Implementing Memoization for Partial Evaluation

Peter Thiemann

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

Abstract. Memoization is a key ingredient in every partial evaluator. It enables folding by caching previously specialized functions. It is essential to make polyvariant specialization terminate. Its implementation is reasonably straightforward in a standard specializer that represents functions by closures. With the advent of handwritten program-generator generators (PGGs), implementing memoization gets harder, because PGGs use efficient standard representations of data at specialization time.

We present several implementations of memoization for PGGs that are able to deal with all features of current partial evaluators, specifically partially static data and functions. The first implementation is based on message passing. It is simple, portable, and efficient, but only suitable for untyped higher-order languages such as Scheme. The second implementation is geared towards typed language such as SML. Whereas the first two implementations are completely portable, our third implementation exploits introspective features that may not be available in all implementations.

Finally, we demonstrate that PGGs can solve the termination problem for partial evaluation. Our new incremental memoization algorithm performs incremental specialization and guarantees that specialization terminates whenever standard evaluation does.

Keywords: partial evaluation, automatic program transformation, incremental specialization, termination of partial evaluation, reflection.

Partial evaluation is a powerful program-specialization technique based on constant propagation. Given the *static* (known) parameters of a source program, partial evaluation constructs a *residual program*—a specialized version of the program, which on application to the remaining *dynamic* parameters produces the same result as the original program applied to all parameters. *Offline* partial evaluation [10,18] consists of two phases, binding-time analysis (BTA) and static reduction. BTA transforms a program and the binding times (static/dynamic) of the parameters into an annotated program. Subsequently, the specializer applies the program to the static input, reducing static expressions and rebuilding dynamic ones.

Specialization and Memoization Specialization is *polyvariant*: Each function in the source program may result in an arbitrary number of specialized versions in the residual program. The specializer must maintain a *specialization cache* to produce a finite residual program. The cache is a mapping from static parts of argument lists to residual function names. It works as follows:

Suppose the specializer calls function f on arguments $\overline{a}$. First, the specializer *extracts the static skeleton* $\overline{s}$ and *the dynamic parts* $\overline{d}$ (predetermined by the BTA) from $\overline{a}$. Then, the specializer checks whether $f_{\overline{s}}$ is present in the cache. If that is not the case, the specializer enters $f_{\overline{s}}$ into the cache, *creates a cloned version* $\overline{c}$ of $\overline{a}$, all dynamic parts replaced by fresh variables, and specializes f's body with respect to $\overline{c}$. In any case, the specializer generates a residual function call $f_{\overline{s}}(\overline{d})$.

Specialization Points Not every function call needs to be memoized. However, memoizing too many function calls leads to poor residual programs, whereas memoizing too few calls leads to nonterminating specialization. Nontermination is a problem, because specialization is more eager than standard evaluation. It processes both branches of dynamic conditionals and also the bodies of dynamic abstractions. This may introduce loops not present under standard evaluation. The most effective way to break these loops is to introduce new *sp-functions* for dynamic conditionals and abstractions [5, 7]. With this setup the specializer only memoizes calls to sp-functions and unfolds all others.

Partially Static Data Partially static data [27, 28] consists of a static skeleton and dynamic parts. For example, a (static) pair can have a static component and a dynamic one; the length of a list may be known statically, but the elements of the list may be dynamic; or a static function may have free variables with dynamic components. Partially static data increases the power of a partial evaluator considerably. It is a standard feature of current partial evaluators (e.g., [6, 9]).

Program-Generator Generators In the standard approach to partial evaluation, there is a specializer `spec`, a source program `p`, and some input data divided into a static part $\mathtt{inp}_s$ and a dynamic part $\mathtt{inp}_d$. The specification for `spec` is the *mix equation* [18, 19]:

$$[\![\mathrm{p}']\!]\ \mathtt{inp}_d = [\![\mathrm{p}]\!]\ \mathtt{inp}_s\ \mathtt{inp}_d \qquad \text{where} \qquad \mathrm{p}' = [\![\mathtt{spec}]\!]\ \mathtt{p}\ \mathtt{inp}_s \tag{1}$$

Double self-application of the specializer constructs a program-generator generator `cogen` [11, 37]. This `cogen` is a generator of program generators. Consider again the program `p` with inputs $\mathtt{inp}_s$ and $\mathtt{inp}_d$. Applying `cogen` to `p` yields a program `p-gen` with some interesting properties:

$$\begin{aligned} \mathtt{p\text{-}gen} &= [\![\mathtt{cogen}]\!]\ \mathtt{p} \\ \mathrm{p}' &= [\![\mathtt{p\text{-}gen}]\!]\ \mathtt{inp}_s \\ \mathtt{result} &= [\![\mathrm{p}']\!]\ \mathtt{inp}_d = [\![\mathrm{p}]\!]\ \mathtt{inp}_s\ \mathtt{inp}_d \end{aligned} \tag{2}$$

p-gen is a program generator specific to p. It accepts the static input of p and produces a specialized residual program p′. For this reason p-gen is called a *generating extension* of p. The interest in cogens and generating extensions is not just theoretical: Specialization with generating extensions is usually four to ten times faster than direct specialization [5, 27] [18, Chapter 4.10].

Hand-written Program-Generator Generators Holst and Launchbury [21] have discovered that cogen can be written by hand. We call such a hand-written program-generator generator **PGG**. PGGs solve the infamous encoding problem which hindered the development of efficient self-applicable specializers for typed languages; a PGG can be written in a different language, as long as it can parse and unparse programs; a PGG is not constrained by the limited capabilities of a specializer. The same is true for the generating extensions produced. They can freely exploit all features of their implementation language, not just those that are amenable to specialization. A final point in favor of PGGs is the simplification of correctness proofs. By now, there are several PGGs for functional languages: SML [3], Scheme [12], the lambda calculus [8, 23, 36].

1 Overview of This Work

The goal of PGGs in using the underlying language implementation to perform static computations conflicts with the necessity to provide an (intensional) equality for functions. Therefore, PGGs for functional languages [3, 12] have so far lacked a memoization mechanism for functional values. Section 2 describes a simple and efficient implementation of memoization for arbitrary partially static data, including functions. It is based on message passing and applies to dynamically typed higher-order programming languages (e.g., Scheme [17]). Unfortunately, this implementation cannot be used in a PGG for an ML-style typed language, because it is not well-typed. Therefore, Section 3 develops a memoization algorithm for typed higher-order languages. The key problem here is the development of a well-typed representation for closures.

Section 4 introduces an approach to memoization that exploits introspective features. The specializer reuses the closure representation of the underlying language implementation to perform memoization.

Section 5 shows that PGGs (in Scheme) are well suited to incremental specialization. An incremental specializer constructs the residual program on demand. It specializes a function only if the execution of the residual program actually calls its specialized version. Combined with a slight modification of the insertion of sp-functions, we can guarantee that specialization terminates for some input if and only if standard evaluation does so the same input.

Some knowledge of Scheme [17], ML [25], and partial evaluation [18] is a prerequisite to this paper.

```
(define (static-constructor ctor closed-value vvs bts)
  (let ((ctor-vvs (cons ctor vvs)))
     (lambda (what)
       (case what
         ((value)   (apply closed-value vvs))
         ((static)  (project-static ctor-vvs bts))
         ((dynamic) (project-dynamic ctor-vvs bts))
         ((clone)   (static-constructor
                     ctor
                     closed-value
                     (cdr (clone-dynamic ctor-vvs bts))
                     bts))
         (else      (error "bad argument ~a" what))))))
```

Fig. 1. Encoding for Partially Static Constructors

2 Memoization by Message Passing

For a dynamically typed language we can implement memoization for partially static data using a simple, portable, and efficient encoding as functions.

Recall that the specializer must perform four operations on every data object:

1. obtain its value (so that static computation can proceed),
2. extract its static parts (for caching),
3. extract its dynamic parts (to build lists of residual arguments and formal parameters), and
4. clone its dynamic parts by replacing them with fresh variables.

We represent all partially static data (be it functions or constructed data) in a generating extension by functions. The function `static-constructor` in Fig. 1 implements this representation (in Scheme). It factorizes the fixed static parts of a partially static value from its changing dynamic parts. Cloning changes the dynamic parts by replacing them with fresh variables. The fixed part `ctor` is the name of the constructor (or the unique label of the lambda) which is used for memoization. `closed-value` is the constructor function. For a partially static function f it is an abstraction which accepts the values of f's free variables and delivers the function as its result. The arguments of the constructor live separately in the list `vvs` which contains the arguments of the constructor (the values of the free variables in case of a function). With this separation there is no restriction regarding the "real" representation of partially static structures. They can be represented by vectors, functions, and so on. `vvs` is the argument list for the constructor. Finally, `bts` contains the binding times of the elements of `vvs`.

As an example, we demonstrate in Fig. 2 the encoding of `(lambda (x) y)` with unique label `cl42`, once with free static variable `y = 5` (left column) and then once with `y` a free dynamic variable (right column).

The projection functions, *project-static*, *project-dynamic*, and *clone-dynamic*, are well-known [18, Ch.10.2]. They extract the static part or the dynamic part

```
> (define consty
>   (static-constructor
>    'cl42
>    (lambda (y) (lambda (x) y))
>    (list y) '(STATIC))
> ; apply the function
> ((consty 'value) 'arg-for-x)
5
> ; get the static skeleton
> (consty 'static)
'(cl42 5)
> ; get the dynamic parts
> (consty 'dynamic)
'()
> ; clone the value
> (define cloned (consty 'clone))
> ; same static skeleton
> (cloned 'static)
'(cl42 5)
```

```
> (define consty
>   (static-constructor
>    'cl42
>    (lambda (y) (lambda (x) y))
>    (list 'y) '(DYNAMIC)))
>
> ((consty 'value) 'arg-for-x)
'y
>
> (consty 'static)
'(cl42)
>
> (consty 'dynamic)
'((y))
>
> (define cloned (consty 'clone))
>
> (cloned 'static)
'(cl42)
> (cloned 'dynamic)
'((clone-3))
```

Fig. 2. A Partially Static Function

```
(define (project-static value bts)
  (let ((ctor (car value)) (values (cdr value)))
    (cons ctor
          (let loop ((values values) (bts bts))
            (if (null? values)
                '()
                (let ((skeleton (loop (cdr values) (cdr bts))))
                  (if (static? (car bts))
                      (let ((static-value (car values)))
                        (if (procedure? static-value)
                            (append (static-value 'STATIC) skeleton)
                            (cons static-value skeleton)))
                      skeleton)))))))
```

Fig. 3. Implementation of *project-static*

from a partially static value, or they replace the dynamic parts of a value by fresh variables. What is new in our setting is that they may encounter static data encoded as a function in which case they must pass the respective message to it as outlined above. To convey the idea, we show the implementations of *project-static* in Fig. 3. We omit the other functions, as they pose no new problems.

3 Typed Memoization

Unfortunately the message-passing approach shown in Sec. 2 does not work in an ML-style typed language. The type of the abstraction `(lambda (what) ...)` in *static-constructor* (Fig. 1) depends on the value of `what`. This is not surprising as the function implements method dispatch in the representation of static constructors. The abstraction `(lambda (what) ...)` has no ML type, because the result of the `'value` message is a function and the result of the `'static` message is a list.

Hence, we have to perform some variant of closure conversion [30]. It is, however, well-known that closure conversion is not straightforward in a typed setting [26]. The untyped appraoch closure conversion introduces a datatype `AllClosures` which has a constructor for each abstraction in the program. The constructor takes the values of the abstraction's free variables. We replace abstractions with the appropriate closure of type `AllClosures` and each function application with a case dispatch on the closure. However, in a typed setting a type conflict arises unless all functions have the same type.

For concreteness we consider a monomorphic subset of Standard ML [25]. We concentrate on a monomorphic language because of typing problems with the specialization cache. This issue is usually side-stepped by monomorphic expansion.

Applying Closure Analysis When we assume the availability of the explicitly typed source program (after type reconstruction) and the results of a closure analysis (see e.g., [14, 33, 35]), we can solve the typing problem. Suppose each program point is marked with a unique label $\ell \in$ Labels. An (equational) flow analysis partitions Labels into disjoint *flow classes* $L_1 \cup \ldots \cup L_n$ such that values of expressions with label $\ell \in L_i$ are never mixed up with values of expressions with labels in L_j, for $i \neq j$. Moreover, unless the source program has a type error, all expressions in the same flow class have the same type.

We further assume that the membership in a flow class is made explicit in the types. That is, the type $\tau \to_i \tau'$ is the type of a function in flow class L_i. We represent such a function as a pair of type $(\mathtt{FC}_i \to \tau \to \tau') \times \mathtt{FC}_i$. That is, we abstract the free variables from the original functions and package them in some data type $\mathtt{FC}_i$.

The data type $\mathtt{FC}_i$ has constructors $\mathtt{FV}_\ell$ for each $\mathtt{fn}_\ell\ x$ `=>` E where $\ell \in L_i$ (in this section, we adopt ML syntax for expressions and types). The arguments of the constructor $\mathtt{FV}_\ell$ are the free variables of $\mathtt{fn}_\ell\ x$ `=>` E and their annotated types are $\tau_{\ell 1}, \ldots, \tau_{\ell m_\ell}$.

We replace

$$\overline{\mathtt{fn}_\ell\ x\ \mathtt{=>}\ E}^c = (\mathtt{fn}\ (\mathtt{FV}_\ell(x_1, \ldots, x_{m_\ell}))\ \mathtt{=>}\ \mathtt{fn}\ x\ \mathtt{=>}\ \overline{E}^c, \mathtt{FV}_\ell(x_1, \ldots, x_{m_\ell}))$$

where $x_1 \ldots x_n = FV(E)$ and we replace each application $(E_1\ E_2)$ in which E_1 has type $\tau \to_i \tau'$ with

$$\overline{(E_1\ E_2)}^c = \mathtt{let}\ (cv, fvs)\ \mathtt{=}\ \overline{E_1}^c\ \mathtt{in}\ ((cv\ fvs)\ \overline{E_2}^c)\ \mathtt{end}.$$

This program transformation induces a type transformation as follows.

$$\begin{aligned}
\overline{\tau_1 \rightarrow_i \tau_2}^c &= (\mathtt{FC}_i \rightarrow \overline{\tau}^c \rightarrow \overline{\tau'}^c) \times \mathtt{FC}_i \\
\overline{(\tau_1, \ldots, \tau_n)\chi}^c &= (\overline{\tau_1}^c, \ldots, \overline{\tau_n}^c)\chi
\end{aligned}$$

Here χ stands for an arbitrary type constructor different from $\rightarrow$. As the type transformation also applies to free variables of functions, we obtain the definition of $\mathtt{FC}_i$:

$\mathtt{datatype\ FC}_i\ \mathtt{=\ \ldots\ \mid\ FV}_\ell\ \mathtt{of}\ (\overline{\tau_{\ell 1}}^c\ \mathtt{*}\ \ldots\ \mathtt{*}\ \overline{\tau_{\ell m_\ell}}^c)\ \mathtt{\mid\ \ldots}$

The program transformation can be performed on a per-flow-class basis. The resulting program is well-typed, since all program points in L_i simultaneously change their type from $\tau \rightarrow_i \tau'$ to $(\mathtt{FC}_i \rightarrow \overline{\tau}^c \rightarrow \overline{\tau'}^c) \times \mathtt{FC}_i$. The transformed application is also well-typed: As $(E_1\ E_2)$ is well-typed, E_2 must have type τ and $(E_1\ E_2)$ must has type τ'. In the transformed expression, *cv* has type $\mathtt{FC}_i \rightarrow \overline{\tau}^c \rightarrow \overline{\tau'}^c$ and *fvs* has type $\mathtt{FC}_i$. Therefore, (*cv fvs*) has type $\overline{\tau}^c \rightarrow \overline{\tau'}^c$ and can be applied to $\overline{E_2}^c$ of type $\overline{\tau}^c$ to yield a result of type $\overline{\tau'}^c$.

The Static Skeleton To encode static skeletons we introduce a type $\mathtt{FCS}_i$ for each flow class L_i with one constructor $\mathtt{FVS}_\ell$ for each function in flow class i with an argument for each free variable of the function.

$\mathtt{datatype\ FCS}_i\ \mathtt{=\ \ldots\ \mid\ FVS}_\ell\ \mathtt{of}\ (\overline{\tau_{\ell 1}}^s\ \mathtt{*}\ \ldots\ \mathtt{*}\ \overline{\tau_{\ell m_\ell}}^s)\ \mathtt{\mid\ \ldots}$

The function $\overline{\tau}^s$ works on *binding-time annotated types* (as output by a BTA) where each type constructor is annotated with a binding time S or D (static or dynamic). It constructs the type of the static skeleton for a value of type τ. We use υ to denote an annotated type with its top level annotation stripped off.

$$\begin{aligned}
\overline{\upsilon^D}^s &= \mathtt{unit} \\
\overline{\tau_1 \rightarrow_i^S \tau_2}^s &= \mathtt{FCS}_i \\
\overline{(\tau_1, \ldots, \tau_n)\chi^S}^s &= (\overline{\tau_1}^s, \ldots, \overline{\tau_n}^s)\chi
\end{aligned}$$

In other words: dynamic values are mapped to the unit type, static functions are mapped to the type of the static skeleton of their free variables, other static data is mapped to itself with the subcomponents mapped recursively.

In the typed case, each memoized function has its own specialization cache [3]. Otherwise, the cache may not be well-typed, for its type depends on the static part of the arguments. We use type and flow information to construct the implementation of each dynamic call. Consider a call $f^D(E_1, \ldots, E_n)$ and assume that E_i has type τ_i. The static skeleton is a tuple $(s_1, \ldots, s_n)$ of type $\overline{\tau_1}^s \times \ldots \times \overline{\tau_n}^s$ with the components determined by $s_i = \mathit{get\_static}(E_i, \tau_i)$.

$$\mathit{get\_static}(E, \tau) = \begin{cases} () & \tau = \upsilon^D \\ E & \tau = B^S \\ \mathit{project\_static}_\ell(E) & \tau = \tau' \rightarrow_\ell^S \tau'' \end{cases}$$

That is, we return () for dynamic values, the value itself for static values of base type $B \in \{\texttt{unit}, \texttt{int}, \ldots\}$, and the static projection of the value for static functional values.

Using *get_static*, the definition of *project_static* is straightforward.

$$\begin{array}{l} \mathit{project\_static}_\ell(\mathit{cv}, \mathit{fvs}) = \mathit{case}\ \mathit{fvs}\ \mathit{of} \\ \quad \ldots \mid \mathtt{FV}_\ell(x_1, \ldots, x_{m_\ell}) \Rightarrow \mathtt{FVS}_\ell(\ldots, \mathit{get\_static}(x_k, \tau_{\ell k}), \ldots) \mid \ldots \end{array}$$

The PGG precomputes the function *get_static* as it is not ML-typable in a generating extension. However, the resulting $\mathit{project\_static}_\ell$ functions are typable, once the occurrences of *get_static* are expanded. Their type is

$$\mathit{project\_static}_i : (\mathtt{FC}_i \rightarrow \overline{\tau}^c \rightarrow \overline{\tau'}^c) \times \mathtt{FC}_i \rightarrow \mathtt{FCS}_i$$

where i ranges over all flow classes. The result type $\mathtt{FCS}_i$ of $\mathit{project\_static}_i$ is an equality type as all function types in the argument have been recursively replaced by their corresponding closure types. Hence, the generating extensions can use these values as keys in the specialization cache.

For each functional flow class L_i we specify a function $\mathit{project\_dynamic}_i$: $\mathtt{FC}_i \rightarrow \mathtt{Code}$ *list*.

$$\begin{array}{l} \mathit{project\_dynamic}_i(\mathit{cv}, \mathit{fvs}) = \mathit{case}\ \mathit{fvs}\ \mathit{of} \\ \quad \ldots \mid \mathtt{FV}_k(x_1, \ldots, x_{m_k}) \Rightarrow \mathit{get\_dynamic}(x_1, \tau_1)\mathit{get\_dynamic}(x_2, \tau_2) \ldots [\,] \mid \ldots \end{array}$$

Here, we assume the macro definition

$$\mathit{get\_dynamic}(w, \tau)l = \begin{cases} w :: l & \tau = v^D \\ l & \tau = B^S \\ \mathit{project\_dynamic}_i(w)@l & \tau = \tau' \rightarrow_i^S \tau'' \end{cases}$$

where $w :: l$ denotes construction of a list with head w and tail l, and @ denotes list concatenation. And finally, the specification of cloning.

$$\mathit{get\_clone}(w, \tau) = \begin{cases} \mathit{genVariable}() & \tau = v^D \\ w & \tau = B^S \\ \mathit{clone\_dynamic}_\ell(w) & \tau = \tau' \rightarrow_\ell^S \tau'' \end{cases}$$

$$\begin{array}{l} \mathit{clone\text{-}dynamic}_i : \mathtt{FC}_i \rightarrow \mathtt{FC}_i \\ \mathit{clone\_dynamic}_\ell(\mathit{cv}, \mathit{fvs}) = \mathit{case}\ \mathit{fvs}\ \mathit{of} \\ \quad \ldots \mid \mathtt{FV}_k(x_1, \ldots, x_{m_k}) \Rightarrow (\mathit{cv}, \mathtt{FV}_k(\ldots, \mathit{get\_clone}(x_i, \tau_i), \ldots)) \mid \ldots \end{array}$$

Failure of the Obvious The extra types $\mathtt{FCS}_i$ for the static skeletons appear artificial. After all, a separate type for the static skeleton is not necessary in the untyped case. Thus, let us consider using a parameterized type $\mathtt{FC}_i$, both for the value and for the static skeleton. This datatype has one parameter for each free variable of each lambda in the flow class L_i.

To see what happens, consider the following example ML program:

```
fun loop (n, g) = if n=0 then g else loop (n-1, fn y => g y)
fun f n = loop (n, fn z => z)
```

As both abstractions can appear as the parameter g of loop it is clear that they belong to the same flow class. Therefore, we define a datatype

```
datatype 'a FC = FV1 of unit | FV2 of 'a
```

encoding the free variables of the abstractions. The result after transformation:

```
val fnx = (fn (FV1 ()) => fn z => z, FV1 ())
fun fny g = (fn (FV2 g) => fn z => let val (cv, fvs) = g in cv fvs z end,
            FV2 g)

fun loop_m (n, g) = if n=0 then g else loop_m (n-1, fny g)
fun f_m n = loop_m (n, fnx)
```

Unfortunately, the transformed program does not type-check anymore. The functions `fnx` and `fny` have the following types:

```
val fnx : ('a FC -> 'b -> 'b) * 'c FC
val fny : 'a -> ((('b -> 'c -> 'd) * 'b) FC -> 'c -> 'd) * 'a FC
```

Function `fny` embeds the type of its first parameter in the type of its result. We have constructed `loop_m` in such a way that `fnx` as well as `fny fnx` and `fny (fny fnx)` (and so on recursively) can occur as parameters of `fny`. Therefore the type checker complains about a circular type:

```
Error: pattern and expression in val rec dec don't agree (circularity)
  pattern:    (((('Z -> 'Y -> 'X) * 'Z) FC -> 'Y -> 'X) * 'W FC)
  expression: 'W
  in declaration:
    loop_m = ...
```

For this reason we have to use two different data types, one to encode functional values and another for the static skeleton.

4 Memoization Using Introspection

Our function `static-constructor` constructs an explicit closure for a function although the underlying implementation represents the function as a closure anyway. If an implementation provides introspective operations to access the underlying representation we can avoid that overhead. Most implementations provide the necessary operations for debugging purposes or for use by the compiler:

- obtain the values of the free variables of a function,
- obtain the code part of a function, and
- construct a function from its code part and the values of its free variables.

However, the specializer must be able to reconstruct the binding times of the free variables. Therefore, each variable must be bound to a pair consisting of the variable's value and its binding time. We have no other hope of reconstructing the binding time, as the sequence of variables in the closure provided by the system is not predictable.

To be more specific, suppose the system provides the following functions:

– `reify-closure`, which accepts a procedure and returns a closure, and
– `reflect-closure`, which accepts a closure and returns a procedure.

In this context a closure is a record type providing the following operations:

– `(make-closure label vals)` creates a new closure with label `label`, which must be obtained from another closure using `closure->label`, and list of free variables `vals`,
– `(closure->label closure)` returns a unique label identifying the function represented by `closure`, for example the address of the function's code, and
– `(closure->vals closure)` returns the list of the values of the free variables of the function represented by the closure.

Next, we need operations to encode bindings:

– `(make-binding value btime)` pairs a value and a binding time,
– `(binding->value binding)` returns the value component, and
– `(binding->btime binding)` returns the binding time.

To support other partially static data structures, we suppose the following encoding of constructed data.

– `(make-constructor ctor args bts)` creates a data structure with constructor `ctor`, arguments `args`, and binding times `bts`,
– `(constructor->ctor cval)` returns the constructor of a constructed value,
– `(constructor->args cval)` returns the arguments of the constructor of a constructed value,
– `(constructor->bts cval)` returns the binding times of the arguments of the constructor of a constructed value, and
– `(constructor? obj)` returns `#f` unless the argument is a constructed data structure.

In order to make it work, we have to change *project-static* etc. In Fig. 3, we replace the body of the `(let ((static-value (car values))) ...)` with

```
(cond ((procedure? static-value)
       (append (project-static-proc static-value) skeleton))
      ((constructor? static-value)
       (append (project-static-ctor static-value) skeleton))
      (else
       (cons static-value skeleton)))
```

and define the procedures `project-static-proc` and `project-static-ctor`:

```
(define (project-static-proc proc)
  (let ((closure (reify-closure proc)))
    (let ((freevals (closure->vals closure)))
      (project-static (cons (closure->label closure)
                            (map binding->value freevals))
                      (map binding->btime freevals)))))
```

```
(define (project-static-ctor cval)
  (project-static (cons (constructor->ctor cval)
                        (constructor->args cval))
                  (constructor->bts cval)))
```

The possibility to exploit reflective features is a unique feature of PGGs. A `cogen` generated by self-application cannot utilize such a feature, as it is derived from a specializer which has its own implementation of environments.

The current version 0.44 of Scheme 48 [20] has exactly the features that we need. Specifically, the procedures

- `make-closure` accepts a template (which contains a unique identification, literals, and code) and an environment (a vector) to build a closure,
- `closure-template` extracts the template from a closure,
- `template-id` extracts the unique identification from a template, and
- `closure-env` extracts the environment from a closure.

This is all we need to get things to work.

5 Incremental Memoization

In Scheme with `eval` [29], generating extensions can perform incremental specialization just by changing the memoization code. The idea is that the specializer only processes code when there is a demand to run it. No time is spent specializing code which is never run. This is analogous to the difference between call-by-value and call-by-need: With call-by-value all arguments of functions are evaluated once, whereas with call-by-need arguments of functions are evaluated at most once. Therefore incremental specialization *terminates more often* than standard specialization.

The fine thing is that we do not have to change anything in our PGG except the treatment of dynamic function calls. Fig. 4 shows the new definition.

When we start the generating extension it performs a call to the goal function (see Fig. 4), which defines a top-level procedure which accepts the dynamic parameters and returns the text of a call to this procedure.

The procedure constructed by the initial call behaves as follows. When it is called with actual dynamic parameters, it first goes away to create the text of the specialized function, using the static parameters remembered from the initial call. Next, it overwrites its own definition with the definition of the specialized function and calls the specialized function just constructed with the dynamic parameters.

When the recursive specialization process reaches a memoized function call, the same mechanism as for the initial call comes into play. Thus, the specialized program is constructed interleaved with its own execution. The specialization of a procedure starts upon the first call to its specialized variant.

The code uses the top-level variable `*dummy-definition-buffer*` to construct a top-level function definition from the compiled abstraction `generate`. We first assign the function to `*dummy-definition-buffer*` and then bind that value to a new name.

```
(define (memo-call fn bts args)
  (let*
      ((full-pp   (cons fn args))
       (static-pp (project-static full-pp bts))
       (dynamics  (project-dynamic full-pp bts))
       (actuals   (apply append dynamics))
       (found
        (or (assoc static-pp *spec-cache*)
            (let* ((new-name    (gensym fn))
                   (cloned-pp   (clone-dynamic full-pp bts))
                   (new-formals (apply append
                                       (project-dynamic cloned-pp bts)))
                   (new-entry   (spec-cache! (cons static-pp new-name)))
                   (generate
                    (lambda actuals
                      (let* ((new-def
                              `(DEFINE (,new-name ,@new-formals)
                                 ,(apply
                                   (eval fn (interaction-environment))
                                   (cdr cloned-pp))))
                             (dummy (add-to-residual-program! new-def))
                             (dummy
                              (eval new-def (interaction-environment))))
                        (apply (eval new-name (interaction-environment))
                               actuals)))))
              (set! *dummy-definition-buffer* generate)
              (eval `(DEFINE ,new-name *DUMMY-DEFINITION-BUFFER*)
                    (interaction-environment))
              (cons static-pp new-name))))
       (res-name  (cdr found)))
    ;; generate call to fn with actual arguments
    `(,res-name ,@actuals)))
(define *dummy-definition-buffer*)
```

Fig. 4. Implementation of Memoized Function Calls for Incremental Specialization

Termination With a little change in the introduction of sp-functions incremental specialization terminates whenever standard evaluation does. We can get the correct behavior by moving the sp-functions to the *branches* of dynamic conditionals and to the *bodies* of dynamic abstractions. This way, we only specialize those branches and bodies that are actually used in the execution of the specialized program.

6 Related Work

Self-applicable partial evaluators [6, 9] also employ dual representations of functions to generate efficient program-generator generators (cogens) and generating extensions. This theme reappears in our closure representations which separates the function part from the environment part in the representation of functions.

Minamide, Morrisett, and Harper [26] propose existential types to type the environment part of closures. This is appropriate in their intermediate code setting, which supports existential types. Standard ML does not support them, though [25]. Moreover, existential types would not solve the memoization problem as the dynamic call must be able to *compare* two closures. If the closure representation is abstract, such a comparison is impossible.

The termination of partial evaluation is usually addressed by means of program analysis. Holst [15] and Glenstrup and Jones [13] have defined enhanced BTA algorithms to ensure termination. Andersen and Holst [1] have defined a termination analysis for higher-order specialization. Our current approach is orthogonal as it builds on incremental specialization.

Intricate call graph analyses have been developed to find a set of memoized calls which is as small as possible but still ensures terminating specialization [19, 31, 32]. Our algorithm to insert sp-functions is a variation of Bondorf and Danvy's algorithm [5, 7] which works satisfactory in practice.

The idea of specializing data types to avoid type problems has been used before, for example by Steele [34].

7 Conclusions

We have presented four contributions to the implementation of memoization for partial evaluation. We have developed implementations of memoization for partially static functions, for untyped and for typed languages. Furthermore, language implementations that supply introspective features support an implementation of memoization which reuses the internal representation of functions. Finally, we have developed a framework for doing incremental specialization with generating extensions. With a slight variation of the standard insertion of sp-functions, incremental specialization solves all termination problems associated with partial evaluation.

The latter two features can only be achieved in the PGG framework: The introspective implementation exploits the representation of functions by the implementation and incremental specialization relies on inerleaving the actual execution with the specialization.

Furthermore, all of these techniques work with multiple levels of binding times [12]. The untyped techniques are all implemented in our PGG [36].

References

1. Peter Holst Andersen and Carsten Kehler Holst. Termination analysis for offline partial evaluation of a higher order functional language. In Radhia Cousot, editor, *Proc. International Static Analysis Symposium, SAS'96*, Aachen, Germany, September 1996. Springer-Verlag. LNCS.

2. Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen, 1993.
3. Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In Manuel V. Hermenegildo and Jaan Penjam, editors, *Programming Languages, Implementations, Logics, and Programs (PLILP '94)*, volume 844 of *Lecture Notes in Computer Science*, pages 198–214, Madrid, Spain, September 1994. Springer-Verlag.
4. Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*, Amsterdam, 1988. North-Holland.
5. Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Programming*, 17:3–34, 1991.
6. Anders Bondorf. *Similix 5.0 Manual.* DIKU, University of Copenhagen, May 1993.
7. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Programming*, 16(2):151–195, 1991.
8. Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In Peter Sestoft and Harald Søndergaard, editors, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, pages 1–10, Orlando, Fla., June 1994. ACM.
9. Charles Consel. A tour of Schism. In David Schmidt, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, pages 134–154, Copenhagen, Denmark, June 1993. ACM Press.
10. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
11. Yoshihiko Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
12. Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In Doaitse Swierstra and Manuel Hermenegildo, editors, *Programming Languages, Implementations, Logics, and Programs (PLILP '95)*, volume 982 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, September 1995. Springer-Verlag.
13. Arne John Glenstrup and Neil D. Jones. Bta algorithms to ensure termination of offline partial evaluation. In *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, Novosibirsk, Russia, June 1996.
14. Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [16], pages 448–472. LNCS 523.
15. Carsten Kehler Holst. Finiteness analysis. In Hughes [16], pages 473–495. LNCS 523.
16. John Hughes, editor. *Functional Programming Languages and Computer Architecture*, Cambridge, MA, 1991. Springer-Verlag. LNCS 523.
17. Institute of Electrical and Electronic Engineers, Inc. IEEE standard for the Scheme programming language. IEEE Std 1178-1990, New York, NY, 1991.
18. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
19. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, 1985. Springer-Verlag. LNCS 202.

20. Richard A. Kelsey and Jonathan Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.
21. John Launchbury and Carsten Kehler Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.
22. Julia Lawall and Olivier Danvy. Continuation-based partial evaluation. In LFP 1994 [24], pages 227–238.
23. Julia Lawall and Olivier Danvy. Continuation-based partial evaluation. Extended version of [22] from ftp://ftp.daimi.aau.dk/pub/danvy/Papers/, January 1995.
24. *Proc. 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, USA, June 1994. ACM Press.
25. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
26. Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Fla., January 1996. ACM Press.
27. Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Bjørner et al. [4], pages 325–347.
28. Torben Æ. Mogensen. Separating binding times in language specifications. In *Proc. Functional Programming Languages and Computer Architecture 1989*, pages 14–25, London, GB, 1989.
29. Jonathan Rees. The Scheme of things: The June 1992 meeting. *Lisp Pointers*, V(4), October 1992.
30. John C. Reynolds. Definitional interpreters for higher-order programming. In *ACM Annual Conference*, pages 717–740, July 1972.
31. Peter Sestoft. The structure of a self-applicable partial evaluator. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, pages 236–256, Copenhagen, Denmark, October 1985. Springer-Verlag. LNCS 217.
32. Peter Sestoft. Automatic call unfolding in a partial evaluator. In Bjørner et al. [4], pages 485–506.
33. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991. Also technical report CMU-CS-91-145.
34. Guy L. Steele Jr. Building interpreters by composing monads. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 472–492, Portland, OG, January 1994. ACM Press.
35. Dan Stefanescu and Yuli Zhou. An equational framework for the flow analysis of higher-order functional programs. In LFP 1994 [24], pages 318–327.
36. Peter Thiemann. Cogen in six lines. In R. Kent Dybvig, editor, *Proc. International Conference on Functional Programming 1996*, pages 180–189, Philadelphia, PA, May 1996. ACM Press, New York.
37. Valentin F. Turchin. The use of metasystem transition in theorem proving and program optimization. In *International Conference on Automata, Languages, and Programming. Seventh ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 645–657, Noordwijkerhout, The Netherlands, 1980. Springer-Verlag.