau_n] \rightarrow \tau\} \vdash_D (\mathsf{define}\ (P\ V_1 \ldots V_n)\ 2E)}
$$

$$
\text{[VDEF]} \quad \frac{A\{P : \tau' \rightarrow \tau\} \vdash 2E : \tau' \rightarrow \tau}{A\{P : \tau' \rightarrow \tau\} \vdash_D (\mathsf{define}\ P\ 2E)}
$$

$$
\text{[PROG]} \quad \frac{A \vdash_D D_1 \ldots A \vdash_D D_n}{\vdash_\Pi D_1 \ldots D_n}
$$

Figure 11: Typing Rule for Programs

and the rule for typing two-level programs in Fig. 11. We have omitted the dynamic variants of the argument list operations as they are identical to the [OP-D] rule in Fig. 6. Definitions and programs are typed using different judgements $A \vdash_D D$, which relates a definition to the remaining program, and $\vdash_\Pi \Pi$, which states that all definitions in Π are consistently typed with respect to all their uses.

Finally, we define a *completion* of a (standard) program Π to be a well-typed two-level program 2Π such that Π is obtained by erasing all overlinings and underlinings, as well as omitting the (lift ...) construct.

In the above description we distinguish four different states that a function may assume: static, completely static, memoized, and dynamic. Every non-memoized function can be represented by a standard abstraction, which may result in faster specialization.

5 Type Reconstruction

In order to put the type system of the preceding section to work we need an efficient type reconstruction algorithm which—given the binding-times of the arguments of a goal function—automatically constructs a well-typed two-level program from a program. This two-level program should be minimal in a sense to be defined below.

5.1 A Syntax-Directed Type System

Here we are interested in finding a completion for a standard program Π. We will do so by deriving a constraint system from the type system shown above where we combine the different rules for the two-level variants of a construct. This combined type system must be syntax-directed (*i.e.*, only one rule is applicable to each construct) to be suitable for type reconstruction. Therefore we have to incorporate applications of [WEAK] and [LIFT] into the appropriate rules. For example, the different rules for (fixed-arity) abstrac-

$$\frac{\begin{array}{c}\{V'_1 : \tau'_1, \ldots, V'_m : \tau'_m\} \cup \{V_1 : \tau_1, \ldots, V_n : \tau_n\} \vdash \mathsf{E} : \tau_0 \\ \{V'_1, \ldots, V'_m\} = FV(\mathsf{lambda}\ (V_1 \ldots V_n)\ E) \\ \text{either } \tau = [\tau_1, \ldots, \tau_n] \rightarrow_M \tau_0 \wedge memo(\tau'_1) \ldots memo(\tau'_m) \\ \text{or } \tau = [\tau_1, \ldots, \tau_n] \rightarrow_S \tau_0 \wedge static(\tau'_1) \ldots static(\tau'_m) \wedge static(\tau_0) \ldots static(\tau_n) \\ \text{or } \tau = [\tau_1, \ldots, \tau_n] \rightarrow \tau_0 \\ \text{or } \tau = \tau_0 = \cdots = \tau_n = D\end{array}}{A\{V'_1 : \tau'_1, \ldots, V'_m : \tau'_m\} \vdash (\mathsf{lambda}\ (V_1 \ldots V_n)\ E) : \tau} \quad \text{[ABS-COMB]}$$

Figure 12: Combined abstraction rules

tion are combined with [WEAK] to (using $FV(E)$ to denote the free variables of E) the rule [ABS-COMB] in Fig. 12.

5.2 Constraint System

A *constraint* consists of an n-ary constraint constructor applied to n constraint variables $\phi, \phi_0, \ldots$. The semantics of an n-ary constraint c is an n-ary relation $\vartheta(c)$ on type terms. A solution σ of a constraint set C is a substitution which maps constraint variables to types such that for each constraint $c(\phi_1, \ldots, \phi_n)$ the types $\sigma(\phi_1), \ldots, \sigma(\phi_n)$ are related by $\vartheta(c)$. Figure 13 summarizes syntax and semantics of the constraints for our type system. We omit the tedious specification of ϑ and simply state the corresponding condition on solutions.

The *equality* constraint denotes equality of its arguments. The *function* constraint specifies a function which may be static, memoized, or dynamic. *S-function* specifies a function which may be completely static or dynamic. *M-function* specifies a function which may be memoized or dynamic. *Structure* specifies a static or dynamic data structure. The *base value* constraint specifies a static or dynamic base value. An *S-dependency* $\phi_1 \underset{S}{\Rightarrow} \phi_2$ states that staticness of ϕ_1 implies staticness of ϕ_2. Likewise, an *M-dependency* $\phi_1 \underset{M}{\Rightarrow} \phi_2$ states that memoizability of ϕ_1 implies memoizability of ϕ_2. The constraint $\neg_S(\phi)$ states that ϕ is not completely static and *memo* (ϕ) states that ϕ must be memoizable. Both do not occur in initial constraint sets, but are introduced later on. A *dependency* $\phi_1 \rhd \phi_2$ states that if ϕ_1 is dynamic then so is ϕ_2. An *L-dependency* $\phi_1 \underset{L}{\rhd} \phi_2$ is needed to specify the type constraints for apply: If ϕ_1 is dynamic then ϕ_2 is a list of dynamic elements; if both are lists then their elements are equal. A *lift* constraint $\phi_1 \rightsquigarrow \phi_2$ states that either ϕ_1 is a static base value and ϕ_2 is dynamic or that both ϕ_1 and ϕ_2 are equal.

By collapsing the different typing rules for a construct and its different overlined and underlined versions into one we can generate the constraint set which must be solved by a type derivation for some expression E. A minimal solution of the resulting constraint set can be found using methods similar to those in Henglein's work [18]. However, the resulting type inference algorithm is at least quadratic in the size of the input program, due to the dependency on the environment. Details may be found in a companion technical report [31].

6 Outline of the Specializer

A specification of the specializer for the full language treated in this paper is bound to be bulky. Therefore, we confine the specification to a simple but illustrative subset which excludes multi-argument and variadic functions. It is a continuation-based specializer [4, 22] which makes the treatment of call/cc explicit.

In Fig. 14 we define the specializer $\mathcal{S}$ that maps a two-level expression, an environment, and a specialization continuation to a specialization value. On the right side of the definition we use a call-by-value lambda with the McCarthy conditional _ → _, _ and some other extensions. The specializer has an implicit constant argument Π—the current source program—that is accessed in the interpretation of procedure calls as $\Pi(P)$: the body expression of procedure P. The auxiliary functions $\mathcal{K}$ and $\mathcal{O}$ map constants and operators to their denotations. *eval* is the built-in eval function, *mk-constant* maps a specialization-time constant to a run-time constant. The different *mk-*... functions all receive Variable and Code arguments and produce Code. The exception is $mk\text{-}closure(E, V, \rho)$ which builds a closure $closure(E, V, \rho')$ where the environment ρ' is equal to ρ restricted to the free variables of (lambda (V) E). The use of $V^\diamond$ on the right sides denotes the generation of a fresh variable.

$$\begin{array}{rcl}\mathcal{S} & : & \mathsf{2Expression} \rightarrow \mathsf{Env} \rightarrow \mathsf{Cont} \rightarrow \mathsf{PEVal} \\ \rho & \in & \mathsf{Env} = \mathsf{Variable} \rightarrow \mathsf{PEVal} \\ k & \in & \mathsf{Cont} = \mathsf{PEVal} \rightarrow \mathsf{PEVal} \\ \mathsf{PEVal} & = & \mathsf{BaseValue}+ \\ & & \mathsf{PEVal} \rightarrow \mathsf{PEVal}+ \\ & & closure(\mathsf{2Expression}, \mathsf{Variable}, \mathsf{Env})+ \\ & & \mathsf{Code} \\ \mathcal{K} & : & \mathsf{Constant} \rightarrow \mathsf{PEVal} \\ \mathcal{O} & : & \mathsf{Operator} \rightarrow \mathsf{PEVal}^* \rightarrow \mathsf{PEVal} \\ eval & : & \mathsf{BaseValue} \rightarrow \mathsf{PEVal} \\ mk\text{-}constant & : & \mathsf{BaseValue} \rightarrow \mathsf{PEVal}\end{array}$$

Some further functions are involved with the memoization mechanism. First, there is a global cache which is accessed by $cache\text{-}enter(P, \overline{s})$ where $P \in$ Procedure and $\overline{s}$ is a vector of static values. It associates the pair $(P, \overline{s})$ with a procedure name $P^\diamond$ which is either a fresh name—if the pair was not in the cache—or the name assigned to the pair in a previous cache lookup. The function *cache-definition* (define $(P\ V_1 \ldots V_n)$ E) adds a new function to the residual program, which is also kept in a global variable. The remaining functions *project-static*, *project-dynamic*, and *clone-dynamic* extract the static and dynamic parts from partially static specialization-time values. They are explained else-

constraint	syntax	σ is solution if
equality	$\phi_1 = \phi_2$	$\sigma(\phi_1) = \sigma(\phi_2)$
dynamic	$D = \phi_1$	$\sigma(\phi_1) = D$
function	$\langle\phi_1 \rightarrow \phi_2\rangle \preceq \phi_0$	$\sigma(\phi_1) \rightarrow \sigma(\phi_2) = \sigma(\phi_0) \vee$
		$\sigma(\phi_1) \rightarrow_S \sigma(\phi_2) = \sigma(\phi_0) \vee$
		$\sigma(\phi_1) \rightarrow_M \sigma(\phi_2) = \sigma(\phi_0) \vee$
		$\sigma(\phi_1) = \sigma(\phi_2) = \sigma(\phi_0) = D$
S-function	$\langle\phi_1 \rightarrow_S \phi_2\rangle \preceq \phi_0$	$\sigma(\phi_1) \rightarrow_S \sigma(\phi_2) = \sigma(\phi_0) \vee$
		$\sigma(\phi_1) = \sigma(\phi_2) = \sigma(\phi_0) = D$
M-function	$\langle\phi_1 \rightarrow_M \phi_2\rangle \preceq \phi_0$	$\sigma(\phi_1) \rightarrow_M \sigma(\phi_2) = \sigma(\phi_0) \vee$
		$\sigma(\phi_1) = \sigma(\phi_2) = \sigma(\phi_0) = D$
structure	$\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi_0$	$\mathit{vcons}\ \sigma(\phi_1)\ \sigma(\phi_2) = \sigma(\phi_0) \vee$
		$\sigma(\phi_1) = \sigma(\phi_2) = \sigma(\phi_0) = D$
base value	$S \preceq \phi_1$	$S \preceq \sigma(\phi_1)$
not static	$\neg_S(\phi_1)$	$\neg \mathit{static}(\sigma(\phi_1))$
memoized	$\mathit{memo}\ (\phi_1)$	$\mathit{memo}(\sigma(\phi_1))$
S-dependency	$\phi_1 \underset{S}{\Rightarrow} \phi_2$	$\mathit{static}(\sigma(\phi_1)) \Rightarrow \mathit{static}(\sigma(\phi_2))$
M-dependency	$\phi_1 \underset{M}{\Rightarrow} \phi_2$	$\mathit{memo}(\sigma(\phi_1)) \Rightarrow \mathit{memo}(\sigma(\phi_2))$
dependency	$\phi_1 \rhd \phi_2$	$\sigma(\phi_1) = D \Rightarrow \sigma(\phi_2) = D$
L-dependency	$\phi_1 \underset{L}{\rhd} \phi_2$	$\sigma(\phi_1) = D \wedge \sigma(\phi_2) = [D, \ldots, D] \vee$
		$\sigma(\phi_1) = \sigma(\phi_2) = [\tau_1, \ldots, \tau_n] \vee$
		$\sigma(\phi_1) = \sigma(\phi_2) = D$
lift	$\phi_1 \rightsquigarrow \phi_2$	$\sigma(\phi_1) = S \wedge \sigma(\phi_2) = D \vee \sigma(\phi_1) = \sigma(\phi_2)$

Figure 13: Syntax and Semantics of Constraints

$$
\begin{array}{lcl}
\mathcal{S}[\![V]\!]\rho & = & \lambda k.k(\rho V) \\
\mathcal{S}[\![K]\!]\rho & = & \lambda k.k(\mathcal{K}[\![K]\!]) \\
\mathcal{S}[\![(\mathsf{if}\ E_1\ E_2\ E_3)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda v_1.v_1 \rightarrow \mathcal{S}[\![E_2]\!]\rho k, \mathcal{S}[\![E_3]\!]\rho k) \\
\mathcal{S}[\![(O\ E_1 \ldots E_n)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda v_1 \ldots \mathcal{S}[\![E_n]\!]\rho(\lambda v_n.k(\mathcal{O}[\![O]\!](v_1, \ldots, v_n))) \ldots) \\
\mathcal{S}[\![(\mathsf{call}\ P\ E_1 \ldots E_n)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda v_1 \ldots \mathcal{S}[\![E_n]\!]\rho(\lambda v_n.\mathcal{S}[\![\Pi(P)]\!][V_i \mapsto v_i]k) \ldots) \\
\mathcal{S}[\![(\mathsf{lambda}\ (V)\ E)]\!]\rho & = & \lambda k.k(\lambda y.\lambda k'.\mathcal{S}[\![E]\!]\rho[V \mapsto y]k') \\
\mathcal{S}[\![(\mathsf{@}\ E_1\ E_2)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda v_1.\mathcal{S}[\![E_2]\!]\rho(\lambda v_2.(v_1\ v_2)\ k)) \\
\mathcal{S}[\![(\mathsf{eval}\ E)]\!]\rho & = & \lambda k.\mathcal{S}[\![E]\!]\rho(\lambda v.k(\mathit{eval}\ v)) \\
\mathcal{S}[\![(\mathsf{call/cc}\ E)]\!]\rho & = & \lambda k.\mathcal{S}[\![E]\!]\rho(\lambda v.(v(\lambda a.\lambda k'.ka))k) \\
\mathcal{S}[\![(\mathsf{lift}\ E)]\!]\rho & = & \lambda k.\mathcal{S}[\![E]\!]\rho(\lambda v.k(\mathit{mk\text{-}constant}\ v)) \\
\mathcal{S}[\![(\underline{\mathsf{if}}\ E_1\ E_2\ E_3)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda v_1.\mathit{mk\text{-}if}(v_1, \mathcal{S}[\![E_2]\!]\rho k, \mathcal{S}[\![E_3]\!]\rho k)) \\
\mathcal{S}[\![(\underline{O}\ E_1 \ldots E_n)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda v_1 \ldots \mathcal{S}[\![E_n]\!]\rho(\lambda v_n.k(\mathit{mk\text{-}O}(v_1, \ldots, v_n))) \ldots) \\
\mathcal{S}[\![(\underline{\mathsf{call}}\ P\ E_1 \ldots E_n)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda v_1 \ldots \mathcal{S}[\![E_n]\!]\rho(\lambda v_n. \\
& & \quad \mathit{let}\ (s_1, \ldots, s_k) = \mathit{project\text{-}static}(v_1, \ldots, v_n) \\
& & \quad \mathit{let}\ (d_1, \ldots, d_l) = \mathit{project\text{-}dynamic}(v_1, \ldots, v_n) \\
& & \quad \mathit{let}\ (y_1, \ldots, y_n) = \mathit{clone\text{-}dynamic}(v_1, \ldots, v_n) \\
& & \quad \mathit{let}\ (Z_1, \ldots, Z_l) = \mathit{project\text{-}dynamic}(y_1, \ldots, y_n) \\
& & \quad \mathit{let}\ P^{\diamond} = \mathit{cache\text{-}enter}(P, (s_1, \ldots, s_k)) \\
& & \quad \mathit{if}\ P^{\diamond}\ \mathit{was\ not\ in\ the\ cache\ before\ then} \\
& & \quad\quad \mathit{cache\text{-}definition}\ (\mathsf{define}\ (P^{\diamond}\ Z_1 \ldots Z_l)\ \mathcal{S}[\![\Pi(P)]\!][V_i \mapsto y_i](\lambda z.z)) \\
& & \quad k(\mathit{mk\text{-}call}(P^{\diamond}, d_1, \ldots, d_l)))) \ldots) \\
\mathcal{S}[\![(\underline{\mathsf{lambda}}\ (V)\ E)]\!]\rho & = & \lambda k.k(\mathit{mk\text{-}\lambda}(V^{\diamond}, \mathcal{S}[\![E]\!]\rho[V \mapsto V^{\diamond}](\lambda z.z))) \\
\mathcal{S}[\![(\underline{\mathsf{@}}\ E_1\ E_2)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda v_1.\mathcal{S}[\![E_2]\!]\rho(\lambda v_2.k(\mathit{mk\text{-}@}(v_1, v_2)))) \\
\mathcal{S}[\![(\overline{\mathsf{lambda}}\ (V)\ E)]\!]\rho & = & \lambda k.k(\mathit{mk\text{-}closure}(E, V, \rho)) \\
\mathcal{S}[\![(\overline{\mathsf{@}}\ E_1\ E_2)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda \mathit{closure}(E, V, \rho').\mathcal{S}[\![E_2]\!]\rho(\lambda v_2.\mathcal{S}[\![E]\!]\rho'[V \mapsto v_2]k)) \\
\mathcal{S}[\![(\underline{\mathsf{eval}}\ E)]\!]\rho & = & \lambda k.\mathcal{S}[\![E]\!]\rho(\lambda v.k(\mathit{mk\text{-}eval}(v))) \\
\mathcal{S}[\![(\mathsf{call/cc}\ E)]\!]\rho & = & \lambda k.\mathcal{S}[\![E]\!]\rho(\lambda v.k(\mathit{mk\text{-}call/cc}(v))) \\
\mathcal{S}[\![(\overline{\underline{\underline{\mathsf{eval}}}\ E})]\!]\rho & = & \lambda k.\mathcal{S}[\![E]\!]\rho k
\end{array}
$$

Figure 14: Specification of a core specializer

where [21, 30]. These functions only work on abstractions represented by closures.

7 Related Work

Constraint-based binding-time analysis has been pioneered by Henglein [18]. His main contribution is the efficient linear-time implementation of type reconstruction. His algorithm is the basis of our algorithm. His work in turn has been influenced by Gomard's work on partial type inference [15]. Bondorf and Jørgensen took up Henglein's ideas on constraint normalization to build a binding-time analysis for Similix [7]. Their system is closest to ours. However, whereas our analysis is systematically derived from a two-level type system, theirs is constructed by inspection of the specializer. Our analysis is more powerful as it deals with variadic functions, enables functional arguments and results of operators, and can be used to guide the choice of representations in the specializer.

In our treatment of eval we assume that the argument is an ordinary base type value and leave the connection to the type of the result unconstrained: It can be whatever the context expects it to be. There are also type systems to ensure the type correctness of computed code [28]. These approaches could be used to provide more precise types for eval.

Consel and Danvy [9] deal with a problem similar to the representation problem. They propose to split a program into three parts prior to specialization proper: completely static combinators, completely dynamic combinators, and the part which is actually subject to specialization. Our type system can also be used to identify completely static parts and it can also remove interpretive overhead in other places.

Birkedal and Welinder [2] treat partial evaluation for standard ML. Their binding-time analysis is also monovariant and constraint-based. Their additions to Henglein's framework concern mainly pattern matching. Another interesting point of their work is their proposed treatment of exceptions. However, their proposal is not implemented and it appears that our treatment of call/cc subsumes their treatment of exceptions.

Heintze's set-based analysis [17] is amenable to the analysis of higher-order languages in the presence of side-effects and control features. Its variant for binding-time analysis (by Malmkjaer, Heintze, and Danvy [24]) can therefore also be employed to analyze programs using call/cc. However, it is not clear how their specializer would handle call/cc.

Recently, more liberal type-based approaches to binding-time analysis and partial evaluation have been developed. Our analysis is monovariant and performs binding-time coercions only on base types. The work of Dussart, Mossin, and Henglein [19, 14] pursues polymorphic binding-time type systems where functions can be used at more than one binding time. Danvy, Malmkjær, and Palsberg [12, 13] consider binding-time coercions at higher types, product types, and sum types. Danvy [11] even introduces a type-driven scheme which achieves specialization only through the use of binding-time coercions.

Danvy [10] considers the relation between partial evaluation and reflection. He states that partial evaluation collapses two levels in a tower of interpreters and thus removes reflective procedures mediating between the collapsed levels.

8 Conclusions

We have succeeded in extending offline partial evaluation of Scheme to include reflective features, *i.e.*, eval, apply, and call/cc. The main problem to overcome is a representation problem. We have specified a binding-time type system which solves the representation problem and outlined a specializer which reduces binding-time annotated programs. The type system is amenable to efficient type reconstruction. We have applied our specializer to realistic examples, *e.g.*, parser generation.

We must mention that our treatment of call/cc is quite conservative, as dynamically closed code may be safely transmitted through a static continuation. Also, the "return type" τ' of a continuation need not be identical for all applications of the continuation as it never returns. As the detection of dynamically closed lambdas would require a different type of static function we have not attempted to add the more liberal treatment to our system.

We have not considered uses of eval which modify the subject program or the running interpreter. This is an interesting subject of its own.

It would be interesting to investigate the combination of our system with polymorphic binding-time analysis [14] and/or binding-time coercions [11]. Coercions might also prove useful to transform between completely static and memoized functions.

A final aspect of Scheme that has not been addressed so far is the direct use of mutable variables. The indirect solution which deals with global variables as dynamic abstract data types [6] works in our framework as well, but a direct treatment has not been realized (except for the C language [1]).

Acknowledgements

Thanks to Michael Sperber for reading and commenting on a draft of this paper. Special thanks to the anonymous referees who provided detailed and thought-provoking comments.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, May 1994.

[2] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen, 1993.

[3] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Programming*, 17:3–34, 1991.

[4] A. Bondorf. Improving binding-times without explicit CPS conversion. In *Proc. Conference on Lisp and Functional Programming*, pages 1–10, San Francisco, CA, USA, June 1992.

[5] A. Bondorf. *Similix 5.0 Manual*. DIKU, University of Copenhagen, May 1993.

[6] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Programming*, 19(2):151–195, 1991.

[7] A. Bondorf and J. Jørgensen. Efficient analysis for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[8] C. Consel. A tour of Schism. In D. Schmidt, editor, *Symp. Partial Evaluation and Semantics-Based Program Manipulation '93*, pages 134–154, Copenhagen, Denmark, June 1993. ACM.

[9] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. D. Jones, editor, *Proc. 3rd European Symposium on Programming 1990*, pages 88–105, Copenhagen, Denmark, 1990. Springer-Verlag. LNCS 432.

[10] O. Danvy. Across the bridge between reflection and partial evaluation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 83–116, Amsterdam, 1988. North-Holland.

[11] O. Danvy. Type-directed partial evaluation. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, page ??, St. Petersburg, Fla., Jan. 1996. ACM Press.

[12] O. Danvy, K. Malmkjær, and J. Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, July 1995.

[13] O. Danvy, K. Malmkjær, and J. Palsberg. Eta-expansion does The Trick. Technical Report BRICS RS-95-41, Computer Science Dept., Aarhus University, Denmark, Aug. 1995.

[14] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In A. Mycroft, editor, *Proc. International Static Analysis Symposium, SAS'95*, pages 118–136, Glasgow, Scotland, Sept. 1995. Springer-Verlag. LNCS 983.

[15] C. K. Gomard. Partial type inference for untyped functional programs. In *Proc. Conference on Lisp and Functional Programming*, pages 282–287, Nice, France, 1990. ACM.

[16] C. K. Gomard and N. D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–70, Jan. 1991.

[17] N. Heintze. Set-based analysis of ML-programs. In LFP1994 [23], pages 306–317.

[18] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, pages 448–472, Cambridge, MA, 1991. Springer-Verlag. LNCS 523.

[19] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Proc. 5th European Symposium on Programming*, pages 287–301, Edinburgh, UK, Apr. 1994. Springer-Verlag. LNCS 788.

[20] Institute of Electrical and Electronic Engineers, Inc. IEEE standard for the Scheme programming language. IEEE Std 1178-1990, New York, NY, 1991.

[21] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[22] J. Lawall and O. Danvy. Continuation-based partial evaluation. In LFP1994 [23], pages 227–238.

[23] *Proc. Conference on Lisp and Functional Programming*, Orlando, Fla, USA, June 1994. ACM Press.

[24] K. Malmkjær, O. Danvy, and N. Heintze. ML partial evaluation using set-based analysis. In *Record of the ACM-SIGPLAN Workshop on ML and its Applications*, number 2265 in INRIA Research Report, pages 112–119, BP 105, 78153 Le Chesnay Cedex, France, June 1994.

[25] T. Æ. Mogensen. Separating binding times in language specifications. In *Proc. Functional Programming Languages and Computer Architecture 1989*, pages 14–25, London, GB, 1989.

[26] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[27] J. Rees. The Scheme of things: The June 1992 meeting. *Lisp Pointers*, V(4), Oct. 1992.

[28] T. Sheard and N. Nelson. Type safe abstractions using program generators. Technical Report 95-013, Oregon Graduate Institute of Science and Technology, PO Box 91000, Portland, OR 97291-1000 USA, July 1995.

[29] M. Sperber and P. Thiemann. The essence of LR parsing. In W. Scherlis, editor, *ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation '95*, pages 146–155, La Jolla, CA, June 1995. ACM Press.

[30] P. Thiemann. Cogen in six lines. Technical Report WSI-95-XX, Universität Tübingen, Oct. 1995.

[31] P. Thiemann. Towards partial evaluation of full Scheme. Technical Report WSI-95-XX, Universität Tübingen, Nov. 1995.

A Generating and solving constraint sets

We generate a constraint set from an expression E by application of the recursive function $C(E)$ to be defined below. Due to the presence of the rule [LIFT] we assume that for each expression there are two constraint variables $\check{\phi}_E$ and $\hat{\phi}_E$ related by the constraint $\check{\phi}_E \rightsquigarrow \hat{\phi}_E$. Furthermore, for each binding occurrence of a variable V and each definition of a procedure P there is a constraint variable ϕ_V and ϕ_P, respectively.

The definition of $C(E)$ in Fig. 15 is by cases on E where we omit the obvious recursive calls $\bigcup C(E_i)$ on the subterms. We also use the abbreviation $c[\phi_1, \ldots, \phi_n]$ for $c[\phi_1']$, *vcons* $\phi_i\ \phi_{i+1}' \preceq \phi_i'$ $(1 \leq i \leq n)$.

A.1 Constraint normalization

A minimal solution for an arbitrary set of constraints is found by normalization to a form where the solution is obvious. Figure 16 shows the normalization rules. Constraint set C rewrites to C' in one labeled transition step ($C \Longrightarrow_{id} C'$) if there is a rule $\frac{C_1}{C_1'}$, $C = C_0 \cup C_1$, and $C' = C_0 \cup C_1'$. The label is a variable substitution, here the identity substitution.

A labeled transition rule propagates equalities between constraint variables:

$$C \cup \{\phi = \phi'\} \Longrightarrow_{[\phi \mapsto \phi']} C[\phi \mapsto \phi']$$

We define the reflexive and transitive closure of labeled transitions $\stackrel{*}{\Longrightarrow}$ by: $C \stackrel{*}{\Longrightarrow}_{id} C$, and if $C \Longrightarrow_{\gamma_1} C'$ and $C' \stackrel{*}{\Longrightarrow}_{\gamma_2} C''$ then $C \stackrel{*}{\Longrightarrow}_{\gamma_2 \circ \gamma_1} C'$. The normalization rules are sound and complete: Finite application of transition rules to a constraint set does not change the set of solutions.

Theorem 1 *Let $C \stackrel{*}{\Longrightarrow}_{\gamma} C'$. Substitution $\sigma \circ \gamma$ solves C iff σ solves C'.*

Proof: By inspection of the rules in Fig. 16 against the semantics of constraints in Fig. 13.

As an example we consider rule (25):

$$\frac{\begin{array}{c} \mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad \mathit{vcons}\ \phi_1'\ \phi_2' \preceq \phi' \\ \phi \underset{L}{\triangleright} \phi' \end{array}}{\begin{array}{c} \mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad \mathit{vcons}\ \phi_1'\ \phi_2' \preceq \phi' \\ \phi_1 = \phi_1' \quad \phi' \triangleright \phi \quad \phi_2' \triangleright \phi' \\ \phi_2 \triangleright \phi \quad \phi_2 \underset{L}{\triangleright} \phi_2' \end{array}}$$

which is the most complicated rule. Let σ be a solution for the antecedent. Let us consider the three possibilities for the initial $\phi \underset{L}{\triangleright} \phi'$ to hold.

- $\sigma(\phi) = D \wedge \sigma(\phi') = [D, \ldots, D]$: We have $\sigma(\phi_1) = \sigma(\phi_2) = D$, $\sigma(\phi_1') = D$, and $\sigma(\phi_2') = [D, \ldots, D]$, so σ solves the consequent.
- $\sigma(\phi) = \sigma(\phi') = [\tau_1, \ldots, \tau_n]$: The *vcons* constraints force $\sigma(\phi_1) = \sigma(\phi_1') = \tau_1$ and $\sigma(\phi_2) = \sigma(\phi_2') = [\tau_2, \ldots, \tau_n]$. Obviously, σ solves the consequent.
- $\sigma(\phi) = \sigma(\phi') = D$: In this case we must have $\sigma(\phi_1) = \sigma(\phi_2) = \sigma(\phi_1') = \sigma(\phi_2') = D$ as well. Therefore, the constraints in the consequent are all solved by σ.

For the reverse direction we start with the possible solutions of $\phi_2 \underset{L}{\triangleright} \phi_2'$:

- $\sigma(\phi_2) = D \wedge \sigma(\phi_2') = [D, \ldots, D]$: If $\sigma(\phi_2) = D$ we have $\sigma(\phi) = \sigma(\phi_1) = D$ by dependency and *vcons*. By equality, we have $\sigma(\phi_1') = D$, too. $\sigma(\phi')$ cannot be D because $\sigma(\phi_2')$ is not D. Therefore, $\sigma(\phi') = [\sigma(\phi_1'), D, \ldots, D] = [D, \ldots, D]$.
- $\sigma(\phi_2) = \sigma(\phi_2') = [\tau_2, \ldots, \tau_n]$: Here, $\sigma(\phi)$ and $\sigma(\phi')$ cannot be D, hence $\sigma(\phi_1) = \sigma(\phi_1') =:$ τ_1 and $\sigma(\phi) = \sigma(\phi') = [\tau_1, \tau_2, \ldots, \tau_n]$ as required.
- $\sigma(\phi_2) = \sigma(\phi_2') = D$: The dependency constraints force $\sigma(\phi') = \sigma(\phi) = D$ and hence $\sigma(\phi_1) = \sigma(\phi_1') = D$ as well.

Therefore, the above rule is correct and complete as it neither adds nor removes solutions.

Normalization of an initial constraint set C is the exhaustive application $C \stackrel{*}{\Longrightarrow}_{\gamma} C^*$ of the normalization rules.

Theorem 2 *Constraint normalization terminates for an arbitrary initial constraint set C.*

Proof: For the constraint set C define the tuple (a, b, c, d) as follows:

- a is the sum of the number of function constraints, lift constraints, structure constraints, S-dependencies, M-dependencies, and L-dependencies.
- b is the sum of the number of base value dependencies, non-static dependencies, and memo constraints.
- c is the sum of the number of equality and dependency constraints.
- d is the sum of the number of $\rightarrow$ constraints and the number of clashes ($c \preceq \phi, c' \preceq \phi$ where the constraint constructors of c and c' differ) not resolved by $D = \phi$.

Each application of a rule decreases (a, b, c, d) in the lexicographic order, except rules (25) and (23). If we restrict the applicability of (23) to $S \preceq \phi_2 \notin C$ and apply rule (25) only once to each pair $\phi \underset{L}{\triangleright} \phi'$ we obtain termination.

The normalized constraint set C^* determines a substitution σ_{C^*} by ignoring all dependency constraints, interpreting the constraints of the form $\tau \preceq \phi$ as equalities, resolving cycles introduced through function and structure constraints by recursive types, and mapping the remaining variables to $\perp$.

Theorem 3 *σ_{C^*} is a minimal solution of C^*.*

Proof: By construction of the substitution σ_C^*.

From a solution of C^* we can easily construct a completion. We call the completion constructed from the minimal solution a *minimal completion*. The minimal

let $\{V_1', \ldots, V_m'\} = FV(E)$ **in**		
case E **of**		
V	$\check{\phi}_E = \phi_V$	
K	$S \preceq \check{\phi}_E$	
(if E_1 E_2 E_3)	$\check{\phi}_E = \hat{\phi}_{E_2}, \check{\phi}_E = \hat{\phi}_{E_3}, \hat{\phi}_{E_1} \rhd \check{\phi}_E$	
$(O\ E_1 \ldots E_n)$	$\langle[\hat{\phi}_{E_1}, \ldots, \hat{\phi}_{E_n}] \rightarrow_S \check{\phi}_E\rangle \preceq \phi,$	
	$\phi \underset{S}{\Rrightarrow} \check{\phi}_E, \phi \underset{S}{\Rrightarrow} \hat{\phi}_{E_1}, \ldots, \phi \underset{S}{\Rrightarrow} \hat{\phi}_{E_n}$	ϕ fresh
$(P\ E_1 \ldots E_n)$	$\langle[\hat{\phi}_{E_1}, \ldots, \hat{\phi}_{E_n}] \rightarrow \check{\phi}_E\rangle \preceq \phi_P$	
(lambda $(V_1 \ldots V_n)$ E_1)	$\langle[\phi_{V_1}, \ldots, \phi_{V_n}] \rightarrow \hat{\phi}_{E_1}\rangle \preceq \check{\phi}_E,$	
	$\check{\phi}_E \underset{S}{\Rrightarrow} \phi_{V_1}, \ldots, \check{\phi}_E \underset{S}{\Rrightarrow} \phi_{V_n},$	
	$\check{\phi}_E \underset{S}{\Rrightarrow} \phi_{V_1'}, \ldots, \check{\phi}_E \underset{S}{\Rrightarrow} \phi_{V_m'},$	
	$\check{\phi}_E \underset{M}{\Rrightarrow} \phi_{V_1'}, \ldots, \check{\phi}_E \underset{M}{\Rrightarrow} \phi_{V_m'}$	
$(E_0\ E_1 \ldots E_n)$	$\langle[\hat{\phi}_{E_1}, \ldots, \hat{\phi}_{E_n}] \rightarrow \check{\phi}_E\rangle \preceq \hat{\phi}_{E_0}$	
(lambda V E_1)	$\langle\phi_V \rightarrow \hat{\phi}_{E_1}\rangle \preceq \check{\phi}_E,$	
	$\check{\phi}_E \underset{S}{\Rrightarrow} \phi_V,$	
	$\check{\phi}_E \underset{S}{\Rrightarrow} \phi_{V_1'}, \ldots, \check{\phi}_E \underset{S}{\Rrightarrow} \phi_{V_m'}$	
	$\check{\phi}_E \underset{M}{\Rrightarrow} \phi_{V_1'}, \ldots, \check{\phi}_E \underset{M}{\Rrightarrow} \phi_{V_m'}$	
(apply E_1 E_2)	$\langle\phi_2 \rightarrow \check{\phi}_E\rangle \preceq \hat{\phi}_{E_1},$	
	$\phi_2 \underset{L}{\rhd} \hat{\phi}_{E_2}$	ϕ_2 fresh
(eval E_1)	$\hat{\phi}_{E_1} \rhd \check{\phi}_E, S \preceq \hat{\phi}_{E_1}, \phi \underset{S}{\Rrightarrow} \check{\phi}_E, \langle[\ ] \rightarrow_S \check{\phi}_E\rangle \preceq \phi$	ϕ fresh
(call/cc E_1)	$\langle[\phi_1] \rightarrow_S \check{\phi}_E\rangle \preceq \hat{\phi}_{E_1}, \langle\phi_2 \rightarrow_S \phi_3\rangle \preceq \phi_1, \check{\phi}_E \rhd \hat{\phi}_{E_1},$	ϕ_1, ϕ_2 fresh
	$\hat{\phi}_{E_1} \underset{S}{\Rrightarrow} \phi_1, \hat{\phi}_{E_1} \underset{S}{\Rrightarrow} \check{\phi}_E, \phi_1 \underset{S}{\Rrightarrow} \phi_2, \phi_1 \underset{S}{\Rrightarrow} \phi_3$	
(vcons E_1 E_2)	$vcons\ \hat{\phi}_{E_1}\ \hat{\phi}_{E_2} \preceq \check{\phi}_E, \check{\phi}_E \underset{M}{\Rrightarrow} \hat{\phi}_{E_1}, \check{\phi}_E \underset{M}{\Rrightarrow} \hat{\phi}_{E_2}$	
(vnil)	$vcons\ \phi_1\ \phi_2 \preceq \check{\phi}_E$	ϕ_1, ϕ_2 fresh
(vcar E_1)	$vcons\ \check{\phi}_E\ \phi_2 \preceq \hat{\phi}_{E_1}$	ϕ_2 fresh
(vcdr E_1)	$vcons\ \phi_1\ \check{\phi}_E \preceq \hat{\phi}_{E_1}$	ϕ_1 fresh
(vcons? E_1)	$vcons\ \phi_1\ \phi_2 \preceq \hat{\phi}_{E_1}, S \preceq \check{\phi}_E$	ϕ_1, ϕ_2 fresh
(vnil? E_1)	$vcons\ \phi_1\ \phi_2 \preceq \hat{\phi}_{E_1}, S \preceq \check{\phi}_E$	ϕ_1, ϕ_2 fresh
case D **of**		
(define $(P\ V_1 \ldots V_n)$ E)	$\langle[\phi_{V_1}, \ldots, \phi_{V_n}] \rightarrow \hat{\phi}_E\rangle \preceq \phi_P$	
(define P E)	$\hat{\phi}_E = \phi_P$	

Figure 15: Constraint generation

$$(1)\quad \frac{\langle \phi_1 \rightarrow_X \phi_2 \rangle \preceq \phi \quad D = \phi}{D = \phi \quad D = \phi_1 \quad D = \phi_2} \qquad (2)\quad \frac{\langle \phi_1 \rightarrow_X \phi_2 \rangle \preceq \phi \quad \phi \leadsto \phi'}{\langle \phi_1 \rightarrow_X \phi_2 \rangle \preceq \phi \quad \phi = \phi'}$$

$$(3)\quad \frac{\langle \phi_1 \rightarrow_X \phi_2 \rangle \preceq \phi \quad \mathit{vcons}\ \phi_1'\ \phi_2' \preceq \phi}{\langle \phi_1 \rightarrow_X \phi_2 \rangle \preceq \phi \quad \mathit{vcons}\ \phi_1'\ \phi_2' \preceq \phi \quad D = \phi} \qquad (4)\quad \frac{\langle \phi_1 \rightarrow_X \phi_2 \rangle \preceq \phi \quad S \preceq \phi}{\langle \phi_1 \rightarrow_X \phi_2 \rangle \preceq \phi \quad S \preceq \phi \quad D = \phi}$$

$$(5)\quad \frac{\langle \phi_1 \rightarrow_X \phi_2 \rangle \preceq \phi \quad \langle \phi_1' \rightarrow_X \phi_2' \rangle \preceq \phi}{\langle \phi_1 \rightarrow_X \phi_2 \rangle \preceq \phi \quad \phi_1 = \phi_1' \quad \phi_2 = \phi_2'}$$

$$(6)\quad \frac{\langle \phi_1 \rightarrow \phi_2 \rangle \preceq \phi \quad \langle \phi_1' \rightarrow_S \phi_2' \rangle \preceq \phi}{\langle \phi_1 \rightarrow_S \phi_2 \rangle \preceq \phi \quad \phi_1 = \phi_1' \quad \phi_2 = \phi_2'} \qquad (7)\quad \frac{\langle \phi_1 \rightarrow \phi_2 \rangle \preceq \phi \quad \langle \phi_1' \rightarrow_M \phi_2' \rangle \preceq \phi}{\langle \phi_1 \rightarrow_M \phi_2 \rangle \preceq \phi \quad \phi_1 = \phi_1' \quad \phi_2 = \phi_2'}$$

$$(8)\quad \frac{\langle \phi_1 \rightarrow \phi_2 \rangle \preceq \phi \quad \neg_S(\phi)}{\langle \phi_1 \rightarrow_M \phi_2 \rangle \preceq \phi}$$

$$(9)\quad \frac{\langle \phi_1 \rightarrow \phi_2 \rangle \preceq \phi \quad \mathit{memo}\ (\phi)}{\langle \phi_1 \rightarrow_M \phi_2 \rangle \preceq \phi \quad \mathit{memo}\ (\phi)}$$

$$(10)\quad \frac{\langle \phi_1 \rightarrow_S \phi_2 \rangle \preceq \phi \quad \langle \phi_1' \rightarrow_M \phi_2' \rangle \preceq \phi}{\langle \phi_1 \rightarrow_S \phi_2 \rangle \preceq \phi \quad \langle \phi_1' \rightarrow_M \phi_2' \rangle \preceq \phi \quad D = \phi}$$

$$(11)\quad \frac{\langle \phi_1 \rightarrow_S \phi_2 \rangle \preceq \phi \quad \neg_S \phi}{\langle \phi_1 \rightarrow_S \phi_2 \rangle \preceq \phi \quad D = \phi}$$

$$(12)\quad \frac{\langle \phi_1 \rightarrow_S \phi_2 \rangle \preceq \phi \quad \mathit{memo}\ (\phi)}{\langle \phi_1 \rightarrow_S \phi_2 \rangle \preceq \phi \quad D = \phi}$$

$$(13)\quad \frac{\langle \phi_1 \rightarrow_M \phi_2 \rangle \preceq \phi \quad \phi' \underset{S}{\Rightarrow} \phi}{\langle \phi_1 \rightarrow_M \phi_2 \rangle \preceq \phi \quad \neg_S(\phi')}$$

$$(14)\quad \frac{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad \mathit{vcons}\ \phi_1'\ \phi_2' \preceq \phi}{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad \phi_1 = \phi_1' \quad \phi_2 = \phi_2'} \qquad (15)\quad \frac{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad \phi \leadsto \phi'}{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad \phi = \phi'}$$

$$(16)\quad \frac{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad D = \phi}{D = \phi \quad D = \phi_1 \quad D = \phi_2} \qquad (17)\quad \frac{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad S \preceq \phi}{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad S \preceq \phi \quad D = \phi}$$

$$(18)\quad \frac{\phi' \underset{S}{\Rightarrow} \phi \quad D = \phi}{D = \phi \quad \neg_S(\phi')} \qquad (19)\quad \frac{\neg_S(\phi) \quad \phi' \underset{S}{\Rightarrow} \phi}{\neg_S(\phi) \quad \neg_S(\phi')}$$

$$(20)\quad \frac{\mathit{memo}\ (\phi) \quad \phi \underset{M}{\Rightarrow} \phi'}{\mathit{memo}\ (\phi) \quad \mathit{memo}\ (\phi')} \qquad (21)\quad \frac{\phi_1 \rhd \phi_2 \quad D = \phi_1}{D = \phi_1 \quad D = \phi_2}$$

$$(22)\quad \frac{\phi_1 \leadsto \phi_2 \quad D = \phi_1}{D = \phi_1 \quad D = \phi_2} \qquad (23)\quad \frac{\phi_1 \leadsto \phi_2 \quad S \preceq \phi_1}{\phi_1 \leadsto \phi_2 \quad S \preceq \phi_1 \quad S \preceq \phi_2}$$

$$(24)\quad \frac{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad \mathit{vcons}\ \phi_1'\ \phi_2' \preceq \phi' \quad \phi \underset{L}{\rhd} \phi'}{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad \mathit{vcons}\ \phi_1'\ \phi_2' \preceq \phi' \quad \phi_1 = \phi_1' \quad \phi' \rhd \phi \quad \phi_2' \rhd \phi' \quad \phi_2 \rhd \phi \quad \phi_2 \underset{L}{\rhd} \phi_2'}$$

$$(25)\quad \frac{D = \phi \quad \mathit{vcons}\ \phi_1'\ \phi_2' \preceq \phi' \quad \phi \underset{L}{\rhd} \phi'}{D = \phi \quad \mathit{vcons}\ \phi_1'\ \phi_2' \preceq \phi' \quad D = \phi_1' \quad \phi \underset{L}{\rhd} \phi_2'} \qquad (26)\quad \frac{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad D = \phi' \quad \phi \underset{L}{\rhd} \phi'}{\mathit{vcons}\ \phi_1\ \phi_2 \preceq \phi \quad D = \phi' \quad D = \phi}$$

Figure 16: Constraint normalization rules

completion is the result of the binding-time analysis which is passed on to the specializer.

Type reconstruction can be implemented by term graph manipulation of constraint terms similar to Henglein's implementation in quasi-linear time [18]. However, in our case the complexity is at least quadratic in the size of the program: The number of constraints generated is quadratic in the size of the program because we need to relate the free variables of a function to the function itself. The formal presentation of the algorithm and its complexity is beyond the scope of the present paper.