Fragmental Specialization

Simon Helsen and Peter Thiemann

Universität Freiburg, Institut für Informatik
Georges-Köhler-Allee, D-79110 Freiburg, Germany
{helsen,thiemann}@informatik.uni-freiburg.de

Abstract. Traditional offline program specialization relies on staging of a computation into two levels, static and dynamic. We present a generalization thereof where the static input data of one level is split up in a potentially unknown number of separate fragments. These fragments arrive in an unspecified order and are specialized as they arrive. Potential applications of the technique include separate compilation via specialization, efficient implementation of reflective languages, and just-in-time compilation.
We have implemented our new specialization technique in the PGG specialization system [27] for Scheme. We have proved correctness for a first-order fragment.

1 Introduction

Partial evaluation [15, 5] is an established technique for speeding up a program by specializing it with respect to known parts of its input. One particularly attractive application of partial evaluation is compilation [8, 4, 16, 22, 21]. Compilation is achieved by specializing an interpretive definition of a programming language with respect to a program.

While this scheme is flawless in theory and workable in practice, it always requires the whole program to be present. Hence, it is not applicable to separate compilation. In contrast, real compilers can compile one module at a time. They rely on a linker to combine the compiled modules later on.

Unfortunately, traditional partial evaluators cannot handle this task satisfactorily. Only the work of Heldal and Hughes [12, 13] seriously tackles separate compilation of modular programs by partial evaluation. Their approach requires a complete redesign of the analysis and specialization phase. In consequence, their technique is difficult to understand and lacks a correctness proof.

In specialization terms, separate compilation amounts to repeatedly specializing an interpreter with respect to different compilation units (modules). The combination of the resulting specialized program fragments to a meaningful whole is left to a linker. Abstracting from the problem of separate compilation, the specializer should process the same program repeatedly with respect to different fragments of the static input. The specialized program is the combination of the resulting specialized program fragments. We make the static input fragments available via an index and use a special `access` construct to map an index to its corresponding fragment. Hence, the name *fragmental specialization*.

Fragmental specialization applies to the problem of separate compilation. In this context, the meaning of `access` is quite intuitive. It maps a module name (the index) to its program text (the data fragment), so it corresponds to loading the compilation unit or its interface description. An interpreter must `access` the program text of a compilation unit before it can start executing that code.

What is the meaning of `access` at specialization time? The main difference to the standard semantics, is that only *one* fragment of the indexed set is available to a run of the specializer. Furthermore, there is no control over the order in which the fragments of the indexed set are presented to the specializer.

Consider again separate compilation. Suppose that the specializer specializes an interpreter with respect to a module M (in other words, it compiles module M). If, during that specialization, the `access` construct requests the same module, M, it means that the interpreter is processing a *local* procedure call and compilation/specialization simply proceeds by generating that call. If `access` requests a different module, M', the interpreter is processing a non-local procedure call and then there are two possibilities. If the specializer has processed M' before, it generates a backwards call to the corresponding entry point of M'. In other words, the non-local procedure call is automatically linked to a previously compiled module. Otherwise, if M' is still to be processed, the specializer generates a forward call in the compiled code of M. At the same time, it makes a note of which parts of M' are to be processed when it is presented to the specializer. This technique is tantamount to back-patching, the difference is that the first *reference* determines the target address.

Specialization with respect to an indexed set of static input fragments blends well with standard partial evaluation technology. An extension of the specializer's memoization mechanism (a caching scheme which enables the specializer to share specialized code and to generate mutually recursive procedures, thus improving termination of specialization) is sufficient to implement it. The `access` construct is built into the specializer and controls the extended memoization mechanism to obtain the effect outlined above.

Contribution We have specified specialization with respect to an indexed set of static input fragments. It is implemented in the PGG system [27], an offline partial evaluator for Scheme [17]. Our technique can be used

- to compile modules in an arbitrary order;
- to compile mutually recursive modules; and
- to compile programs comprised of an *a priori* unknown number of modules, enabling dynamic loading of modules.

All these points improve on the results of Heldal and Hughes [12, 13].

We provide a correctness proof of the specializer consisting of the standard type soundness result for the (type-based) binding-time analysis and a proof that specialization preserves the semantics. The proof is formalized for a first-order fragment of the language but the techniques carry over to the full language. The particular difficulty of this proof lies in the necessity to formalize the memoization mechanism of a specializer and prove its correctness. This has not been done before in the literature.

Overview Section 2 introduces the basics of offline partial evaluation. Section 3 discusses separate compilation of modular programs, the main application of our specialization framework. Section 4 sets up the theoretical background for a first-order functional language. It culminates in a correctness proof. Section 5 discusses implementation issues Finally, we discuss related work and conclude.

The examples in Section 3 require some knowledge of Scheme.

2 Offline partial evaluation

Offline partial evaluation [15, 5] is a specialization technique that guides its specialization-time computation with the annotations generated by a *binding-time analysis*. The annotations are S for constructs that process *static* data, which is available at specialization time, and D for constructs that depend on *dynamic* data, which is only available at run-time. Specialization boils down to interpreting the annotated program, evaluating the static constructs and generating code for the dynamic ones. Since specialization performs reductions under dynamic control (it processes the body of a dynamic lambda abstraction and it processes both branches of a conditional if the condition is dynamic), specialization may not terminate even though the original program always terminates. To remedy this behavior and to allow for multiple specialized variants of the same source text, offline partial evaluators usually implement *function memoization* [15, 25], a caching scheme which enables the specializer to share specialized code and to generate mutually recursive procedures (polyvariant program point specialization). It also improves termination of specialization.

3 Separate compilation of modular programs

A simple framework for separate compilation for untyped languages considers a modular program as a set of program fragments, indexed by module names. Every program fragment defines names and references names. Every fragment *exports* some defined names, which are visible to other fragments, and every fragment may *import* some names from other fragments. Separate compilation means to compile single program fragments in isolation so that we obtain an executable after combining (i.e., linking) the compiled program fragments.

In the rest of this section, we demonstrate how specialization of an interpreter using our approach achieves separate compilation in this sense. In particular, we define a tiny language and show two example interpreters for it that achieve compilation via specialization. All specialized code in this section is generated with the PGG specialization system for Scheme [27] and the source files are provided with the distribution. In our approach, the binding-time properties of the interpreter determine to what extent separate compilation is possible and the two examples demonstrate the main possibilities.

Syntax of the register machine language:

$$
\begin{array}{lll}
\textit{module} & ::= & (\textit{block}^{+}) \\
\textit{block} & ::= & (\textit{label}\ .\ \textit{instr}^{*}) \\
\textit{instr} & ::= & (\texttt{Incr}\ \textit{reg}) \mid (\texttt{Decr}\ \textit{reg}) \mid (\texttt{Jz}\ \textit{reg}\ \textit{label}) \mid (\texttt{Jump}\ \textit{label})
\end{array}
$$

Example program:

```
mod1 : ((add . ((jz 1 copy)          mod2 : ((copy . ((jz 2 test)
                (decr 1)                                  (incr 1)
                (incr 0)                                  (decr 2)
                (jump add)))                              (jump copy)))
        (finis . ()))                         (test . ((jz 1 finis)
                                                          (jump add))))
```

Fig. 1. The register machine language

A source language and its interpreter Figure 1 defines the syntax of a register-machine language ([12, 13]) together with an example program. A module is a non-empty sequence of blocks. A block is a sequence of instructions starting with a label. An instruction either increments a register, decrements a register, performs a conditional jump to a label if some register is zero, or it performs an unconditional jump to a label. A target label is either local or implicitly imported from a different module. All defined labels are implicitly exported. This mechanism is similar to the rudimentary support for separate compilation in assemblers.

The example contains two modules, *mod1* and *mod2*, with mutual references to each other. In particular, label `copy` is referenced in *mod1* and defined in *mod2*, whereas `finis` and `add` are defined in *mod1* and referenced in *mod2*.

An interpreter defines the semantics of the language. Its first part (Fig. 2) deals with syntax dispatch and the execution of the arithmetic instructions. The other part is the jump processor, which may be replaced.

In Fig. 2, function `(exec jump instrs regs)` takes three parameters: a function to process jumps, `jump`, a list of instructions to execute, `instrs`, and a list of register values, `regs`. To process a `Jump` instruction, `exec` calls `(jump label regs)`, where `label` is the target label and `regs` is the list of registers.

Besides standard Scheme lists, the code uses a user-defined datatype `list` with some utility functions. This datatype enables partially static treatment in the specializer, in particular, arity raising the elements of the list of registers. The function `::` is the `list` constructor, `hd` and `tl` are the corresponding selectors (head and tail). The function `split` takes a number i and a list l. It splits the list into a prefix of length i and the rest of the list. The result is a pair of lists (also a partially static user-defined datatype) and `fst` and `snd` select the components of this pair. The `concat` function concatenates its list arguments.

The following two subsections concentrate on the `jump` processor. We exhibit two variants that implement different behavior.

```
(define (exec jump instrs regs)
  (let loop ((instrs instrs) (regs regs))
    (if (null? instrs)
        (hd regs)
        (let ((instr (car instrs)) (instrs (cdr instrs)))
          (case (car instr)
            ((Incr) (let* ((regno (cadr instr))
                           (xxx (split regno regs))
                           (regs1 (fst xxx))
                           (regs2 (snd xxx))
                           (reg (hd regs2)))
                      (loop instrs
                            (concat regs1 (:: (+ reg 1) (tl regs2))))))
            ((Decr) (let* ((regno (cadr instr))
                           (xxx (split regno regs))
                           (regs1 (fst xxx))
                           (regs2 (snd xxx))
                           (reg (hd regs2)))
                      (loop instrs
                            (concat regs1 (:: (- reg 1) (tl regs2))))))
            ((Jz) (let* ((regno (cadr instr))
                         (label (caddr instr))
                         (xxx (split regno regs))
                         (reg (hd (snd xxx))))
                    (if (zero? reg)
                        (jump label regs)
                        (loop instrs regs))))
            ((Jump) (let* ((label (cadr instr)))
                      (jump label regs)))
            (else (error "Illegal Instruction")))))))
```

Fig. 2. Dispatch and arithmetic part of the interpreter

```
(define (jump exported-names modname)
  (lambda (name args)
    (let* ((is-exported (assoc name exported-names))
           (name+mod (or is-exported (cons name modname))))
      (access (cdr name+mod)
       (lambda (this-modname this-mod)
         (let ((found (assoc name this-mod)))
           (if found
               (exec (jump exported-names this-modname) (cdr found) args)
               (error "Undefined name"))))))))

(define (main exported-names name nargs initial-args)
  (let ((args (clone nargs initial-args)))
    (let loop ((exports exported-names))
      (if (null? exports)
          (dyn-error "Unknown name")
          (let ((export (car exports)))
            (if (eqv? name (car export))
                ((jump exported-names (cdr export)) (car export) args)
                (loop (cdr exports)))))))))
```

Fig. 3. Jump processor with declarations

Static module structure The first approach relies on making as much information static as possible. A separate export declaration associates each globally visible label with the module in which it is defined. Each label that is not mentioned in the global declaration is considered local. Here is such a declaration:

```
((add . mod1) (finis . mod1) (copy . mod2)))
```

Since it is unknown which of the exported procedures of a program gets called, the `main` function of the interpreter has a `name` parameter with binding time dynamic. The interpreter tests the supplied `name` against the list of exported names to obtain the name of the module containing `name`'s definition. At specialization time, these tests create a dispatch to all functions declared in the list. This technique coincides with a standard binding-time improving transformation ("The Trick" [15]).

Figure 3 implements such a `jump` processor. The `main` procedure first copies the dynamic argument list into a partially-static list of register values of predetermined length, `nargs`. The `loop` procedure dispatches over the `exported-names`, an association list mapping a label to its defining module. Once the corresponding module is found, it calls `jump` on the static version of the name.

In `jump`, the parameters `exported-names`, `modname`, and `name` are all static. It first constructs the fully qualified name, `name+mod`, of the entry point. Then it loads the corresponding module and searches for the name locally. If successful, this search binds `found` to a pair of the `name` and the code referenced by it. Finally, it calls `exec` to run the code.

Compiled code from the initial specialization run

```
(define (main-1 x-2 x-1)
  (let*
     ((mlet-3 (car x-1))    (mlet-4 (cdr x-1))
      (mlet-5 (car mlet-4)) (mlet-6 (cdr mlet-4))
      (mlet-7 (car mlet-6)) (mlet-8 (cdr mlet-6)))
    (case x-2
      ((add)   (jump-2 mlet-3 mlet-5 mlet-7))
      ((finis) (jump-3 mlet-3 mlet-5 mlet-7))
      ((copy)  (jump-4 mlet-3 mlet-5 mlet-7))
      (else    (error "Unknown name")))))
```

Compiled code for *mod1*

```
(define (jump-2 mlet-3 mlet-2 mlet-1)
  (if (zero? mlet-2)
      (jump-4 mlet-3 mlet-2 mlet-1)
      (jump-2 (+ mlet-3 1) (- mlet-2 1) mlet-1)))
(define (jump-3 mlet-3 mlet-2 mlet-1) mlet-3)
```

Compiled code for *mod2*

```
(define (jump-5 mlet-3 mlet-2 mlet-1)
  (if (zero? mlet-2)
      (jump-3 mlet-3 mlet-2 mlet-1)
      (jump-2 mlet-3 mlet-2 mlet-1)))
(define (jump-4 mlet-3 mlet-2 mlet-1)
  (if (zero? mlet-1)
      (jump-5 mlet-3 mlet-2 mlet-1)
      (jump-4 mlet-3 (+ mlet-2 1) (- mlet-1 1))))
```

Fig. 4. Compiled code for static module structure

```
(define (jump mod+name args)
  (access (car mod+name)
   (lambda (this-modname this-mod)
     (let ((found (assoc (cdr mod+name) this-mod)))
       (if found
           (exec jump (cdr found) args)
           (error "Undefined name"))))))

(define (jump-initial modulename name args)
  (access modulename
   (lambda (mod-name mod-body)
     (let loop ((names (map (lambda (block) (car block)) mod-body)))
       (if (null? names) (dyn-error "unknown name")
           (let ((this-name (car names)))
             (if (eqv? name this-name)
                 (jump (cons mod-name this-name) args)
                 (loop (cdr names)))))))))

(define (main modulename name nargs initial-args)
  (let ((args (clone nargs initial-args)))
    (jump-initial modulename name args)))
```

Fig. 5. Jump processor without declarations

The procedure `(access` *`index process`*`)`, used to load the module's text, is built into the specializer. Its first parameter, *`index`*, is an index value (the name of a module) and its second parameter, *`process`*, is a procedure with two parameters. Once the indexed value is successfully accessed, *`process`* is called with the index value as the first parameter and the indexed value as the second parameter.

Each call to `access` builds a memoization point. Since all procedure calls, including local ones, are processed through the `jump` function, each block in the source program gives rise to a procedure in the specialized program. The only concession towards specialization in this code is the repeated lookup of the initial entry point in the export declaration.

Compilation with this interpreter yields satisfactory results. The specializer compiles modules in any order. It compiles all exported functions in every module, so that mutual recursion is possible. Each module is compiled at most once and the compiled code looks quite natural.

Figure 4 shows compiled code for the register-machine program from Figure 1. The parameter, `x-2`, of the specialized `main-1` procedure is the entry label and the parameter, `x-1`, contains the argument list. Arity raising of the argument list with respect to three registers yields the three parameters, `mlet-3`, `mlet-5`, and `mlet-7`, to the specialized variants of the `jump` function. The `(case x-2 ...)` construct implements the required dispatch.

The dispatch code seems inefficient but it executes only once, when the compiled program starts. All other calls, intra-module as well as cross-module ones, are compiled as direct jumps.

Dynamic module structure The only inconvenience of the previous interpreter is the specification of the association of labels to modules before a compilation can start. While this is not unreasonable in ordinary software development, it becomes problematic if this association is not statically known. For example, a program might compute the name of a module or it might receive it over the network. At this point, another feature of `(access` ***`index process`***`)` comes to bear. Previously, the ***`index`*** value was static but the specializer can deal with a dynamic ***`index`*** value, too. In the latter case, the specializer processes *any* index and corresponding fragment and calls the ***`process`*** function with that index value and the indexed fragment, which at that point become statically known to the specializer. In addition, the specializer reinstalls an identical dynamic `access` construct to wait for further indexes. This re-installation generates code that dispatches on the index value at run-time.

Figure 5 shows an implementation of the `jump` processor, which requires that all non-local calls must use qualified names of the form `(`***`module . label`***`)`. The `main` function calls `jump-initial` with all arguments dynamic. The `loop` function implements The Trick over the unknown `name` by scanning the labels from the current module. Thus, in the call to `jump` only `args` is dynamic.

The example source (see Fig. 1) changes accordingly: *mod1* calls `(mod2 . copy)` instead of `copy` and so on. Figure 6 shows the compiled code after these changes. The startup function initializes the register list and calls `jump-initial-2`. The dispatch is now distributed over the modules since the existence of a function name is only known upon arrival of a module. This also explains the code for the empty module, which is produced when no more modules arrive. As before, the run-time dispatch happens only once.

4 Theoretical background

For clarity, the specification of the specialization and the statement and proof of its correctness deal with a first-order functional Mini-language. We use, $\overline{x}$, for the sequence, $x_1, \ldots, x_n$, and similar shorthands apply to other entities.

A simple first-order language Figure 7 presents the first-order language. A program is a list of definitions followed by an expression. A definition is a triple of a function name, a list of formal parameters, and an expression. Expressions can be variables, constants, conditionals, primitive operations, and function calls. Function calls are special. Their first parameter always accesses the underlying indexed value.

The semantics is standard [18], except for the partial function ω. It is the index function for the underlying indexed value. It is a parameter of the program

The startup function

```
(define (main-1 x-3 x-2 x-1)
  (let* ((mlet-4 (car x-1))
         (mlet-5 (cdr x-1))
         (mlet-6 (car mlet-5))
         (mlet-7 (cdr mlet-5))
         (mlet-8 (car mlet-7))
         (mlet-9 (cdr mlet-7)))
    (jump-initial-2 x-3 x-2 mlet-4 mlet-6 mlet-8)))
```

Compiled code for *mod1*

```
(define (jump-3 mlet-3 mlet-2 mlet-1)
  (if (zero? mlet-2)
      (jump-6 mlet-3 mlet-2 mlet-1)
      (jump-3 (+ mlet-3 1) (- mlet-2 1) mlet-1)))
(define (jump-4 mlet-3 mlet-2 mlet-1) mlet-3)
(define (jump-initial-2 x-5 x-4 mlet-3 mlet-2 mlet-1)
  (if (eq? 'mod1 x-5)
      (case x-4
        ((add) (jump-3 mlet-3 mlet-2 mlet-1))
        ((finis) (jump-4 mlet-3 mlet-2 mlet-1))
        (else (error "unknown name")))
      (jump-initial-5 x-5 x-4 mlet-3 mlet-2 mlet-1)))
```

Compiled code for *mod2*

```
(define (jump-6 mlet-3 mlet-2 mlet-1)
  (if (zero? mlet-1)
      (jump-7 mlet-3 mlet-2 mlet-1)
      (jump-6 mlet-3 (+ mlet-2 1) (- mlet-1 1))))
(define (jump-7 mlet-3 mlet-2 mlet-1)
  (if (zero? mlet-2)
      (jump-4 mlet-3 mlet-2 mlet-1)
      (jump-3 mlet-3 mlet-2 mlet-1)))
(define (jump-initial-5 x-5 x-4 mlet-3 mlet-2 mlet-1)
  (if (eq? 'mod2 x-5)
      (case x-4
        ((copy) (jump-6 mlet-3 mlet-2 mlet-1))
        ((test) (jump-7 mlet-3 mlet-2 mlet-1))
        (else (error "unknown name")))
      (jump-initial-8 x-5 x-4 mlet-3 mlet-2 mlet-1)))
```

Compiled code for the empty module

```
(define (jump-initial-8 x-5 x-4 mlet-3 mlet-2 mlet-1)
  (error "invalid index"))
```

Fig. 6. Compiled code for dynamic module structure

Syntactic domains

$$
\begin{array}{rll}
\mathit{Ident} \ni x & & \text{denumerable set of identifiers} \\
\mathit{Const} \ni c & & \text{denumerable set of constant symbols} \\
\mathit{Op} \ni o & & \text{denumerable set of operator symbols} \\
\mathit{Fname} \ni f & & \text{denumerable set of function symbols} \\
\mathit{Prog} \ni p & ::= d; \ldots d; e & \\
\mathit{Def} \ni d & ::= f(x, \ldots, x) = e & \\
\mathit{Exp} \ni e & ::= x \mid c \mid \texttt{if}\ e\ e\ e \mid o(e \ldots e) \mid \texttt{access}\ f(e \ldots e) &
\end{array}
$$

Semantic domains

$$
\begin{array}{ll}
b \in \mathit{Base} & = \text{set of base values, as discrete CPO} \\
\psi \in \Psi & = \mathit{Ident} \to \mathit{Base} \\
\varphi \in \Phi & = \mathit{Ident} \to \mathit{Base}^{n+1} \to \mathit{Base}_\perp \\
\omega \in \Omega & = \mathit{Base} \hookrightarrow \mathit{Base}
\end{array}
$$

Ω is the pointed CPO of partial functions.
Semantic functions

$$
\begin{array}{lll}
\mathcal{C}[\![]\!] & : \mathit{Const} \to \mathit{Base} & \text{unspecified} \\
\mathcal{O}[\![]\!] & : \mathit{Op} \to \mathit{Base}^n \to \mathit{Base} & \text{unspecified} \\
\mathcal{E}[\![]\!] & : \mathit{Exp} \to \Psi \to \Phi \to \mathit{Base}_\perp & \\
\mathcal{P}[\![]\!] & : \mathit{Prog} \to \Psi \to \Omega \to \mathit{Base}_\perp &
\end{array}
$$

Semantic equations

$$
\begin{array}{ll}
\mathcal{E}[\![x]\!]\psi\varphi & = (\psi(x))_\perp \\
\mathcal{E}[\![c]\!]\psi\varphi & = \mathcal{C}[\![c]\!]_\perp \\
\mathcal{E}[\![\texttt{if}\ e_1\ e_2\ e_3]\!]\psi\varphi & = \mathit{let}\ b_\perp = \mathcal{E}[\![e_1]\!]\psi\varphi\ \mathit{in} \\
& \quad \mathit{if}\ b\ \mathit{then}\ \mathcal{E}[\![e_2]\!]\psi\varphi\ \mathit{else}\ \mathcal{E}[\![e_3]\!]\psi\varphi \\
\mathcal{E}[\![o(e_1 \ldots e_n)]\!]\psi\varphi & = \mathit{let}\ b_{1\perp} = \mathcal{E}[\![e_1]\!]\psi\varphi\ \mathit{in} \\
& \quad \ldots \\
& \quad \mathit{let}\ b_{n\perp} = \mathcal{E}[\![e_n]\!]\psi\varphi\ \mathit{in} \\
& \quad (\mathcal{O}[\![o]\!](b_1, \ldots, b_n))_\perp \\
\mathcal{E}[\![\texttt{access}\ f(e_0 \ldots e_n)]\!]\psi\varphi & = \mathit{let}\ b_{0\perp} = \mathcal{E}[\![e_0]\!]\psi\varphi\ \mathit{in} \\
& \quad \ldots \\
& \quad \mathit{let}\ b_{n\perp} = \mathcal{E}[\![e_n]\!]\psi\varphi\ \mathit{in} \\
& \quad \varphi(f)(b_0, \ldots, b_n)
\end{array}
$$

Semantics of a program

$$
\begin{array}{l}
\mathcal{P}[\![d_1; \ldots; d_k; e]\!]\psi\omega = \\
\mathcal{E}[\![e]\!]\psi(\mathit{fix}\ \lambda\varphi.[f \mapsto \lambda v_0, \overline{v}.\mathit{if}\ v_0 \in \mathit{dom}(\omega)\ \mathit{then}\ \mathcal{E}[\![e]\!][x_0 \mapsto \omega(v_0), \overline{x} \mapsto \overline{v}]\varphi\ \mathit{else}\ \perp \\
\qquad \mid f(x_0, \overline{x}) = e \in \{d_1, \ldots, d_k\}])
\end{array}
$$

Fig. 7. A first-order language and its semantics

Syntactic domains (additions)

$$
\begin{aligned}
\mathit{2Prog} \ni p' &::= d'; \ldots d'; e' \\
\mathit{2Def} \ni d' &::= f(x, \ldots, x) = e' \\
\mathit{2Exp} \ni e' &::= x \mid c \mid \texttt{lift}\ e' \mid \texttt{if}^{\beta}\ e'\ e'\ e' \mid o^{\beta}(e' \ldots e') \mid \texttt{access}\ f^{\beta}(e' \ldots e') \\
\mathit{BTime} \ni \beta &::= S \mid D \qquad \text{where } S < D
\end{aligned}
$$

Semantic domains

$$
\begin{aligned}
b \in \mathit{Base} &= \text{set of base values, as discrete CPO} \\
r \in \mathit{RExp} &\quad \text{residual expressions (see Fig. 11)} \\
\mathit{RDef} &\quad \text{residual definition (see Fig. 11)} \\
v \in \mathit{Val} &= \mathit{Base} \cup \mathit{RExp} \times H \\
\psi' \in \Psi' &= \mathit{Ident} \to \mathit{Val} \\
\eta \in H &= \mathit{Set}(\mathit{Fname} \times \mathit{Base}^{k+1}) \\
\pi \in \Pi &= \mathit{Set}(\mathit{RDef}) \\
\omega \in \Omega &= \mathit{Base} \hookrightarrow \mathit{Base}
\end{aligned}
$$

The sets H and Π are CPOs under the subset ordering.
Semantic functions

$$
\begin{aligned}
&\mathcal{C}[\![]\!] : \mathit{Const} \to \mathit{Base} && \text{unspecified} \\
&\mathcal{O}[\![]\!] : \mathit{Op} \to \mathit{Base}^n \to \mathit{Base} && \text{unspecified} \\
&\mathcal{S}[\![]\!] : \mathit{2Exp} \to \Psi' \to \mathit{Val}_\perp \\
&\mathcal{T}[\![]\!] : \mathit{2Prog} \to \Psi' \to \Omega \to (\Pi \times \mathit{RExp})_\perp
\end{aligned}
$$

Fig. 8. Two-level first-order language, syntax and semantics, part 1

semantics $\mathcal{P}[\![]\!]$ and is fed into the first parameter of every function. The result of a function call cannot be defined unless the indexing function ω is defined at v_0. In that case, the actual parameter is the *indexed value*, that is $\omega(v_0)$.

In the equations, *let* $x_\perp = v$ *in* v' stands for *case* v *of* $\perp \to \perp \mid x_\perp \to v'$.

A two-level first-order language Figures 8 and 9 specify syntax and semantics of a two-level variant of the first-order language. The grammar specifies the additional items with respect to Fig. 7. The interesting aspect is in the treatment of function calls in the specialized program. The result, *Val*, of specializing an expression is a (lifted) base value, if the expression is static, or a (lifted) pair of an expression and a set H. The set H models the *pending list*, which collects generated function calls that still need to be processed by the specializer. If the specialization of an expression returns the pair (r, η) then the specializer must provide specialized function definitions as prescribed by η. The final result of specialization is a specialized program π and a residual expression, r.

In the equations, underlining indicates primitive operations that generate code. For example, $\underline{b}$ is a constant expression with $\mathcal{C}[\![\underline{b}]\!] = b$. To avoid formalizing the generation of names for specialized variants of functions, we assume that there is an injective function $\lceil\cdot\rceil$ that maps a function name and a list of (static)

Semantic equations

$$
\begin{array}{ll}
\mathcal{S}[\![x]\!]\psi' & = \psi(x)_\perp \\
\mathcal{S}[\![c]\!]\psi' & = (\mathcal{C}[\![c]\!], [])_\perp \\
\mathcal{S}[\![\texttt{lift}\ e']\!]\psi' & = let\ b_\perp = \mathcal{S}[\![e']\!]\psi'\ \ in\ (\underline{b}, [])_\perp \\
\mathcal{S}[\![\texttt{if}^D\ e'_1\ e'_2\ e'_3]\!]\psi' & = let\ (r_1,\eta_1)_\perp = \mathcal{S}[\![e'_1]\!]\psi'\ \ in \\
 & \quad let\ (r_2,\eta_2)_\perp = \mathcal{S}[\![e'_2]\!]\psi'\ \ in \\
 & \quad let\ (r_3,\eta_3)_\perp = \mathcal{S}[\![e'_3]\!]\psi'\ \ in \\
 & \quad (\underline{\texttt{if}}\ r_1\ r_2\ r_3, \eta_1 \cup \eta_2 \cup \eta_3)_\perp \\
\mathcal{S}[\![o^D(e'_1 \ldots e'_n)]\!]\psi' & = let\ (r_1,\eta_1)_\perp = \mathcal{S}[\![e'_1]\!]\psi'\ \ in \\
 & \quad \ldots \\
 & \quad let\ (r_n,\eta_n)_\perp = \mathcal{S}[\![e'_n]\!]\psi'\ \ in \\
 & \quad (\underline{o}(r_1,\ldots,r_n), \eta_1 \cup \ldots \cup \eta_n)_\perp \\
\mathcal{S}[\![\texttt{access}\ f^S(e'_0, e'_1 \ldots e'_{k+u})]\!]\psi' & = let\ b_{0\perp} = \mathcal{S}[\![e'_0]\!]\psi'\ \ in \\
 & \quad \ldots \\
 & \quad let\ b_{k\perp} = \mathcal{S}[\![e'_k]\!]\psi'\ \ in \\
 & \quad let\ (r_1,\eta_1)_\perp = \mathcal{S}[\![e'_{k+1}]\!]\psi'\ \ in \\
 & \quad \ldots \\
 & \quad let\ (r_u,\eta_u)_\perp = \mathcal{S}[\![e'_{k+u}]\!]\psi'\ \ in \\
 & \quad (\underline{\lceil f, b_0, \ldots, b_k \rceil}(r_1,\ldots,r_u), \\
 & \quad\ \eta_1 \cup \ldots \cup \eta_u \cup \{\lceil f, b_0, \ldots, b_k \rceil\})_\perp \\
\mathcal{S}[\![\texttt{access}\ f^D(e'_0, e'_1 \ldots e'_{k+u})]\!]\psi' & = let\ (r_0,\eta_0)_\perp = \mathcal{S}[\![e'_0]\!]\psi'\ \ in \\
 & \quad let\ b_{1\perp} = \mathcal{S}[\![e'_1]\!]\psi'\ \ in \\
 & \quad \ldots \\
 & \quad let\ b_{k\perp} = \mathcal{S}[\![e'_k]\!]\psi'\ \ in \\
 & \quad let\ (r_1,\eta_1)_\perp = \mathcal{S}[\![e'_{k+1}]\!]\psi'\ \ in \\
 & \quad \ldots \\
 & \quad let\ (r_u,\eta_u)_\perp = \mathcal{S}[\![e'_{k+u}]\!]\psi'\ \ in \\
 & \quad\ (\underline{\lceil f, *, b_1, \ldots, b_k \rceil}(r_0, r_1,\ldots,r_u), \\
 & \quad\ \eta_1 \cup \ldots \cup \eta_u \cup \{\lceil f, *, b_1, \ldots, b_k \rceil\})_\perp
\end{array}
$$

Semantics of specialization
Let $d'_i \equiv f_i(x_{(i,0)}, \overline{x}_i) = e'_i$, for $1 \le i \le k$.

$$
\begin{array}{l}
\mathcal{T}[\![d'_1; \ldots; d'_k; e']\!]\psi'\omega \\
\quad = let\ (r_0,\eta_0)_\perp = \mathcal{S}[\![e']\!]\psi'\ \ in \\
\quad\quad letrec\ (\pi,\eta) = \\
\quad\quad\quad let\ X = \{(buildDef\{\lceil f_i, b_0, \ldots, b_k\rceil(x_{(i,k+1)}, \ldots, x_{(i,k+u)}) = r\}, \eta'') \\
\quad\quad\quad\quad \mid \lceil f_i, b_0, \ldots, b_k \rceil \in \eta \cup \eta_0,\ b_0 \in dom(\omega), \\
\quad\quad\quad\quad\ (r,\eta'')_\perp = \mathcal{S}[\![e'_i]\!]\,[x_{(i,0)} \mapsto \omega(b_0), x_{(i,1)} \mapsto b_1, \ldots, x_{(i,k)} \mapsto b_k, \\
\quad\quad\quad\quad\quad\quad\quad\quad x_{(i,k+1)} \mapsto (x_{(i,k+1)}, []), \ldots, x_{(i,k+u)} \mapsto (x_{(i,k+u)}, [])]\} \\
\quad\quad\quad \cup\{(buildDef\{\lceil f_i, *, b_1, \ldots, b_k\rceil(b_0, x_{(i,k+1)}, \ldots, x_{(i,k+u)}) = r\}, \eta'') \\
\quad\quad\quad\quad \mid \lceil f_i, *, b_1, \ldots, b_k \rceil \in \eta \cup \eta_0,\ b_0 \in dom(\omega), \\
\quad\quad\quad\quad\ (r,\eta')_\perp = \mathcal{S}[\![e'_i]\!]\,[x_{(i,0)} \mapsto \omega(b_0), x_{(i,1)} \mapsto b_1, \ldots, x_{(i,k)} \mapsto b_k, \\
\quad\quad\quad\quad\quad\quad\quad\quad x_{(i,k+1)} \mapsto (x_{(i,k+1)}, []), \ldots, x_{(i,k+u)} \mapsto (x_{(i,k+u)}, [])]\} \\
\quad\quad\quad in\ (Set(d' \mid (d',\eta'') \in X), \bigcup Set(\eta'' \mid (d',\eta'') \in X))
\end{array}
$$

Fig. 9. Two-level first-order language, syntax and semantics, part 2

base values to a function name. Since the language is first-order and deals only with base values, the original names can be reused without conflict.

To simplify the specification further, function arguments are grouped by their binding time. The parameters 1 through k are static, whereas the parameters $k+1$ through $k+u$ are dynamic. The special indexing parameter can be static or dynamic. It determines the binding time of the `access` construct.

Specialization is also standard up to the collection of the H component from dynamic results. Most static constructs are omitted since their specialization is identical to the standard semantics (Fig. 7). The interesting bit is the definition of the two versions of

$$\texttt{access}\ f^{\beta}(e'_0 e'_1 \ldots e'_k e'_{k+1} \ldots e'_{k+u})$$

Regardless of β, the specializer specializes all the parameter expressions, expecting results of the specified binding times. If e'_0 is static with value b_0, the specializer generates a function call to $\lceil f, b_0, b_1, \ldots, b_k \rceil$ where $b_1, \ldots, b_k$ are the values of the static parameters. The H component of the result is set up to contain the tuple $\lceil f, b_0, b_1, \ldots, b_k \rceil$. The fixpoint in function $\mathcal{T}[\![]\!]$ picks up this tuple and generates the corresponding specialized function, but only if the indexed value is available.

If e'_0 is dynamic, the specializer also generates a residual function call, but to $\lceil f, *, b_1, \ldots, b_k \rceil$, and notes this in the H component of the result. The symbol $*$ is a special, otherwise unused, base value, and is taken to stand for *any* module. The *letrec* in $\mathcal{T}[\![]\!]$ picks up this tuple. It generates a specialized function, which is defined by pattern matching on the index parameter. In contrast to the previous case, it does not pick up one static value for b_0 but it picks up *all possible* static values for $b_0 \in dom(\omega)$ at once. This behavior will turn out to be crucial in the proof of semantics preservation.

Specialization of fragments can be understood as repeating $\mathcal{T}[\![p']\!]\psi'\omega$ with ω drawn from an increasing sequence in Ω, as more and more indexed values are presented to the specializer. The implementation reflects this behaviour by increasing the memoization cache, but it only generates new code.

Binding-time analysis Figure 10 presents the rules for binding-time analysis for the two-level language. Besides function types for top-level functions, the language has binding-time annotated types ($\texttt{base}; \beta$). Function and variable environments are standard. There are typing judgments for expressions ($\Delta, \Gamma \vdash e' : \rho$: function environment Δ and variable environment Γ entails that e' has annotated type ρ), definitions ($\Delta \vdash_d d'$) and programs ($\Delta, \Gamma \vdash_p p'$) with the usual meanings. The binding-time rules are standard [15, 26], but the rule for `access` needs some attention. The binding-time of the call to f depends solely on the binding-time of the index e'_0. The first k parameters are static and the remaining are dynamic. The result of specializing a call to f, and the result type of its definition, are dynamic since the specializer never unfolds calls.

Annotated Types

Function Types	$\sigma ::= \rho \times \ldots \times \rho \to \rho$
Annotated Types	$\rho ::= (\tau; \beta) \qquad \tau ::= \mathtt{base}$
Function Environments	$\Delta ::= \emptyset \mid \Delta[f : \sigma]$
Variable Environments	$\Gamma ::= \emptyset \mid \Gamma[x : \rho]$

Typing rules

$$(var)\frac{\Gamma(x) = \rho}{\Delta, \Gamma \vdash x : \rho} \qquad (const)\frac{}{\Delta, \Gamma \vdash c : (\mathtt{base}; S)} \qquad (lift)\frac{\Delta, \Gamma \vdash e' : (\mathtt{base}; S)}{\Delta, \Gamma \vdash \mathtt{lift}\ e' : (\mathtt{base}; D)}$$

$$(if)\frac{\Delta, \Gamma \vdash e_1' : (\mathtt{base}; \beta) \quad \Delta, \Gamma \vdash e_2' : (\tau; \beta') \quad \Delta, \Gamma \vdash e_3' : (\tau; \beta') \quad \beta \leq \beta'}{\Delta, \Gamma \vdash \mathtt{if}^{\beta}\ e_1'\ e_2'\ e_3' : (\tau; \beta')}$$

$$(prim)\frac{\Delta, \Gamma \vdash e_1' : (\mathtt{base}; \beta) \quad \ldots \quad \Delta, \Gamma \vdash e_n' : (\mathtt{base}; \beta)}{\Delta, \Gamma \vdash o^{\beta}(e_1' \ldots e_n') : (\mathtt{base}; \beta)}$$

$$(call\text{-}access)\frac{\begin{array}{c}\Delta(f) = \rho_1 \times \ldots \times \rho_n \to (\tau; D) \quad \Delta, \Gamma \vdash e_0' : (\mathtt{base}; \beta) \\ (\forall 1 \leq i \leq k)\ \Delta, \Gamma \vdash e_i' : (\tau_i; S) \quad (\forall 1 \leq i \leq u)\ \Delta, \Gamma \vdash e_{k+i}' : (\tau_{k+i}; D)\end{array}}{\Delta, \Gamma \vdash \mathtt{access}\ f^{\beta}(e_0' e_1' \ldots e_k' e_{k+1}' \ldots e_{k+u}') : (\tau; D)}$$

$$(def)\frac{\Delta(f) = \rho_1 \times \ldots \times \rho_n \to (\tau; D) \quad \Delta, [x_i : \rho_i] \vdash e' : (\tau; D)}{\Delta \vdash_d f(x_1 \ldots x_n) = e'}$$

$$(prog)\frac{\Delta \vdash_d d_1' \quad \ldots \quad \Delta \vdash_d d_m' \quad \Delta, \Gamma \vdash e' : (\tau; D)}{\Delta, \Gamma \vdash_p d_1'; \ldots d_m'; e'}$$

Fig. 10. Binding-time analysis for the first-order language

The target language The target language (Fig. 11) differs from the source language in that it has ordinary functions instead of `access` procedures. The semantic equations for $\mathcal{E}'[\![]\!]$ are identical to the ones for $\mathcal{E}[\![]\!]$ up to the rule for function call. There are two kinds of function definitions. The g-style definitions perform a pattern match on a base value in the first argument. They originate from the specialization of an `access` call with a dynamic index parameter. The remaining f-style definitions are standard. They originate from the specialization of an `access` call with a static index parameter.

Correctness The correctness of specialization has two facets. The first facet is a type preservation result for the binding-time analysis. We need a few auxiliary definitions.

Definition 1. *Let $\psi \in \Psi$ and $\psi'' \in \Psi'$.*

– $Val^S = Base$ *and* $Val^D = RExp \times H$;

Syntactic domains

$$\begin{array}{rl} RExp \ni r & ::= x \mid c \mid \texttt{if}\ r\ r\ r \mid o(r\ldots r) \mid f(r\ldots r) \\ RDef \ni d'' & ::= f(x,\ldots,x) = r \\ RProg \ni p'' & ::= (\Pi; r) \end{array}$$

Semantic domains

$$\varphi' \in \Phi' = Ident \to Base^* \to Base_\perp$$

Semantic functions

$$\begin{array}{l} \mathcal{E}'[\![]\!] : RExp \to \Psi \to \Phi' \to Base_\perp \\ \mathcal{P}'[\![]\!] : RProg \to \Psi \to Base_\perp \end{array}$$

Semantic equations

$$\begin{array}{rl} \mathcal{E}'[\![f(r_1 \ldots r_n)]\!]\psi\varphi' = & let\ b_{1\perp} = \mathcal{E}'[\![r_1]\!]\psi\varphi'\ in\ \ldots \\ & let\ b_{n\perp} = \mathcal{E}'[\![r_n]\!]\psi\varphi'\ in\ \varphi'(f)(b_1,\ldots,b_n) \end{array}$$

Semantics of a program

$$\begin{array}{l} \mathcal{P}'[\![(\pi;r)]\!]\psi = \mathcal{E}'[\![r]\!]\psi(\mathit{fix}\ \lambda\varphi'.[\ f \mapsto \lambda\overline{v}.\mathcal{E}[\![r]\!][\overline{x} \mapsto \overline{v}]\varphi' \mid f(\overline{x}) = r \in \pi] \cup \\ \qquad [\ g \mapsto \lambda v_0\overline{v}.if\ v_0 = b_1\ then\ \mathcal{E}[\![r_1]\!][\overline{x} \mapsto \overline{v}]\varphi'\ else\ \ldots \\ \qquad\qquad else\ if\ v_0 = b_n\ then\ \mathcal{E}[\![r_n]\!][\overline{x} \mapsto \overline{v}]\varphi'\ else\ \perp \\ \qquad \mid g(b_1\overline{x}) = r_1, \ldots, g(b_n\overline{x}) = r_n \in \pi]) \end{array}$$

Fig. 11. Syntax and semantics of the target language

- $\Gamma \models \psi'$ *iff, for all* $x : (\tau;\beta) \in \Gamma$, $\psi'(x) \in Val^\beta$;
- $\Gamma \models \psi \sim \psi'$ *iff,* $\Gamma \models \psi'$, $dom(\psi) = dom(\psi')$, *and for all* $x : (\tau;\beta) \in \Gamma$, *if* $\beta = S$ *then* $\psi'(x) = \psi(x)$ *and if* $\beta = D$ *then* $\psi'(x) = (x, [])$.

Theorem 1. (Type preservation)
Suppose $\Delta, \Gamma \vdash_p p'$ *and* $\Gamma \models \psi'$. *Then* $\mathcal{T}[\![p']\!]\psi'\omega \in {Val^D}_\perp$.

The proof of this theorem rests on the following lemma for expressions, which is standard and straightforward to prove. Specialization of an expression always terminates because there is no call unfolding.

Lemma 1. *Suppose* $\Delta, \Gamma \vdash e' :\ (\tau;\beta)$ *and* $\Gamma \models \psi'$.
Then $\mathcal{S}[\![e']\!]\psi' \in {Val^\beta}_\perp \setminus \{\perp\}$.

Given type preservation, we can show semantics preservation. The function $|\cdot|$ is the erasure function that maps two-level entities (expressions, definitions, programs, types, etc) to the underlying standard entities. It erases all annotations and `lift` constructs.

Theorem 2. (Semantics preservation) *Let* p' *be an annotated program with* $\Delta, \Gamma \vdash_p p'$. *Let* $\psi \in \Psi$ *and* $\psi' \in \Psi'$ *so that* $\Gamma \models \psi \sim \psi'$.
If $\mathcal{P}[\![|p'|]\!]\psi\omega = v$ *and* $\mathcal{T}[\![p']\!]\psi'\omega = (\pi, r)_\perp$ *then* $\mathcal{P}'[\![\pi; r]\!]\psi = v$.

Proving this theorem requires a lemma that states the appropriate result for expressions. The lemma is proved by induction on the two-level expression e'.

Lemma 2. *Let e' be an annotated expression with $\Delta, \Gamma \vdash e' : \rho$. Let $\psi \in \Psi$ and $\psi' \in \Psi'$ so that $\Gamma \models \psi \sim \psi'$. Let $\varphi \in \Phi$.*

Suppose $\mathcal{E}[\![|e'|]\!]\psi\varphi\omega = y$ and $\mathcal{S}[\![e']\!]\psi' = v'_\perp$.

- *If $v' = (r, \eta)$ and for all $\varphi' \in \Phi'$ such that, for all $\lceil f, b_0, \dots, b_k \rceil \in \eta$ and $b_{k+1}, \dots, b_{k+u} \in Base^*$, it holds that*

$$\varphi(f)(b_0, \dots, b_{k+u}) = \varphi'(\lceil f, b_0, \dots, b_k \rceil)(b_{k+1}, \dots, b_{k+u})$$

  and, for all $\lceil f, *, b_1, \dots, b_k \rceil \in \eta$ and $b_0, b_{k+1}, \dots, b_{k+u} \in Base^*$, it holds that

$$\varphi(f)(b_0, \dots, b_{k+u}) = \varphi'(\lceil f, *, b_1, \dots, b_k \rceil)(b_0, b_{k+1}, \dots, b_{k+u})$$

  then $\mathcal{E}'[\![r]\!]\psi\varphi' = y$.
- *If $v' = b$ then $y = b_\perp$.*

The proof of the main theorem proceeds by fixpoint induction.

5 Implementation

We have implemented specialization with respect to an indexed set of static values in the latest release of the PGG specialization system for Scheme [27]. In the implementation, a lot of the code for dealing with `access` can be shared with the standard memoization code. Type signatures, a standard feature of PGG, are used internally to impose the proper types on the arguments of `access`.

Furthermore, the implementation can process the fragments of static input in two modes. Either there is a separate session with the specializer for each group of fragments as it arrives, or all fragments arrive during a single session. To optimize the latter case, the implementation additionally puts a function value in every entry of the memoization table. This function is the second parameter of the `access` construct. It takes the index and the corresponding value as parameters and delivers the specialized body of the function determined by the entry.

The `access` construct is implemented in terms of another primitive, `_access`, which behaves non-deterministically under the standard semantics. It is simpler to implement because –when its first argument is dynamic– it always grabs the next indexed value that arrives and calls the *`process`* procedure with the static index and the value. It is easy to implement the better behaved `access` in terms of `_access` with a macro definition.

```
(define-syntax access
  (syntax-rules ()
    ((access index f)
     (let ((dyn-index index))
       (letrec
         ((jump
           (lambda ()
             (_access
              dyn-index
```

```
      (lambda (static-index value)
        (if (null? value)
            (dyn-error "invalid index")
            (if (eq? dyn-index static-index)
                (f static-index value)
                (jump))))))))
  (jump))))))
```

The macro implements "The Trick" over the indexes. It cannot be implemented as a function with PGG's monovariant analyses because every use of `access` might have a different type and different binding times.

6 Related work

Heldal and Hughes [12, 13] have developed a partial evaluator specifically tailored towards compiling modular languages via specialization. While they develop much of the technology from scratch, we reuse standard solutions from an existing specializer as much as possible. Their framework requires that module dependencies be acyclic, whereas our approach does not place such restrictions: as demonstrated in Section 3, the specializer compiles mutually dependent modules in arbitrary order. Their partial evaluator employs a third binding time, *late static*, to generate a dispatch function in an interface file. In our approach, the memoization cache plays this role. We substitute a lookup in the memoization cache for executing their late static code from the interface file. The memoization cache can be kept in memory between specializations or it can be dumped to a file and resurrected when specialization continues. Otherwise, our approach reuses existing specialization techniques.

One could argue that a dynamic `access` is late static, but we actually obtain *late staticness* by having `access` itself static (it is specialized away) while its first argument, the index, is dynamic.

The third binding time in Heldal and Hughes's specializer can be used to transmit additional "link-time" data between modules. It is not clear how to transmit such data in our approach. However, if module signatures contain type information on imports and exports it is possible to encode type information so that type-safe linking is ensured.

Sundaresh and Hudak [23, 24] describe a framework to construct *incremental programs*. Such programs can update the result of a computation efficiently if only a small part of the input changes. They obtain this effect by fixing a decomposition of the input, specializing the program with respect to each part of the input separately, and combining the results with a special linking combinator. Although this technique shares some similarities with ours, there are two fundamental differences. First, the input is split up in a fixed number of parts which have a fixed position relative to each other. In contrast, our approach allows an unbounded number of parts to be specialized separately and in any order. Second, Sundaresh's and Hudak's system is biased towards incremental changes of individual parts. Specialization of every part happens in isolation and the linking combinator recombines the specialized parts to one monolithic program. In our approach the result is a list of compiled parts and we cannot specialize in complete isolation.

A quite different problem is the specialization of modular programs as considered by Dussart et al. [6]. It considers a *subject program* divided into modules, and the authors argue that polymorphic binding-time analysis combined with the cogen approach to program specialization is the right tool for this endeavor. In contrast, we consider applications where the *static data is divided* into several parts. As an application we consider the *compilation of modular programs* by repeated specialization of the same interpreter with respect to one module at a time.

Glück and Jørgensen [9–11] have studied multi-level specialization. It can deal with any finite number of specialization stages. Every specialization round, one level is specialized away to produce a newly annotated program. In our work, we only maintain two binding times and all fragments have the same binding time. It might be possible to differentiate between flavors of staticness, e.g., "static now" and "static after access", as a further refinement. Moreover, in multi-level specialization, the number of stages is fixed before specialization starts. In contrast, the number of fragments in our approach need not be known *a priori.*

7 Conclusion

We have presented specialization with respect to an indexed set of static input fragments. It is sufficiently expressive to perform separate compilation for simple modular languages via specialization of an interpreter. Our technique has been integrated with the PGG specialization system and it opens interesting new applications. Our correctness proof is interesting in its own right because it formalizes the function memoization mechanism, which is usually left out of formal treatments.

In combination with a specializer that delivers directly executable specialized code [22], our technique can be used for just-in-time compilation of mobile code. In this scenario, modules in a higher-level language arrive, for example, from the network. Upon arrival, they trigger their own compilation, that is, specialization of the language's interpreter with respect to the module's text. The compiled code is directly linked with the rest of the program and thus immediately executable. This work is currently in progress.

Acknowledgments We thank Olivier Danvy, John Hughes, Michael Sperber and the anonymous referees for their helpful comments on earlier drafts of this paper.

References

1. Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation — for a better understanding of reflective languages. *Lisp and Symbolic Computation*, 9(2/3):203–241, May/June 1996.

2. Luca Cardelli. Program fragments, linking, and modularization. In Neil D. Jones, editor, *Proc. 24th Annual ACM Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, January 1997. ACM Press.
3. Charles Consel. Polyvariant binding-time analysis for applicative languages. In David Schmidt, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.
4. Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In POPL1991 [20], pages 14–24.
5. 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.
6. Dirk Dussart, Rogardt Heldal, and John Hughes. Module-sensitive program specialisation. In PLDI1997 [19], pages 206–214.
7. Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Proc. International Static Analysis Symposium, SAS'95*, volume 983 of *Lecture Notes in Computer Science*, pages 118–136, Glasgow, Scotland, September 1995. Springer-Verlag.
8. Yoshihiko Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
9. Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In Doaitse Swierstra and Manuel Hermenegildo, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278, Utrecht, The Netherlands, September 1995. Springer-Verlag.
10. Robert Glück and Jesper Jørgensen. Fast multi-level binding-time analysis for multiple program specialization. In *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, Novosibirsk, Russia, June 1996. Springer-Verlag.
11. Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, July 1997.
12. Rogardt Heldal and John Hughes. Partial evaluation and separate compilation. In Charles Consel, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*, pages 1–11, Amsterdam, The Netherlands, June 1997. ACM Press.
13. Rogardt Heldal and John Hughes. Extending a partial evaluator which supports separate compilation. *Theoretical Computer Science*, to appear.
14. Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.
15. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
16. Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 258–268, Albuquerque, New Mexico, January 1992. ACM Press.
17. Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105,

1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available electronically as http://www.neci.nj.nec.com/homepages/kelsey/r5rs.ps.gz.
18. Peter D. Mosses. *Denotational Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11. Elsevier Science Publishers, Amsterdam, 1990.
19. *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, USA, June 1997. ACM Press.
20. *18th Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 1991. ACM Press.
21. Michael Sperber and Peter Thiemann. Realistic compilation by partial evaluation. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 206–214, Philadelphia, PA, USA, May 1996. ACM Press.
22. Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In PLDI1997 [19], pages 215–225.
23. Raman S. Sundaresh. Building incremental programs using partial evaluation. In Paul Hudak and Neil D. Jones, editors, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, pages 83–93, New Haven, CT, June 1991. ACM. SIGPLAN Notices 26(9).
24. Raman S. Sundaresh and Paul Hudak. Incremental computation via partial evaluation. In POPL1991 [20], pages 1–13.
25. Peter Thiemann. Implementing memoization for partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, volume 1140 of *Lecture Notes in Computer Science*, pages 198–212, Aachen, Germany, September 1996. Springer-Verlag.
26. Peter Thiemann. A unified framework for binding-time analysis. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 742–756, Lille, France, April 1997. Springer-Verlag.
27. Peter Thiemann. *The PGG System—User Manual.* Universität Freiburg, Freiburg, Germany, March 2000. Available from http://www.informatik.uni-freiburg.de/proglang/software/pgg/.