Science of Computer Programming 17 (1991) 3–34
Elsevier

Automatic autoprojection of higher order recursive equations

Anders Bondorf*

DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark

Communicated by N.D. Jones
Revised March 1991

Abstract

Bondorf, A., Automatic autoprojection of higher order recursive equations, Science of Computer Programming 17 (1991) 3–34.

Autoprojection, or self-applicable partial evaluation, has been implemented for first order functional languages for some years now. This paper describes an approach to treat a *higher order* subset of the Scheme language. The system has been implemented as an extension to the existing autoprojector *Similix* [6, 7] that treats a first order Scheme subset. To our knowledge, our system is the first fully automatic and implemented autoprojector for a higher order language.

We describe a new automatic binding time analysis for higher order programs. The analysis requires no type information. It is based on a *closure analysis* [34], which for any application point finds the set of lambda abstractions that can possibly be applied at that point. The binding time analysis has the interesting property that no structured binding time values are needed.

Since our language is higher order, interpreters written in a higher order style can be partially evaluated. To exemplify this, we present and partially evaluate four versions of an interpreter for a lambda calculus language: one written in direct style, one written in continuation passing style, one implementing call-by-name reduction, and one implementing call-by-need reduction. The three latter interpreters are heavily based on higher order functions. To our knowledge, this is the first time autoprojection has been used to generate compilers from so sophisticated interpreters.

This paper is a modified and slightly extended version of [2].

1. Introduction

Partial evaluation is a program transformation that *specializes* programs: given a source program and a part of its input (the *static* input), a partial evaluator generates a *residual* program. When applied to the remaining input (the *dynamic* input), the residual program yields the same result as the source program would when applied

* This work was partly supported by ESPRIT Basic Research Actions project 3124 "Semantique" and partly by the Danish Research Academy. A part of the work was carried out during a stay at Lehrstuhl Informatk V, University of Dortmund, Germany.

to all of the input. *Autoprojection* is a synonym for *self-applicable* partial evaulation, that is, specialization of the partial evaluator itself. It was established in the seventies that autoprojection can be used for automatic semantics directed compiler generation: specializing a partial evaluator with static input being (the text of) an *interpreter* for some programming language *S* yields a *compiler* for *S* [16, 15, 37]. Specializing the partial evaluator with static input being (the text of) the partial evaluator itself even yields a compiler generator (automatic compiler generator generation!).

The first successfully implemented autoprojector was *Mix* [22]. The language treated by Mix was a subset of statically scoped first order pure Lisp, and Mix was able to generate compilers from interpreters written in this language. The experiment showed that autoprojection was possible in practice; an automatic version of Mix was developed later [23]. Since then, autoprojectors for several languages have been implemented: for a subset of Turchin's Refal language [32], for an imperative flowchart language [19], for pattern matching based programs in the form of restricted term rewriting systems [5], and for first order functional languages with global variables [7].

Similix-2 has been developed and implemented by extending *Similix* [7], an existing autoprojector for a first order Scheme subset; Similix is referred to as Similix-1 in this paper. Similix-2 has inherited a number of features from Similix-1: primitive operators are user specified (as introduced in [12]), side-effecting operations on global variables (such as i/o operations) are treated in a semantically correct way, and residual programs never duplicate nor discard computations (cf. *call duplication* [33]). This paper does not cover these aspects; we refer to [7]. Handling global variables and duplication/discarding in the higher order case is described in [4].

1.1. Partial evaluation

Partial evaluation works by propagating the static input and reducing static operations. As an example, a conditional (`if` E_1 E_2 E_3) can be reduced at program specialization time if the expression E_1 is static, that is, if the value of E_1 depends only on the static input, not on the dynamic input. In that case the result of partially evaluating the conditional is the result of partially evaluating the branch chosen by evaluating the test E_1. If E_1 is dynamic, the conditional is left residual: a residual expression (`if` R-E_1 R-E_2 R-E_3) is produced. Here R-E_i is the residual expression that is the result of partially evaluating E_i.

We consider a specific form of partial evaluation called *polyvariant specialization* [9]. In the context of a recursive equation language, a function call is either *unfolded* or a residual call to a *specialized* function is generated [22, 23]. Where the source program is a set of (recursive) functions `f`, `g`, . . . , the residual program is a set of (recursive) specialized functions `f-1`, `f-2`, . . . , `g-1`, `g-2`, ... Each residual function is an instance of a source function, specialized with respect to values of its static parameters.

1.2. Binding time analysis

Experience shows that an important component of an autoprojector is the *preprocessor.* Preprocessing is performed *before* program specialization. Its purpose is to add *annotations* (attributes) to the source program [22]. The annotations guide the program specializer (which actually produces the residual program) in various ways: they tell whether variables are static or dynamic, that is, whether they will be bound to static values or residual expressions, whether operations can be reduced during program specialization, and whether calls and let-expressions should be unfolded. Annotations provide a way to relieve the specializer from taking decisions depending on the static input to the program being specialized, and this gives major improvements, especially when the specializer is self-applied [8] (the essential reason: the static input to the program with respect to which the specializer is being specialized is not available).

The central preprocessing phase is *binding time analysis* [23]. Binding time analysis is usually done by *abstract interpretation* (and is therefore *approximative*): the program is abstractly interpreted over a binding time lattice, in the simplest case the two-point lattice $\mathsf{S} \sqsubseteq \mathsf{D}$. The binding time value S is to be interpreted as "definitely static", i.e., it abstracts values that are available (*known*) at program specialization time. D means "possibly non-static" and abstracts values that are possibly not available (possibly *unknown*) at program specialization time. Variables and operations are then classified according to their binding times. Static operations are reduced during program specialization whereas residual code is generated for the dynamic ones. Static operations correspond to the overlined ones of [29], dynamic operations to the underlined ones.

1.3. Binding time analysing higher order programs

The difficult point in binding time analysing higher order programs is to associate lambda abstractions with applications. If, for example, a program contains the application `(x y)`, and if `x` during (partial) evaluation may be bound to (the value of) some abstraction, say `(lambda (z) (+ z 3))`, occurring elsewhere in the program, then the binding time value of `y` influences that of `z`. If, for instance, `y` is classified D, then `z` cannot be S. On the other hand, if `x` can never possibly be bound to (the value of) `(lambda (z) (+ z 3))`, then `y` has no influence on `z`. The—not very useful—conservative extreme would be to assume that *any* abstraction might be applied at *any* application point.

For first order languages, the control flow is easy to follow from the program syntax. But for higher order programs, the control is difficult to trace: how does one deduce from the program text `(x y)` that `y` influences `z`?

Nielson and Nielson have described an automatic binding time analysis for a higher order functional language [29] (they have not used their binding time analysis for partial evaluation). Their analysis treats the *typed* lambda calculus, using a two-level type system. The analysis is based on *type inference* and depends on the type information in the program being analysed: from the expression `(x y)`, it would identify that x had type $\mathsf{D} \rightarrow \cdots$ (since y has type D), and eventually this type would be unified (using least upper bounds) with the type of `(lambda (z) (+ z 3))`. Its type if $Z \rightarrow \cdots$, where Z is the type of z. Unifying the types implies $Z \sqsupseteq \mathsf{D}$.

Mogensen has described an abstract interpretation based analysis for a polymorphically typed higher order functional language where programs are written in curried named combinator form [26]. Mogensen describes binding time values for function types as a kind of abstract closures: an abstract closure consists of the name of the combinator and the binding time values of the free variables. A rather complex recursion detection machinery based on the type information is used to avoid generating infinite abstract closures. Mogensen has not implemented a partial evaluator that uses the result of the binding time analysis.

Both Nielson and Nielson's and Mogensen's binding time analyses are for typed languages and depend on the type knowledge. Our language is a subset of Scheme and thus untyped: a different binding time analysis is needed. In this paper we present an abstract interpretation based binding time analysis that uses information computed by a variant of Sestoft's *closure analysis* [34, 35]: a closure analysis is first performed on program *p*, then the binding time analysis is performed:

$$\textit{bt-annotations} = \textit{bt-analyse}(p, \textit{cl-analyse}(p)).$$

For each application point in the program, the closure analysis collects the set of lambda abstractions that for any evaluation of the program possibly may be applied at that point—for instance that x above may be bound to (the value of) `(lambda (z) (+ z 3))`. The analysis addresses *any* possible evaluation, not a particular one, so it must necessarily be approximative: it can give a *safe* description which, however, may be too conservative.

Using the information computed by the closure analysis, the binding time analysis immediately knows which formal parameters to lambda expressions that may be affected by an application (for instance that z depends on y). In this approach, binding time analysis is relatively simple to express; in contrast to Mogensen's analysis (which is also abstract interpretation based), no *structured* binding time values (such as $\mathsf{D} \rightarrow \mathsf{D}$) are needed. Termination of the binding time analysis is easily guaranteed: the binding time description is changed monotonically, and the set of binding time values is trivially finite since there are no structured values.

1.4. Outline

This paper is organized as follows. Section 2 describes the language treated by Similix-2. Section 3 discusses the unfolding (reduction) strategy. In Section 4, we present an interpreter for a lambda calculus language "Λ". The interpreter serves as an example of a higher order program. When it is specialized, programs in the Λ-language are in effect compiled into Scheme. Section 5 describes the closure analysis needed by the higher order binding time analysis. In Section 6, we discuss the representation of higher order values. The binding time analysis is presented in Section 7.

In Section 8, we exemplify partial evaluation of higher order programs: the Λ-interpreter is specialized, and we also specialize three other Λ-interpreters: one written in continuation passing style, one implementing call-by-name reduction, and one implementing call-by-need reduction. Benchmarks for Similix-2 are given in Section 9. Section 10 discusses how currying procedures may influence partial evaluation. Section 11 discusses related work and in Section 12 we conclude.

1.5. Prerequisites

Some knowledge about partial evaluation is required, e.g. as presented in [22] or [23].

2. Programming language

Similix-2 processes recursive equations expressed in a subset of Scheme [31]. The language is an extension of the language treated by Similix-1 [7]; the added expression forms are lambda abstractions and applications. Since programs follow the syntax of Scheme, they are directly executable in a Scheme environment.

A source program is expressed by a set of user defined procedures and a set of user defined operators. Following Scheme terminology, we use the term "procedure" rather than "function". Procedures are treated intensionally, whereas operators are treated extensionally. The partial evaluator knows the internal code of procedures. In contrast, an operator is a primitive operation: the partial evaluator never worries about how the internal operations are performed by a primitive operator. It can only do two things with a primitive operation: either evaluate the operation or suspend it, generating a residual call.

The BNF of the abstract syntax of programs is given in Fig. 1. Every expression is identified by a unique *label.* The labels are not part of the concrete syntax of a program, but they are important in the abstract one. Except for the labels, this abstract syntax is identical to the concrete one.

```
Pr ∈ Program,  PD ∈ Definition,  F ∈ FileName,
L-E ∈ LabeledExpression,  L ∈ Label,  E ∈ Expression,
C ∈ Constant,  V ∈ Variable,  O ∈ OperatorName,  P ∈ ProcedureName

Pr    ::=  (loadt F)* (load F)* PD+
PD    ::=  (define (P V*) L-E)
L-E   ::=  L E
E     ::=  C | V | (if L-E1 L-E2 L-E3) | (let ((V L-E1)) L-E2) |
           (O L-E*) | (P L-E*) | (lambda (V*) L-E1) | (L-E0 L-E*)
```

Fig. 1. Abstract syntax of Similix-2 Scheme subset.

The user defined primitive operators are defined in external modules in files loaded by the `loadt` expressions. Procedure definitions and `loadt` expressions from other files can be reused using `load`.

An expression is a constant C (boolean, number, string, or quoted construction), a variable V, a conditional (if L-E_1 L-E_2 L-E_3), a let-expression (let ((V L-E_1)) L-E_2) (unary for simplicity; L-E_1 is called the *actual parameter* expression and L-E_2 the *body* expression), a primitive operation (O L-E^*) (applying a user defined operator), a procedure call (P L-E^*) (applying a procedure defined with (define (P V^*) L-E)), a lambda abstraction (lambda (V^*) L-E_1), or an application (L-E_0 L-E^*) (applying an expression that evaluates to the value of a lambda abstraction). The order of evaluation is applicative (strict, call-by-value, inside-out), and arguments are evaluated in an unspecified order. To keep the language simple, there is no `letrec` (nor any `rec`); recursion is expressed using named procedures.

We note that a let-expression is treated as a construct on its own. Contrasting to what is usual in Scheme, a let-expression (let ((V L-E_1)) L-E_2) is thus *not* syntactically expanded into ((lambda (V) L-E_2) L-E_1). The expanded form is unnecessarily complex for our purpose: it contains two syntactic forms, a lambda abstraction and an application, both higher order constructs. On the other hand, a let-expression is a single first order construct, thus simpler to partially evaluate.

Procedure calls are treated differently than higher order applications; the two forms are therefore distinguished syntactically. Both procedure calls and higher order applications are, in turn, distinguished from primitive operations. The distinctions are made during parsing. Notice that the application forms (O L-E^*), (P L-E^*), and (L-E_0 L-E^*) are *not* curried; L-E^* is a tuple. Each primitive operator, each procedure, and each lambda expression has its own fixed arity. Currying is expressible by using nested `lambda` definitions.

Program input is assumed to be first order (ground values, i.e., constants). The reason is that higher order values are treated intensionally during program specialization; the internal representation of functional values depends on the text of the program being partially evaluated (see Section 6).

2.1. *Syntactic extensions*

A number of built-in syntactic extensions are treated by Similix. We mention one standard Scheme extension which is used in the examples later: `cond`. It is expanded into a sequence of `if` expressions. The system also treats user defined syntactic extensions following the syntax of [25] (only a subset of Kohlbecker's language is treated).

3. Unfolding strategy

In this section we discuss Similix-2's unfolding (reduction) strategy.

3.1. Background

During program specialization, calls are unfolded to increase efficiency of residual programs. However, when unfolding recursive definitions, infinite unfolding may result. Some calls must therefore be suspended (not unfolded, kept residual). A *strategy* is therefore needed to decide when to unfold and when to suspend. There exist successful strategies for first order languages [33, 7]. Based on binding time information, some calls are (pre-)annotated as unfoldable. During program specialization, only these calls are unfolded (additional post-unfolding also takes place).

Similix-2 is an extension of Similix-1. Let us therefore recapitulate the unfolding startegy of Similix-1 [7] that treats a first order language. Compared to standard evaluation, partial evaluation uses a different reduction strategy when—and, in the case of a first order language, only when—processing *dynamic conditionals*, i.e., conditionals with a dynamic test (according to the binding time analysis). Standard evaluation evaluates only one of the branches, but partial evaluation specializes both. Partial evaluation is therefore strict in both branches and thus *less terminating*: there is a danger of infinite unfolding even if standard evaluation always terminates.

Dynamic conditionals are therefore chosen as (the only) specialization points; all calls to user defined procedures are unfolded. This is implemented by *lifting* out each dynamic conditional by defining new procedures. Such a procedure's parameters are the free variables of the conditional and its body is the conditional expression itself. This process of "if lifting" is comparable to *lambda lifting* [20] and is perfomed in preprocessing prior to program specialization. Calls to these new procedures are not unfolded, so the procedures are specialized during program specialization. Therefore, if the same dynamic conditional is encountered twice (during program specialization) with the same values of its free static variables, only one residual version is produced. This residual definition is then shared, possibly recursively (it may call itself).

The strategy guarantees that infinite unfolding is only possible if the program specializer enters a completely statically controlled infinite loop. But standard evaluation would not terminate in that case either (if the loop were entered), and then we accept that the program specializer does not terminate.

3.2. Beta reducing lambda expressions

The two higher order expression forms `(lambda (V*) L-E`$_1$`)` and `(L-E`$_0$` L-E*)`, not handled by Similix-1, need additional consideration. Partial evaluation should beta reduce applications `(E ...)` when `E` evaluates to (the value of) a lambda expression at program specialization time. It is not always possible to beta reduce: `E` may be dynamic in which case it evaluates to a residual expression. We choose the liberal reduction strategy "beta reduce whenever possible".

This may potentially give infinite reduction even when processing a pure lambda expression that does not contain procedure calls (the Y-combinator can be programmed as a lambda expression). Infinite reduction can, however, only happen if the loop is completely statically controlled. A dynamic conditional will in pure lambda calculus form correspond to a dynamic application `(E ...)`, i.e., E will evaluate to a residual expression. Therefore `(E ...)` cannot be beta reduced, so beta reduction stops. This behavior corresponds nicely to the one described in the previous section: dynamic conditionals are used to break procedure call unfolding. Infinite unfolding may only happen when processing a completely statically controlled loop.

A lambda expression (lambda ($V_1 \ldots V_n$) E) cannot be beta reduced away when its value occurs as part of a residual code piece; this is explained later (Section 7.2). In that case, residual code (lambda ($V_1 \ldots V_n$) R-E) is generated (R-E is the result of partially evaluating E). Hence, when processing a *dynamic lambda expression*—a lambda expression which is not beta reducible at program specialization time, partial evaluation is strict in the body expression E. Standard evaluation, however, is not strict in E.

Dynamic lambda expressions and dynamic conditionals therefore have in common that partial evaluation is more strict (less terminating) than standard evaluation. This suggests that we create specialization points for dynamic lambda expressions, just as we did for dynamic conditionals. So similarly to the "if lifting" described above, dynamic lambda expressions are lambda lifted to create specialization points. This prevents infinite unfolding when for instance the fixed point operator is defined as in Fig. 4 and applied to a dynamic argument.

4. An example interpreter for the language Λ

To illustrate how Similix-2 works, we now present a language Λ and an interpreter for Λ written in Scheme. Λ is a statically scoped lambda calculus language with unary abstractions and applications, constants, binary primitive operations, a conditional, and a recursive "let". A program is an expression following the (abstract) syntax given in Fig. 2. A program takes one input value; all free variables of the expression constituting the program are bound to this input value. For an example, this program computes the factorial function:

```
(letrec f (lambda x (if (=x 0) 1 (* x (f (-x 1))))) (f input))
```

The free variable `input` (its name is arbitrary) refers to the input value.

```
E ∈ Expr,  C ∈ Const,  V ∈ Var,  B ∈ Binop

E  ::=  C | V | (B E1 E2) | (if E1 E2 E3) |
        (lambda V E) | (letrec V E1 E2) | (E1 E2)
```

Fig. 2. Abstract syntax of Λ.

4.1. Denotational semantics

The denotational semantics of the language is specified in Fig. 3. No type checking is performed; this would require injection tags on values and has been omitted for simplicity. Notice that the initial environment $\lambda \mathrm{V}.w$ binds all (free) variables to the input value.

Semantic domains:
$w \in \mathit{Value}$, $r \in \mathit{Environment} = \mathtt{Var} \to \mathit{Value}$

Valuation functions:
$\mathit{run} : \mathtt{Expr} \to \mathit{Value} \to \mathit{Value}$
$\mathit{run}[\![\mathrm{E}]\!]w = E[\![\mathrm{E}]\!]\, \lambda\mathrm{V}.w$

$E : \mathtt{Expr} \to \mathit{Environment} \to \mathit{Value}$

$$\begin{array}{lcl}
E[\![\mathtt{C}]\!]r & = & C[\![\mathtt{C}]\!] \\
E[\![\mathtt{V}]\!]r & = & r([\![\mathtt{V}]\!]) \\
E[\![\mathtt{(B\ E_1\ E_2)}]\!]r & = & B[\![\mathtt{B}]\!](E[\![\mathtt{E_1}]\!]r)(E[\![\mathtt{E_2}]\!]r) \\
E[\![\mathtt{(if\ E_1\ E_2\ E_3)}]\!]r & = & E[\![\mathtt{E_1}]\!]r \to E[\![\mathtt{E_2}]\!]r \,[\!]\, E[\![\mathtt{E_3}]\!]r \\
E[\![\mathtt{(lambda\ V\ E)}]\!]r & = & \lambda w.E[\![\mathtt{E}]\!][[\![\mathtt{V}]\!] \mapsto w]r \\
E[\![\mathtt{(letrec\ V\ E_1\ E_2)}]\!]r & = & E[\![\mathtt{E_2}]\!](\mathit{fix}(\lambda r_1.[[\![\mathtt{V}]\!] \mapsto E[\![\mathtt{E_1}]\!]r_1]r)) \\
E[\![\mathtt{(E_1\ E_2)}]\!]r & = & (E[\![\mathtt{E_1}]\!]r)(E[\![\mathtt{E_2}]\!]r)
\end{array}$$

$C : \mathtt{Const} \to \mathit{Value}$ *unspecified*

$B : \mathtt{Binop} \to \mathit{Value} \to \mathit{Value} \to \mathit{Value}$ *unspecified*

Fig. 3. Denotatinal semantics of A.

4.2. A-interpreter text

Because Scheme uses strict evaluation, it is straightforward to convert the denotational semantics into a Scheme program—an interpreter—if all functions are considered strict in all arguments. This of course defines a strict semantics of the interpreted language. In Section 8.3, we show an interpreter that defines a non-strict semantics.

To translate the semantics into Scheme, we first uncurry the functions *run*, *E*, and *B*; this is simple since the functions already are used in an uncurried way. Uncurrying is beneficial from a readability point of view (`(f x y)` vs. `((f x) y)`), and it also sometimes gives better specialization (more about this in Section 10).

We now give the interpreter text (Fig. 4). *C* is just the identity function and has been omitted. The comments (`0-5` and `g-m`) are used for reference later (Section 5.4).

Syntax accessors (`cst-C`, `var-V`, etc.), syntax predicates (`isCst?`, `isVar?`, etc.), and `ext` have been defined as primitive operations in the file `"lam-int.adt"`. The primitive `ext` corresponds to the *B* function in the semantics and applies binary operators to their values. The standard Scheme primitives `equal?` and `error` are

```
(loadt "scheme.adt")
(loadt "lam-int.adt")
(load "lam-aux.sim")

(define (run E w) (_E E (lambda (V) w)))                          ; 0

(define (_E E r)
  (cond
   ((isCst? E)
    (cst-C E))
   ((isVar? E)
    (r (var-V E)))                                                ; g
   ((isBinop? E)
    (ext (binop-B E) (_E (binop-E1 E) r) (_E (binop-E2 E) r)))
   ((isIf? E)
    (if (_E (if-E1 E) r)
        (_E (if-E2 E) r)
        (_E (if-E3 E) r)))
   ((isLambda? E)
    (lambda (w)                                                   ; 2
      (_E (lambda-E E) (upd (lambda-V E) w r))))                  ; 1,h
   ((isLetrec? E)
    (_E (letrec-E2 E)
      (fix (lambda (r1)                                           ; 4
             (upd (letrec-V E) (_E (letrec-E1 E) r1) r)))))       ; 3,i
   ((isApply? E)
    ((_E (apply-E1 E) r) (_E (apply-E2 E) r)))                    ; j
   (else
    (error '_E "unknown form: ~s" E))))

(define (fix f) (lambda (x) ((f (fix f)) x)))                     ; 5,k,m
```

Fig. 4. Direct style Λ-interpreter written in Scheme.

defined in "scheme.adt". The file "lam-aux.sim" (Fig. 5) defines environment updating as a syntactic extension.

4.3. *Analysing the interpreter*

Partially evaluating the interpreter with static program input (run's E parameter) and dynamic data input (run's w parameter) in effect compiles Λ-programs into Scheme (since Similix generates residual code in Scheme).

```
(extend-syntax (upd)
  ((upd V w r)
   (lambda (V1)
     (if (equal? V V1)
         w
         (r V1)))))
```

Fig. 5. Environment updating.

What can be expected from binding time analysing the interpreter? Λ is statically scoped, so environment operations should be classified reducible: they can be performed at program specialization time (compile time). For instance, the analysis should detect that `r` is statically available in the expression `(r (var-V E))`, and hence the application of the environment should be classified (beta) reducible.

On the other hand, the expression `A` =

```
((_E (apply-E1 E) r) (_E (apply-E2 E) r))
```

clearly is a *run time* application, so we would expect it to be classified residual. That is, if the interpreted program contains an expression `E` satisfying `(isApply? E)`, then we expect (a residual/compiled version of) `A` to occur in the specialized interpreter, i.e., in the target program.

In the following sections we describe our binding time analysis in detail. We first address the closure analysis.

5. Closure analysis

In this section, we give a formal presentation of the closure analysis. The purpose of the analysis is for every variable and every expression to collect the set of possible (values of) lambda abstractions that the variable may be bound to/the expression may evaluate to.

The analysis originates from one developed by Sestoft (for the purpose of globalizing variables in higher order programs) for untyped higher order programs in curried named combinator form [34, 35]. Our analysis is basically an extended version of Sestoft's, adapted to our concrete language. The extension is that we handle *multi-applications*, that is, our lambda abstractions are n-ary, not just unary (Sestoft mentions this as a possible extension).

We describe the analysis in a different and more algorithmic way than Sestoft's. Our closure analysis works iteratively by continuously updating global mappings of variables and expressions to closure information. Using this method, the program text need only be traversed once for each fixed point iteration. This gives a relatively simple description (only one function traversing syntax), and it also naturally leads to an efficient implementation: the closure information is kept as attributes in the abstract syntax, which is destructively updated.

5.1. Semantic domains and functions

We assume given the following injective functions from syntactic to semantic domains:

$$\mathscr{L}:\texttt{Label} \rightarrow \textit{Label},$$
$$\mathscr{P}:\texttt{ProcedureName} \rightarrow \textit{Label},$$
$$\mathscr{V}:\texttt{Variable} \rightarrow \textit{Variable}.$$

$\mathscr{L}$ and $\mathscr{V}$ are just for "purity". $\mathscr{P}$ associates a procedure name with the label of the procedure's body expression: when analysing a procedure call, this gives access to information about the procedure body. The semantic domains are defined by

$$\begin{aligned}&Index = \{1, 2, \ldots\},\\ &\ell \in Label,\\ &v \in Variable = Label \times Index.\end{aligned}$$

Formal parameters to a procedure P are identified as $(\ell, 1)$, $(\ell, 2)$, etc., where $\ell = \mathscr{P}[\![\mathrm{P}]\!]$. The formal parameters of a lambda expression with *body* expression with label L will be identified as $(\ell, 1)$, $(\ell, 2)$, etc., where $\ell = \mathscr{L}[\![\mathrm{L}]\!]$. These identifications are unique as they do not conflict with formal procedure parameter labels. The formal parameter V of a let-expression is identified by some arbitrary unique value v in the domain *Variable*.

During standard evaluation, a lambda expression operationally evaluates to a *closure*. A closure may, depending on the implementation, for instance be an expression/environment pair or a pair containing the lambda expression and a list of values for its free variables. An *abstract closure* abstracts away the environment (alternatively the list of values for the free variables), thus only keeping the lambda expression itself. Abstract closures thus only depend on program text, not on values; the closure analysis works only on the program text.

In the closure analysis, we identify an abstract closure—a lambda expression—by a label. For technical conveniency, we use the label of the lambda expression's *body* rather than the label of the lambda expression itself.

The closure analysis computes two mappings, μ_{cl} and ρ_{cl}, the first one binding labels and the second one binding variables. The codomain of both mappings is the powerset of abstract closures (with the usual subset inclusion ordering). For every expression, μ_{cl} thus abstracts the set of closures that the expression may possibly evaluate to (during any possible standard program execution); for every variable, ρ_{cl} abstracts the set of closures that the variable may possibly be bound to:

$$\begin{aligned}&AbsClosure = Label,\\ &acs \in AbsClSet = \wp(AbsClosure),\\ &\mu_{cl} \in ClMap = Label \to AbsClSet,\\ &\rho_{cl} \in ClEnv = Variable \to AbsClSet.\end{aligned}$$

Maps and environments are updated by corresponding monotonic update functions. Map updating is performed by the function *upd*:

$$\begin{aligned}&upd : Label \to AbsClSet \to ClMap \to ClMap,\\ &upd\ \ell\ acs\ \mu_{cl} = \mu_{cl} \sqcup [\ell \mapsto acs]\bot_{ClMap}.\end{aligned}$$

Environment updating has functionality

$$Variable \to AbsClSet \to ClEnv \to ClEnv$$

and is defined in a similar way. For readability, we uniformly refer to all updating functions simply as *upd*; the functionality is clear from the context. The least upper bound on abstract closure sets (from the domain *AbsClSet*) is set union. The least upper bounds on functions and cartesian products are defined pointwise:

$$\mu_{cl} \sqcup \mu'_{cl} = \lambda \ell .\, \mu_{cl}(\ell) \sqcup \mu'_{cl}(\ell),$$
$$(\mu_{cl}, \rho_{cl}) \sqcup (\mu'_{cl}, \rho'_{cl}) = (\mu_{cl} \sqcup \mu'_{cl}, \rho_{cl} \sqcup \rho'_{cl}).$$

Finally, we use a function for getting the arity of the lambda expression identified by an abstract closure:

$$\mathit{arity} : \mathit{AbsClosure} \to \{0, 1, 2, \ldots\}.$$

5.2. The analysis

The closure analysis rules are given in Fig. 6. Given a set of procedure definitions PD^{+}, the function *Cl* computes the two mappings μ_{cl} and ρ_{cl}. The mappings are computed as simultaneous fixed points. Initially, all labels and all variables are mapped onto the empty abstract closure set (since the input to a program is first order and thus contains no closures).

Explicit quantification of indices is avoided when clear from the context; primitive operators and procedures may be nullary in which case the index i ranges over the empty set.

The rules for constants, variables, conditionals, and let-expressions are straightforward. For primitive operations `(O ...)`, note that a closure occurring in an argument may possibly be returned, but no new closure may be introduced. For procedure calls `(P ...)`, a closure returned by the procedure body may be returned; care must be taken to account for the influence on the formal parameters of the procedure. Both for primitive operations and for procedure calls, it is taken into account that n may be 0 (therefore the term "$(\mu_{cl}, \rho_{cl}) \sqcup \cdots$" in Fig. 6). Lambda abstractions are the "sources" of closures; note that (as mentioned earlier) an abstract closure is identified by the label of the lambda expression's body.

The rule for applications is the most complex one. First, the set *acs* of lambda abstractions that E_0 may evaluate to is found. Then μ_{cl} is updated: the application (E) may evaluate to a closure which is the result of evaluating the *body* of any of the lambda abstractions in the set *acs*. Lambda abstractions are identified by the body labels, so μ'_{cl} is simply applied to the elements (ℓ') in *acs*. Finally, ρ_{cl} is updated: E influences the formal parameters of all lambda abstractions which E_0 may evaluate to. The ith parameter is influenced by E_i.

5.3. Finiteness

For any given program, there is a finite number of abstract closure sets. The mappings μ_{cl} and ρ_{cl} have finite domains (the set of labels and the set of variables are both finite) and they are updated monotonically; hence they can only be updated

```
$Cl : \texttt{Definition}^{+} \rightarrow ClMap \times ClEnv$
$Cl[\![\texttt{(define (...) L}_1\texttt{E}_1\texttt{) ... (define (...) L}_n\texttt{E}_n\texttt{)}]\!] = fix(\lambda(\mu_{cl}, \rho_{cl}) \,.\, \bigsqcup_i cl[\![\texttt{L}_i\texttt{E}_i]\!]\mu_{cl}\rho_{cl})$

$cl : \texttt{LabeledExpression} \rightarrow ClMap \rightarrow ClEnv \rightarrow ClMap \times ClEnv$
$cl[\![\texttt{L E}]\!]\mu_{cl}\rho_{cl} =$
  $let\ \ell = \mathcal{L}[\![\texttt{L}]\!]\ in$
    $case\ [\![\texttt{E}]\!]\ of$
      $[\![\texttt{C}]\!] : (upd\ \ell\ \{\}\ \mu_{cl}, \rho_{cl})$
      $[\![\texttt{V}]\!] : (upd\ \ell\ \rho_{cl}(\mathcal{V}[\![\texttt{V}]\!])\ \mu_{cl}, \rho_{cl})$
      $[\![\texttt{(if L}_1\texttt{E}_1\texttt{ L}_2\texttt{E}_2\texttt{ L}_3\texttt{E}_3\texttt{)}]\!] : let\ (\mu'_{cl}, \rho'_{cl}) = \bigsqcup_i cl[\![\texttt{L}_i\texttt{E}_i]\!]\mu_{cl}\rho_{cl}\ in$
        $(upd\ \ell\ (\mu'_{cl}(\mathcal{L}[\![\texttt{L}_2]\!]) \sqcup \mu'_{cl}(\mathcal{L}[\![\texttt{L}_3]\!]))\ \mu'_{cl}, \rho'_{cl})$
      $[\![\texttt{(let ((V L}_1\texttt{E}_1\texttt{)) L}_2\texttt{E}_2\texttt{)}]\!] : let\ (\mu'_{cl}, \rho'_{cl}) = \bigsqcup_i cl[\![\texttt{L}_i\texttt{E}_i]\!]\mu_{cl}\rho_{cl}\ in$
        $(upd\ \ell\ \mu'_{cl}(\mathcal{L}[\![\texttt{L}_2]\!])\ \mu'_{cl}, upd\ \mathcal{V}[\![\texttt{V}]\!]\ \mu'_{cl}(\mathcal{L}[\![\texttt{L}_1]\!])\ \rho'_{cl})$
      $[\![\texttt{(O L}_1\texttt{E}_1\texttt{ ... L}_n\texttt{E}_n\texttt{)}]\!] : let\ (\mu'_{cl}, \rho'_{cl}) = (\mu_{cl}, \rho_{cl}) \sqcup \bigsqcup_i cl[\![\texttt{L}_i\texttt{E}_i]\!]\mu_{cl}\rho_{cl}\ in$
        $(upd\ \ell\ (\bigsqcup_i \mu'_{cl}(\mathcal{L}[\![\texttt{L}_i]\!]))\ \mu'_{cl}, \rho'_{cl})$
      $[\![\texttt{(P L}_1\texttt{E}_1\texttt{ ... L}_n\texttt{E}_n\texttt{)}]\!] : let\ (\mu'_{cl}, \rho'_{cl}) = (\mu_{cl}, \rho_{cl}) \sqcup \bigsqcup_i cl[\![\texttt{L}_i\texttt{E}_i]\!]\mu_{cl}\rho_{cl}\ in$
        $(upd\ \ell\ \mu'_{cl}(\mathcal{P}[\![\texttt{P}]\!])\ \mu'_{cl}, \rho'_{cl} \sqcup \bigsqcup_i(upd\ (\mathcal{P}[\![\texttt{P}]\!], i)\ \mu'_{cl}(\mathcal{L}[\![\texttt{L}_i]\!]))\ \rho'_{cl})$
      $[\![\texttt{(lambda (V}_1\texttt{ ... V}_n\texttt{) L}_1\texttt{E}_1\texttt{)}]\!] : let\ (\mu'_{cl}, \rho'_{cl}) = cl[\![\texttt{L}_1\texttt{E}_1]\!]\mu_{cl}\rho_{cl}\ in$
        $(upd\ \ell\ \{\mathcal{L}[\![\texttt{L}_1]\!]\}\ \mu'_{cl}, \rho'_{cl})$
      $[\![\texttt{(L}_0\texttt{E}_0\texttt{ L}_1\texttt{E}_1\texttt{ ... L}_n\texttt{E}_n\texttt{)}]\!] : let\ (\mu'_{cl}, \rho'_{cl}) = \bigsqcup_i cl[\![\texttt{L}_i\texttt{E}_i]\!]\mu_{cl}\rho_{cl}\ in$
        $let\ acs = \{\ell' \mid \ell' \in \mu'_{cl}(\mathcal{L}[\![\texttt{L}_0]\!]) \wedge arity(\ell') = n\}\ in$
          $(upd\ \ell\ (\bigsqcup_{\ell' \in acs} \mu'_{cl}(\ell'))\ \mu'_{cl}, \rho'_{cl} \sqcup \bigsqcup_{i \geq 1, \ell' \in acs}(upd\ (\ell', i)\ \mu'_{cl}(\mathcal{L}[\![\texttt{L}_i]\!])\ \rho'_{cl}))$
    $end$
```

Fig. 6. Closure analysis.

a finite number of times. Fixed point iteration will therefore stabilize after a finite number of iterations.

5.4. *Application to the A-interpreter*

We end the description of the closure analysis by showing what it gives when applied to the *A*-interpreter.

The lambda abstractions are referred to by a number (0–5), the application points by a letter (g–m); see the comments in the interpreter text. Each use of upd is macro expanded into an expression containing a lambda abstraction `(lambda (V1)...)` and an application `(r V1)`. `k` identifies the application of `f` to `(fix f)`, `l` the application of `(f (fix f))` to `x`. The closure analysis gives the following possible abstractions at the application points:

g, h, i: 0, 1, 5   j: 2   k: 4   m: 3.

We see that at environment application points, g, h, and i, the environment abstractions 0, 1, and 5 (but not abstraction 3!) are the (only) possibilities. Abstraction 2, which implements lambda abstraction in the interpreted language, is the only one which may be applied at application point j that implements application in the interpreted language. The only closure values that the functional f may be bound to at point k are the ones coming from abstraction 4 that defines environments recursively. Finally, an "unrolled" recursive environment at point m can only come from abstraction 3.

6. Representing higher order values

When partially evaluating first order languages, values are first order and can be represented by themselves. When partially evaluating a higher order language, a problem arises: how should we represent higher order values at program specialization time?

A first proposal would be simply to represent a higher order value by a higher order value, thus in effect relying on the Scheme implementation's internal representation. This would suffice if the partial evaluator did not specialize procedure calls. However, because of specialization, we need to compare higher order values for intensional equality. This is necessary when testing whether a procedure has been specialized with respect to the same values before—and some values may be higher order.

We therefore represent a higher order value explicitly by a *pe-closure.* A pe-closure is a data structure containing an identification of the lambda expression and values for its free variables. This representation corresponds to closures used in implementations (mentioned in Section 5.1), but in pe-closures the values of the free variables are *pe-values.* A pe-value is either a static (first order) value, a residual expression, or a pe-closure. Static (free) variables are bound to static values, dynamic variables to residual expressions, and other variables are bound to pe-closures. Notice that it is possible to build nested pe-closures. A pe-closures is a *partially static structure* [28]: its structure (spine) is available at program specialization time and it contains static as well as dynamic leaves.

Two pe-values are considered equal if both (1) their lambda expression identifications are equal, and (2) the pe-values of all free variables bound to static (first order) values or pe-closures are equal. This identifies many (but not all) higher order values which are functionally equal.

When specializing a procedure call with respect to a pe-closure value, each dynamic leaf of the (possibly nested) pe-closure becomes an argument to the residual procedure call. This achieves *arity raising* [32, 27] when specializing interpreters that represent environments as functions.

7. Higher order binding time analysis

This section describes the higher order binding time analysis that assigns a binding time value to every variable and every expression in a program. The analysis uses the information collected in closure analysis.

7.1. The binding time lattice

The binding time lattice has four values. It is an extension of the classical two point binding time lattice; the two new values are Cl and $\perp$:

$$b \in BTValue = (\{\perp, \mathsf{S}, \mathsf{Cl}, \mathsf{D}\}, \sqsubseteq).$$

The partial ordering is given by

$$\begin{tikzcd}
 & \mathsf{D} & \\
\mathsf{S} \arrow[ur, dash] & & \mathsf{Cl} \arrow[ul, dash] \\
 & \perp \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

S approximates ordinary first order static values (constants), Cl approximates pe-closure values, and D approximates dynamic residual code expressions. The value $\perp$ is needed because S and Cl are incomparable. $\perp$ is the initial value before binding time analysis; if the binding time value of an expression remains $\perp$ after the binding time analysis, that expression is either unreachable or always non-terminating.

7.2. Residual code contexts

Sometimes static values occur in *residual code contexts* during program specialization. A residual code context is one in which the value must be *lifted* into residual code (note: this is a different use of the word "lifting" than in Section 3). Let us for instance consider specialization of the expression `(cons x y)` where x is static and y dynamic. During program specialization, x may be bound to, say, the value `(1 . 2)` and y to the residual expression `(cdr z_27)`. The result of specializing `(cons x y)` will then be the residual expression `(cons (quote (1 . 2)) (cdr z_27))`. Notice how the static *value* `(1 . 2)` has been lifted into the residual *expression* `(quote (1 . 2))`.

We take the "orthodox" view on binding times: if a variable has binding time value D, it will always be bound to a residual expression at program specialization time. Similarly, the value of any expression with binding time tag D will always be a residual expression. The advantage of this approach is that tags on values are avoided during program specialization: the binding time information uniquely determines which kind of pe-value a variable is bound to/an expression evaluates to. The disadvantage is that some possible reductions are not performed; some of these are, however, performed in postprocessing.

There are three kinds of residual code contexts that cause values to be lifted.

(1) If a value is the result of specializing an expression E with binding time tag D, then this value occurs in a residual code context since the result of specializing E is a residual expression.

(2) If a value is an argument to a compound expression which is not reduced during program specialization, then it sometimes occurs in a residual code context. This was exemplified by the `(cons x y)` example above and is detailed below.

(3) If a value is the result of specializing the body of the program's goal procedure, then it occurs in a residual code context. Residual code is always produced for the body of the goal procedure, even if all inputs to the partial evaluator are static (the goal procedure is therefore a special case).

An argument to a compond expression which is not reduced during program specialization is not always a residual code context: procedure calls are an exception. When calls are not reduced (unfolded), they are specialized with respect to the non-dynamic parameter values (static first order values and pe-closures). Residual code is thus *not* generated for the non-dynamic parameter values which consequently are not lifted into residual code.

7.3. *Annotating lambda expressions*

Lifting pe-closures into residual lambda expressions "on-line" during program specialization when a pe-closure occurs in a residual code context is relatively complex: possibly nested pe-closures need to be traversed. Furthermore, on-line lifting results in duplicated code if the same pe-closure is lifted more than once.

We therefore use a different approach: when specializing a lambda expression `(lambda (`$V_1 \ldots V_n$`) E)` whose (pe-)value may *possibly* later be used in a residual code context, we immediately generate a residual lambda expression `(lambda (`$V_1 \ldots V_n$`) R-E)` (cf. also Section 3.2). Notice that the parameters `(`$V_1 \ldots V_n$`)` then become dynamic since no beta reduction takes place.

Only when specializing a lambda expression whose value will *definitely* not be used in a residual code context do we generate a pe-closure. An application `(E . . . )` is then reducible if and only if `E` is guaranteed to evaluate to a pe-closure.

Detecting possible residual code contexts can be done already in binding time analysis, so lambda expressions will be annotated here, either with binding time tag Cl or binding time tag D. During program specialization, Cl-annotated lambda expressions are then specialized to pe-closures and D-annotated lambda expressions are specialized to residual lambda expressions. In binding time analysis, lambda expressions will initially optimistically get binding time tag Cl. The binding time analysis will then *raise* annotations of lambda expressions from Cl to D when necessary. When raising the annotation of a lambda expression, its formal parameters must also be raised (since they become dynamic).

The mapping μ_{cl} computed in closure analysis is used to find the lambda expressions whose pe-values (pe-closures) may possibly occur in a given residual code context. Raising is necessary for the three kinds of residual code contexts described above (Section 7.2). Some optimization is possible for the context kind (2). An analysis shows that no explicit raising is necessary for arguments to the following compound expressions forms: conditionals, let-expressions, and primitive operations.

The reason is that these raisings are "covered" by the raising done for an expression with binding time tag D (residual code context kind (1)). To see this, we first note that conditionals are not reduced in case of a dynamic test and that let-expressions are possibly not reduced in case of a dynamic actual parameter expression. The point now is that if any *other* argument expression, which gets "caught" in a residual code context (conditionals: the "then" and "else" branches; let-expressions: the body; primitive operations: any argument expression), may return a higher order value abstracted by some abstract closure, then the closure analysis assumes that whole compound expression may return a higher order value abstracted by the same abstract closure. Hence, the occurrence of the argument in a residual code context is "captured" by the raising done for the whole compound expression (which has binding time value D).

One might think of introducing a binding time value S-*or*-Cl lying above S and Cl, but below D. This would make sense since Scheme is dynamically typed. Introducing S-*or*-Cl would give additional precision in the description, but the program specializer would be burdened: any value of the S-*or*-Cl type would need to be *tagged* since the binding time tag would no longer uniquely determine the kind of pe-value. This is avoided by letting $\mathsf{S} \sqcup \mathsf{Cl} = \mathsf{D}$.

7.4. *Domains and functions*

The binding time analysis computes two mappings:

$$\mu_{bt} \in \mathit{BTMap} = \mathit{Label} \to \mathit{BTValue},$$
$$\rho_{bt} \in \mathit{BTEnv} = \mathit{Variable} \to \mathit{BTValue}.$$

These are dual to the closure mappings μ_{cl} and ρ_{cl}, and they are updated in a similar way.

The closure analysis identifies an abstract closure by the label of the body of the lambda expression. The binding time value of a lambda abstraction will be assigned to the label of the lambda expression itself (the body has its own binding time value), so we introduce the function $\ell 2\ell$:

$$\ell 2\ell : \mathit{Label} \to \mathit{Label}.$$

Given the label of the body of a lambda expression, $\ell 2\ell$ returns the label of the lambda expression itself.

Given the label of an expression, *raise* is the function that raises the annotations of the set of lambda expressions, which that expression may return:

$$\begin{aligned}
&\mathit{raise} : \mathit{Label} \to \mathit{BTMap} \to \mathit{BTEnv} \to \mathit{BTMap} \times \mathit{BTEnv},\\
&\mathit{raise}\ \ell\ \mu_{bt}\ \rho_{bt} =\\
&\quad \mathit{let}\ \mathit{acs} = \mu_{cl}(\ell)\\
&\qquad \bigsqcup_{\ell' \in \mathit{acs}} \left(\mathit{upd}\ \ell 2\ell(\ell')\ \mathsf{D}\ \mu_{bt},\ \bigsqcup_{i \in \{1 \ldots \mathit{arity}(\ell')\}} (\mathit{upd}\ (\ell', i)\ \mathsf{D}\ \rho_{bt}) \right) \sqcup (\mu_{bt}, \rho_{bt})
\end{aligned}$$

Note that the formal parameters of the lambda expressions are also raised (ρ_{bt} is updated).

The binding time value of an expression ($\texttt{(O}\ \mathrm{L_1E_1}\dots\ \mathrm{L_nE_n}\texttt{)}$ is typically $\bigsqcup_i$ ("the binding time value of $\mathrm{L_iE_i}$"). It is possible for the user to define a more conservative binding time function for primitive operations than the one above. This is for instance useful for *generalizing* [36], i.e. forcing a value to become dynamic (sometimes needed for ensuring termination of program specialization). The binding time value of a primitive application is therefore defined via a function $\mathcal{O}$:

$$\mathcal{O}: \texttt{OperatorName} \rightarrow \mathit{BTValue}^* \rightarrow \mathit{BTValue}.$$

In practice, a binding time function is user defined for each primitive [7].

7.5. *The analysis*

The binding time analysis rules are given in Fig. 7. Given a set of procedure definitions, a label identifying the body of the goal procedure, and an initial binding time description ρ_{bt}^{input}, the program is binding time analysed by propagating binding time values through the program.

Treating primitive operations working on higher order structures (such as Scheme's `procedure?`) introduces complications since the program specializer represents higher order values as pe-closures, not as Scheme procedure values. By least upper bounding the arguments with S, any primitive operation on a higher order value becomes dynamic whereby the problem is avoided (since no reduction takes place at program specialization time).

The applications of *raise* have been superscripted; the numbers refer to the three kinds of residual code contexts (Section 7.2) that cause lambda annotations to be raised (Section 7.3).

7.6. *Finiteness*

There is a finite number of binding time values. The mappings μ_{bt} and ρ_{bt} have finite domains (the set of labels and the set of variables are both finite) and they are updated monotonically; hence they can only be updated a finite number of times. Fixed point iteration will therefore stabilize after a finite number of iterations.

7.7. *Application to the Λ-interpreter*

When applied to the Λ-interpreter with static program input and dynamic data input, the binding time analysis correctly annotates the lambda abstractions for environment processing, 0, 1, 3, 4, and 5, as (beta) reducible. Dually, the applications `g`, `h`, `i`, `k`, and `m`, become reducible: the expression to be applied in all cases gets binding time value Cl. The lambda expression 2 and the application `j` become residual.

The formal parameters to the lambda expressions 0, 1, 3, and 5 are all static (abstracted by S). The parameter of abstraction 4 is a pe-closure (abstracted by Cl), but this is only what one could expect: the parameter is an environment. Finally, the parameter of abstraction 2 is dynamic (abstracted by D).

```
$BT : \mathtt{Definition}^{+} \rightarrow Label \rightarrow BTEnv \rightarrow BTMap \times BTEnv$
$BT\llbracket\mathtt{(define\ (\ldots)\ L_1E_1)\ \ldots(define\ (\ldots)\ L_nE_n)}\rrbracket =$
  $fix(\lambda(\mu_{bt}, \rho_{bt}) \,.\, (\mu_{bt}^{init}, \rho_{bt}^{init}) \sqcup \bigsqcup_i bt\llbracket\mathtt{L_iE_i}\rrbracket\mu_{bt}\rho_{bt})$
    $where\ (\mu_{bt}^{init}, \rho_{bt}^{init}) = raise^3\ \ell_{goal}\ \bot_{BTMap}\ \rho_{bt}^{input}$

$bt : \mathtt{LabeledExpression} \rightarrow BTMap \rightarrow BTEnv \rightarrow BTMap \times BTEnv$
$bt\llbracket\mathtt{L\ E}\rrbracket\mu_{bt}\rho_{bt} =$
  $let\ \ell = \mathcal{L}\llbracket\mathtt{L}\rrbracket\ in\ \mu_{bt}'''(\ell) = \mathsf{D} \rightarrow raise^1\ \ell\ \mu_{bt}'''\ \rho_{bt}'''\ [\,]\ (\mu_{bt}''', \rho_{bt}''')$
    $where\ (\mu_{bt}''', \rho_{bt}''') =$
      $case\ \llbracket\mathtt{E}\rrbracket\ of$
        $\llbracket\mathtt{C}\rrbracket : (upd\ \ell\ \mathsf{S}\ \mu_{bt}, \rho_{bt})$
        $\llbracket\mathtt{V}\rrbracket : (upd\ \ell\ \rho_{bt}(\mathcal{V}\llbracket\mathtt{V}\rrbracket)\ \mu_{bt}, \rho_{bt})$
        $\llbracket\mathtt{(if\ L_1E_1\ L_2E_2\ L_3E_3)}\rrbracket :$
          $let\ (\mu_{bt}', \rho_{bt}') = \bigsqcup_i bt\llbracket\mathtt{L_iE_i}\rrbracket\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu_{bt}'(\mathcal{L}\llbracket\mathtt{L_i}\rrbracket)\ in$
            $(upd\ \ell\ (b_1 = \mathsf{D} \rightarrow \mathsf{D}\ [\,]\ b_2 \sqcup b_3)\ \mu_{bt}', \rho_{bt}')$
        $\llbracket\mathtt{(let\ ((V\ L_1E_1))\ L_2E_2)}\rrbracket :$
          $let\ (\mu_{bt}', \rho_{bt}') = \bigsqcup_i bt\llbracket\mathtt{L_iE_i}\rrbracket\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu_{bt}'(\mathcal{L}\llbracket\mathtt{L_i}\rrbracket)\ in$
            $(upd\ \ell\ (b_1 = \mathsf{D} \rightarrow \mathsf{D}\ [\,]\ b_2)\ \mu_{bt}', upd\ \mathcal{V}\llbracket\mathtt{V}\rrbracket\ b_1\ \rho_{bt}')$
        $\llbracket\mathtt{(O\ L_1E_1\ \ldots\ L_nE_n)}\rrbracket :$
          $let\ (\mu_{bt}', \rho_{bt}') = (\mu_{bt}, \rho_{bt}) \sqcup \bigsqcup_i bt\llbracket\mathtt{L_iE_i}\rrbracket\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu_{bt}'(\mathcal{L}\llbracket\mathtt{L_i}\rrbracket)\ in$
            $(upd\ \ell\ (\mathsf{S} \sqcup \mathcal{O}\llbracket\mathtt{O}\rrbracket[b_1, \ldots b_n])\ \mu_{bt}', \rho_{bt}')$
        $\llbracket\mathtt{(P\ L_1E_1\ \ldots\ L_nE_n)}\rrbracket :$
          $let\ (\mu_{bt}', \rho_{bt}') = (\mu_{bt}, \rho_{bt}) \sqcup \bigsqcup_i bt\llbracket\mathtt{L_iE_i}\rrbracket\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu_{bt}'(\mathcal{L}\llbracket\mathtt{L_i}\rrbracket)\ in$
            $(upd\ \ell\ (some\ b_i = \mathsf{D} \rightarrow \mathsf{D}\ [\,]\ \mu_{bt}'(\mathcal{P}\llbracket\mathtt{P}\rrbracket))\ \mu_{bt}',$
             $\rho_{bt}' \sqcup \bigsqcup_i(upd\ (\mathcal{P}\llbracket\mathtt{P}\rrbracket, i)\ b_i\ \rho_{bt}')$
        $\llbracket\mathtt{(lambda\ (V_1\ \ldots\ V_n)\ L_1E_1)}\rrbracket :$
          $let\ (\mu_{bt}', \rho_{bt}') = bt\llbracket\mathtt{L_1E_1}\rrbracket\mu_{bt}\rho_{bt}\ in$
            $\mu_{bt}''(\ell) = \mathsf{D} \rightarrow raise^2\ \mathcal{L}\llbracket\mathtt{L_1}\rrbracket\ \mu_{bt}''\ \rho_{bt}'\ [\,]\ (\mu_{bt}'', \rho_{bt}')$
            $where\ \mu_{bt}'' = upd\ \ell\ \mathsf{Cl}\ \mu_{bt}'$
        $\llbracket\mathtt{(L_0E_0\ L_1E_1\ \ldots\ L_nE_n)}\rrbracket :$
          $let\ (\mu_{bt}', \rho_{bt}') = \bigsqcup_i bt\llbracket\mathtt{L_iE_i}\rrbracket\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu_{bt}'(\mathcal{L}\llbracket\mathtt{L_i}\rrbracket)\ in$
            $let\ acs = \{\ell' \mid \ell' \in \mu_{cl}(\mathcal{L}\llbracket\mathtt{L_0}\rrbracket) \wedge arity(\ell') = n\}\ in$
              $\mu_{bt}''(\mathcal{L}\llbracket\mathtt{L_0}\rrbracket) = \mathsf{D} \rightarrow (\mu_{bt}'', \rho_{bt}'') \sqcup \bigsqcup_{i\geq 1}(raise^2\ \mathcal{L}\llbracket\mathtt{L_i}\rrbracket\ \mu_{bt}''\ \rho_{bt}'')\ [\,]\ (\mu_{bt}'', \rho_{bt}'')$
              $where\ \mu_{bt}'' = upd\ \ell\ (some\ b_i = \mathsf{D} \rightarrow \mathsf{D}\ [\,]\ \bigsqcup_{\ell' \in acs}\mu_{bt}'(\ell'))\ \mu_{bt}'$
                    $\rho_{bt}'' = \rho_{bt}' \sqcup \bigsqcup_{i\geq 1, \ell' \in acs}(upd\ (\ell', i)\ \mu_{bt}'(\mathcal{L}\llbracket\mathtt{L_i}\rrbracket)\ \rho_{bt}')$
    $end$
```

Fig. 7. Higher order binding time analysis.

8. Results

In this section we use Similix-2 to specialize the direct style Λ-interpreter and three other Λ-interpreters: one written in continuation passing style and two implementing call-by-name semantics. All these examples were produced by Similix-2 without manual assistance, except some renaming of target program procedure names to increase readability.

8.1. Direct style

Specializing the Λ-interpreter with respect to the factorial Λ-program yields the Scheme target program given in Fig. 8. run-0 is the name of the goal procedure in the target program, i.e., run-0 computes the factorial function. We observe that the interpretation level has almost been completely removed: the interpreter's syntax analysis and environment operations have been performed. Only run time operations are left, with a small overhead due to the ext encodings. When computing factorial of 10, it is around 13 times faster to run the target program than to interpret the source program (see section 9 on performance).

```
(loadt "scheme.adt")
(loadt "lam-int.adt")

(define (run-0 w_0) ((_E-1 w_0) w_0))

(define (_E-1 r_0)
   (lambda (w_1)
      (if (ext '= w_1 0)
          1
          (ext '* w_1 ((_E-1 r_0) (ext '- w_1 1))))))
```

Fig. 8. Factorial target program, generated from direct style Λ-interpreter.

Recursion is expressed by the procedure _E-1. The redundant variable r_0 corresponds to the input variable in the source program: it is not actually referred to inside the recursive body of the letrec, but it is accessible, and this is reflected in the target program. The variable w_1 corresponds to x in the source program. Notice that the target program is in arity raised from (cf. Section 6): each source program variable (input, x) is represented by its own target program variable (r_0, w_1).

The target program can be generated either by directly specializing the Λ-interpreter with respect to the factorial program or by first generating a stand-alone compiler (using self-application) and then applying it to the factorial program.

8.2. Continuation passing style

The interpreter below (Fig. 9) can be derived from a continuation semantics for Λ. Continuations are strict and map values into values.

Binding time analysis (with static program and dynamic data input) classifies the environments reducible (Cl). The lambda expression (lambda (w1 c1)...) is classified residual (just as the corresponding lambda expression in the direct style interpreter was), and therefore the formal parameter continuation c1 also becomes

```
(loadt "scheme.adt")
(loadt "lam-int.adt")
(load "lam-aux.sim")

(extend-syntax (eta-convert) ((eta-convert c) (lambda (w) (c w))))
(extend-syntax (c-id) ((c-id) (lambda (w) w)))

(define (run E w (_E E (lambda (V) w) (c-id))))

(define (_E E r c)
  (cond
   ((isCst? E)
    (c (cst-C E)))
   ((isVar? E)
    (c (r (var-V E))))
   ((isBinop? E)
    (_E (binop-E1 E)
        r
        (lambda (w1)
          (_E (binop-E2 E) r (lambda (w2) (c (ext (binop-B E) w1 w2)))))))
   ((isIf? E)
    (_E (if-E1 E)
        r
        (lambda (w1) (if w1 (_E (if-E2 E) r c) (_E (if-E3 E) r c)))))
   ((isLambda? E)
    (c (lambda (w1 c1)
         (_E (lambda-E E) (upd (lambda-V E) w1 r) (eta-convert c1)))))
   ((isLetrec? E)
    (_E (letrec-E2 E)
        (fix (lambda (r1)
               (upd (letrec-V E) (_E (letrec-E1 E) r1 (c-id)) r)))
        c))
   ((isApply? E)
    (_E (apply-E1 E)
        r
        (lambda (w1)
          (_E (apply-E2 E) r (lambda (w2) (w1 w2 (eta-convert c)))))))
   (else
    (error '_E "unknown form: ~s" E))))
 (define (fix f) (lambda (x) ((f (fix f)) x)))
```

Fig. 9. Continuation passing style Λ-interpreter.

residual (D). The eta-conversions are inserted to make the binding time analysis classify _E's continuation parameter c reducible rather than residual. This implies that the program specializer will beta reduce continuation applications, thus giving better, more reduced target programs.

The target program in Fig. 10 is generated when specializing the interpreter with respect to the factorial program. The target program is written in continuation passing style since the interpreter was.

```
(loadt "scheme.adt")
(loadt "lam-int.adt")

(define (run-0 w_0) ((_E-1 w_0) w_0 (lambda (w_2) w_2)))
(define (_E-1 r_0)
   (lambda (w1_1 c1_2)
      (if (ext '= w1_1 0)
          (c1_2 1)
          ((_E-1 r_0) (ext '- w1_1 1)
                      (lambda (w_4) (c1_2 (ext '* w1_1 w_4)))))))
```

Fig. 10. Continuation passing style factorial target program.

Similix-2's ability to handle continuation passing style programs is important to many applications. Using continuations, it is well-known that it is straightforward to express functions that return *multiple results.* And, essential to partial evaluation, these results may have different binding times which do *not* intermingled by the binding time analysis (as they would if they were "cons'ed" together in a single result).

8.3. Call-by-name

The third interpreter (Fig. 11) is a variant of the direct style one, but it implements call-by-name semantics rather than call-by-value. Call-by-name evaluation is achieved by suspending the evaluation of arguments to applications. Instead of keeping values in environments, we thus now keep suspensions of the form (lambda ()...). Note that primitive operations are still call-by-value; only applications of lambda abstractions are call-by-name.

The following program (Fig. 12) produces a list of the first *n* even numbers. The function evens-from produces an infinite list of even numbers starting from a given number. Since lazy-cons is a lambda expression, the evaluation of its arguments is suspended and therefore calls to evens-from do not loop. Using a call-by-value interpreter, any call to evens-from would loop. The Λ-language has no let-expressions and only unary lambda expressions, so the program looks somewhat clumsy.

Specializing the call-by-name interpreter with respect to the even number program yields a target program in which syntax analysis and environment operations have

```
(loadt "scheme.adt")
(loadt "lam-int.adt")
(load "lam-aux.sim")

(extend-syntax (my-delay) ((my-delay w) (lambda () w)))
(extend-syntax (my-force) ((my-force w-delayed) (w-delayed)))

(define (run E w) (_E E (lambda (V) (my-delay w))))

(define (_E E r)
  (cond
   ((isCst? E)
    (cst-C E))
   ((isVar? E)
    (my-force (r (var-V E))))
   ((isBinop? E)
    (ext (binop-B E) (_E (binop-E1 E) r) (_E (binop-E2 E) r)))
   ((isIf? E)
    (if (_E (if-E1 E) r)
        (_E (if-E2 E) r)
        (_E (if-E3 E) r)))
   ((isLambda? E)
    (lambda (w) (_E (lambda-E E)(upd (lambda-V E) w r))))
   ((isLetrec? E)
    (_E (letrec-E2 E)
        (fix (lambda (r1)
               (upd (letrec-V E) (my-delay (_E (letrec-E1 E) r1)) r)))))
   ((isApply? E)
    ((_E (apply-E1 E) r) (my-delay (_E (apply-E2 E) r))))
   (else
    (error '_E "unknown form: ~s" E))))

(define (fix f) (lambda (x) ((f (fix f)) x)))
```

Fig. 11. Call-by-name Λ-interpreter.

```
((lambda lazy-cons
   ((lambda lazy-car
      ((lambda lazy-cdr

         (letrec first-n
            (lambda n (lambda l
               (if (= n 0)
                  '()
                  (cons (lazy-car l) ((first-n (- n 1)) (lazy-cdr l))))))
           (letrec evens-from
              (lambda n ((lazy-cons n) (evens-from (+ n 2))))
              ((first-n input) (evens-from 0)))))                ; main

       (lambda x (x (lambda a (lambda d d))))))
    (lambda x (x (lambda a (lambda d a))))))
 (lambda x (lambda y (lambda z ((z x) y)))))
```

Fig. 12. Even number program written in call-by-name Λ.

all been performed. The program contains many suspensions and is rather hard to read (we do not include it here). It is, however, quite efficient: running the target program is around 24 times faster than interpreting.

This example nicely shows the effect of partial evaluation: Scheme is call-by-value, so to achieve call-by-name evaluation, one would need to insert suspensions everywhere by hand. This is complex, so instead one can write an interpreter for a call-by-name language. However, running the interpreter gives a significant interpretation overhead. But using partial evaluation, programs in the call-by-name language are compiled into efficient Scheme code (which is eventually itself compiled).

8.4. Lazy evaluation—call-by-need

The call-by-name interpreter implements call-by-name evaluation: an argument to an application is evaluated each time it is referred to. Lazy functional programming languages such as Lazy ML (see for instance [30]) use call-by-*need* evaluation: an argument is only evaluated the first time it is referred to. Is it possible to rewrite the call-by-name interpreter to implement call-by-need evaluation?

This is in fact rather simple to do by changing the definition of `my-delay` into

```
(extend-syntax (my-delay) ((my-delay w) (save (lambda () w))))
```

where `save` is a primitive operation implemented in Scheme as follows [14]:

```
(lambda (suspension)
  (let ((thunk suspension))
    (lambda ()
      (let ((w thunk)))
        (set! thunk (lambda () w))
        w))))
```

The first time the saved lambda expression is applied (by `my-force`), the lambda expression is effectively overwritten by a new lambda expression `(lambda () w)`. Here `w` is the value E evaluates to. This achieves lazy evaluation: all subsequent "forces" will apply the suspension `(lambda () w)` rather than `(lambda () E)`, so E is evaluated at most once.

The `save` primitive involves side effects on the local variable `thunk`; Similix does not in general guarantee to handle such a primitive in a semantically correct way. It turns out, however, that `save` *is* actually handled correctly (see [3] for details), and therefore we can safely specialize the call-by-need interpreter.

Specializing the call-by-need interpreter with respect to the "evens" program yields an efficient target program (running the target program is around 11 times faster than interpreting with the lazy interpreter). Stand-alone compilers are also generated with speedups comparable to those obtained for the other interpreters. As one would expect, using call-by-need is much faster than call-by-name. This can be seen by comparing the run times in the tables given in Section 9.

In [3], a lazy curried named combinator language is compiled into Scheme by specializing an interpreter that uses `save` to implement laziness.

9. Performance

This section contains some benchmarks for Similix-2. Tables 1–5 below show the speedups achieved by partial evaluation. Each table has three columns. The first column describes the computation (job), the second one contains the run time, and the third one the speedup.

For simplicity, we identify programs with the functions they compute. Following the tradition, the program specializer is referred to as *mix*, the compiler generator as *cogen*. Binding time annotated (preprocessed) programs have the superscript *ann*. The run time figures are in CPU seconds with one or two decimals; they exclude time for garbage collection (typically 0 to 10% additional time, but in the worst case, $cogen = cogen(mix^{ann})$, more than 100%), but include postprocessing. The speedup ratios have been computed using more decimals than the ones given here. In some cases, the run time has been computed by performing 10 successive runs and then dividing. For the direct and continuation style examples, the source Λ-program is the factorial program; the figures are for 100 computations of factorial of 10. For the call-by-name and call-by-need examples, the even number program is used; the figures are for 10 computations of "evens" of 20. The system is implemented in Chez Scheme [14] version 2.0.3, and the figures are for a Sun 3/160.

Table 1 shows that running the factorial target program is around 13 times faster than interpreting the factorial source program. Compiling by the stand-alone compiler is 6 times faster than by specializing the interpreter; this shows that the partial evaluator really is effectively self-applicable. Finally, generating the compiler by the compiler generator *cogen* is 4 times faster than by specializing *mix*. Tables 2, 3 and

Table 1
Similix-2 performance, direct style Λ-interpreter example

job	time/s	speedup
$output = int(source, data)$	5.3	13.3
$output = target(data)$	0.40	
$target = mix(int^{ann}, source)$	0.36	5.9
$target = comp(source)$	0.062	
$comp = mix(mix^{ann}, int^{ann})$	13.4	3.9
$comp = cogen(int^{ann})$	3.4	

Table 2
Similix-2 performance, continuation passing style Λ-interpreter example

job	time/s	speedup
$output = int(source, data)$	5.8	13.7
$output = target(data)$	0.42	
$target = mix(int^{ann}, source)$	0.55	5.1
$target = comp(source)$	0.11	
$comp = mix(mix^{ann}, int^{ann})$	47.3	3.2
$comp = cogen(int^{ann})$	14.6	

Table 3
Similix-2 performance, call-by-name A-interpreter example

job	time/s	speedup
$output = int(source, data)$	32.5	24.2
$output = target(data)$	1.3	
$target = mix(int^{ann}, source)$	2.8	2.9
$target = comp(source)$	0.98	
$comp = mix(mix^{ann}, int^{ann})$	16.8	3.6
$comp = cogen(int^{ann})$	4.7	

4 are similar. Table 5 shows that generating *cogen* by running *cogen* is 3 times faster than by specializing *mix*.

Here are some additional figures: it takes 2-4 seconds to preprocess one A-interpreter (includes closure ad binding time analyses); preprocessing *mix* takes around 37 seconds. The size of *mix* is 2.3 K cells (measured as the number of "cons" cells needed to represent the program as a list), *cogen* 17.2 K cells, the interpreters 0.18 K-0.26 K cells, and the compilers 1.3 K-4.3 K cells. For *mix*, this gives an expansion factor of 7.4 (17.2/2.3), for the interpreters it gives factors in the range 7.3-16.6.

The figures all in all compare well to similar published benchmarks for first order languages [23, 7, 10], and also to those of Lambda-mix [17].

10. Currying

It was mentioned earlier (Section 4.2) that uncurrying functions sometimes gives better specialization. If a curried expression is always applied to all its arguments

Table 4
Similix-2 performance, call-by-name A-interpreter example

job	time/s	speedup
$output = int(source, data)$	5.4	10.8
$output = target(data)$	0.50	
$target = mix(int^{ann}, source)$	2.9	2.8
$target = comp(source)$	1.0	
$comp = mix(mix^{ann}, int^{ann})$	17.1	3.6
$comp = cogen(int^{ann})$	4.8	

Table 5
Similix-2 performance, compiler generator example

job	time/s	speedup
$cogen = mix(mix^{ann}, mix^{ann})$	247.	2.7
$cogen = cogen(mix^{ann})$	92.	

simultaneously, then it is beneficial to use an uncurried version. In the uncurried version, the binding time values of the parameters do not influence each other. Uncurrying thus prevents a possible loss of static information.

When binding time analysing a curried expression such as E =

```
(lambda (x) (lambda (y) (+ (+ y y) x)))
```

the binding time analysis might annotate x dynamic and y static. That is, a dynamic argument is supplied before a static argument. During program specialization, an application of E like `((E "code") 3)` could be beta reduced to the residual code piece `(+6 "code")`.

However, to avoid duplicating or discarding computations, let-expressions are inserted for all formal lambda expression parameters in source programs before preprocessing [7, 4]. The expression which is actually binding time analysed is therefore not E, but the semantically equivalent.

```
(lambda (x) (let ((x x)) (lambda (y) (let ((y y)) (+ (+ y y) x)))))
```

The outer let-expression `(let ((x x)) ...)` ensures that x is not discarded, even if the the body of `(lambda (x) ...)` is never applied.

Now, since x is dynamic, the result of the body of the outer lambda expression becomes dynamic. The (value of the) inner lambda expression is a possible result of evaluating this body, and therefore the inner lambda expression becomes annotated residual (D). Hence, its parameter y becomes dynamic whereby static information is lost. It thus cannot happen that dynamic arguments are supplied before static arguments. This confirms the intuition in [29]: "early bindings before late bindings".

11. Related work

11.1. Lambda-mix

The first autoprojector for a higher order functional language is (to our knowledge) *Lambda-mix* [17, 21]. Lambda-mix treats the untyped call-by-value lambda calculus with an explicit fixed point operator `fix`. The system is surprisingly simple and easy to understand; even the generated compilers are small and readable (which is quite uncommon for compilers generated by autoprojectors!). Lambda-mix's treatment of recursion is, however, quite limited; as a consequence, many interesting programs cannot be specialized well by Lambda-mix.

The problem is that Lambda-mix does not generate named specialized program points. In Similix, code for the same specialized version `foo-x` of `foo` is only generated once. The specialized procedure `foo-x` may be referred to by its name (`foo-x`), also recursively. This is the (only) way recursive code appears in residual programs.

Lambda-mix does not generate named residual program points and therefore cannot generate residual recursive code in this way. In fact, the only way Lambda-mix can produce a residual loop is by leaving a `fix` residual. But when Lambda-mix suspends the `fix`-operation in an expression `fix` λ`f`.λ`x`.E, the variable `x` becomes dynamic.

This implies that Lambda-mix cannot handle non-inductive static parameters to recursive functions properly (a parameter is *inductive* if it becomes smaller for each recursion, according to some well-founded ordering). If `fix` is unfolded, unfolding will not terminate since the parameter is not inductive. But if `fix` is suspended, the parameter (`x` above) becomes dynamic and thus cannot be processed at program specialization time.

Non-inductive parameters are often used in practice, for instance when using partial evaluation to generate string pattern matchers as described in [13]. The Λ-interpreters' recursive environments are another example: these are partially static structures which are not inductive either; if Lambda-mix were used to specialize these interpreters, environments would have to be considered dynamic to make program specialization terminate.

An interesting aspect of Lambda-mix is its new type inference based binding time analysis [18]. The analysis differs from Nielson and Nielson's in that it handles the untyped lambda calculus and thus also treats untyped programs. Contrasting to Nielson and Nielson's analysis, it has only one dynamic type called "Code". A closer study of the relation between Gomard's type inference based analysis and our abstract interpretation based one is currently being performed [1].

11.2 Schism

In [11] a higher order abstract interpretation based binding time analysis is presented. This analysis does not perform a separate closure analysis before the binding time analysis, but instead it integrates the two. The analysis is used in a higher order extension of the partial evaluator Schim [10]. This extended version of Schism handles both partially static structures [28] and higher order constructs.

12. Conclusion

We have presented an approach to treat a higher order subset of Scheme in autoprojection. We have implemented the ideas by extending Similix-1. To our knowledge, our system is the first fully automated and implemented autoprojector for a higher order language. We have presented a binding time analysis based on a closure analysis. The binding time lattice is finite and no structured binding time values are needed.

We have shown examples of interpreters from which target programs and stand-alone compilers were generated. Because the language is higher order, we are able to treat continuation passing style interpreters and interpreters that use suspensions to implement call-by-name and call-by-need reduction.

One problem is that the binding time analysis is *monovariant*, i.e., it only generates one binding time annotated version of each procedure. If a procedure is called with different binding time patterns, the least upper bound is taken. This gives a possible loss of static information at program specialization time. It is not clear how to extend the closure analysis based binding time analysis to a polyvariant one.

For future work, the system should be applied to larger and more realistic examples. One application is described in [24]: here Similix-2 is used to generate a compiler for a large lazy functional language.

Acknowledgements

This work has been carried out within the Similix project, a joint work of Olivier Danvy and the author. I am most grateful to Olivier for his many comments and proposals. Many other people have contributed in various ways; thanks to Lars Ole Andersen, Hans Dybkjær, Frank Frauendorf, Harald Ganzinger, Carsten K. Gomard, Chris Hankin, Kristoffer H. Holm, Neil D. Jones, Jesper Jõrgensen, Karoline Malmkjær, Torben Mogensen, Peter Sestoft, and Jörg Süggel.

References

[1] L.O. Andersen and C. Mossin, Binding time analysis via type inference. Student Report 90-10-12, DIKU, University of Copenhagen, Denmark, Oct. 1990.

[2] A. Bondorf, Automatic autoprojection of higher order recursive equations, in: N.D. Jones, ed., *ESOP'90, 3rd European Symposium on Programming*, Copenhagen, Denmark, May 1990, Lecture Notes in Computer Science **432** (Springer, Berlin, 1990) 70-87.

[3] A. Bondorf, Compiling laziness by partial evaluations, in: S.L Peyton Jones, G. Hutton and C. Kehler Holst, eds., *Functional Programming, Glasgow 1990. Workshops in Computing* (Springer, Berlin, 1990) 9-22.

[4] A. Bondorf, Self-Applicable Partial Evaluation (revised version), PhD thesis, DIKU, University of Copenhagen, Denmark, Dec. 1990, DIKU report 90-17.

[5] A. Bondorf, A self-applicable partial evaluator for term rewriting systems, in: J. Diaz and F. Orejas, eds., *TAPSOFT'89. Proc. International Joint Conference on Theory and Practice of Software Development*, Barcelona, Spain, March 1989, Lecture Notes in Computer Science **352** (Springer, Berlin, 1989).

[6] A. Bondorf and O. Danvy, Automatic autoprojection of recursive equations with global variables and abstract data types, Technical Report 90-4, DIKU, University of Copenhagen, Denmark, 1990.

[7] A. Bondorf and O. Danvy, Automatic autoprojection of recursive equations with global variables and abstract data types, *Sci. Comput. Programming* 16 (1991) 151-195. Journal version of [6].

[8] A. Bondorf, N.D. Jones, T. Æ. Mogensen and P. Sestoft, Binding time analysis and the taming of self-application, Draft, 1988, DIKU, University of Copenhagen, Denmark.

[9] M.A. Bulyonkov, Polyvariant mixed computation for analyzer programs, *Acta Inform.* **21** (1984) 473-484.

[10] C. Consel, Analyse de programmes, Evaluation partielle et Génération de compilateurs, PhD thesis, LITP, University of Paris 6, France, June 1989.

[11] C. Consel, Binding time analysis for higher order untyped functional languages, *Proc. 1990 ACM Conference on Lisp and Functional Languages*, Nice, France, June 1990, 264-272.

[12] C. Consel, New insights into partial evaluation: the SCHISM experiment, in: H. Ganzinger, ed., *ESOP'88, 2nd European Symposium on Programming*, Nancy, France, March 1988, Lecture Notes in Computer Science **300** (Springer-Berlin, 1988) 236-247.

[13] C. Conel and O. Danvy, Partial evaluation of pattern matching in strings, *Inform. Process. Lett.* **30**(2) (1989) 79-86.

[14] R.K. Dybvig, *The SCHEME Programming Language* (Prentice-Hall, Englewood Cliffs, NJ, 1987).

[15] A.P. Ershov, On the partial computation principle, *Inform. Process. Lett.* **6**(2) (1977) 38-41.

[16] Y. Futamura, Partial evaluation of computing process—an approach to a compiler-compiler, *Systems, Computers, Controls* **2**(5) (1971) 45-50.

[17] C.K. Gomard, Higher Order Partial Evaluation—HOPE for the Lambda Calculus, Master's thesis, DIKU, University of Copenhagen, Denmark, Student Report 89-9-11, Sept. 1989.

[18] C.K. Gomard, Partial type inference for untyped functional programs, *Proc. 1990 ACM Conference on Lisp and Functional Languages*, Nice, France, June 1990, 282-287.

[19] C.K. Gomard and N.D. Jones, Compiler generation by partial evaluation: a case study, *Proc. Twelfth IFIP World Computer Congress*, 1989.

[20] T. Johnsson, Lambda lifting: transforming programs to recursive equations, in: J.-P. Jouannaud, ed., *Proc. Conference on Funtional Languages and Computer Architecture*, Nancy, France, Sept. 1985, Lecture Notes in Computer Science **201** (Springer, Berlin, 1985) 190-203.

[21] N.D. Jones, C.K. Gomard, A. Bondorf, O. Danvy and T. Æ. Mogensen, A self-applicable partial evaluator for the lambda calculus, *Proc. IEEE Computer Society 1990 International Conference on Computer Languages*, March 1990, 49-58.

[22] N.D. Jones, P. Sestoft and H. Søndergaard, An experiment in partial evaluation: the generation of a compiler generator, in: J.-P. Jouannaud, ed., *Rewriting Techniques and Applications*, Dijon, France, Lecture Notes in Computer Science **202** (Springer, Berlin, 1985) 124-140.

[23] N.D. Jones, P. Sestoft and H. Söndergaard, MIX: a self-applicable partial evaluator for experiments in compiler generation, *LISP and Symbolic Computation* 2(1) (1989) 9-50.

[24] J. Jørgensen and L. Mathiesen, Generating a compiler for a lazy functional language, Student Report 90-5-16, DIKU, University of Copenhagen, Denmark, Nov. 1990.

[25] E.E. Kohlbecker, Syntactic Extensions in the Programming Language Lisp, PhD thesis, Indiana University, Bloomington, 1986.

[26] T.Æ. Mogensen, Binding time analysis for polymorphically typed higher order languages, in: J. Diaz and F. Orejas, eds., *TAPSOFT'89, Proc. International Joint Conference on Theory and Practice of Software Development*, Barcelona, Spain, March 1989, Lecture Notes in Computer Science **352** (Springer, Berlin, 1989) 298-312

[27] T.Æ. Mogensen, Binding Time Aspects of Partial Evaluation, PhD thesis, DIKU, University of Copenhagen, Denmark, March 1989.

[28] T.Æ. Mogensen, Partially static structures in a self-applicable partial evaluator, in: D. Bjõrner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 325-347.

[29] H.R. Nielson and F. Nielson, Automatic binding time analysis for a typed λ-calculus, *Proc. Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Diego, CA, 1988, 98-106.

[30] S.L. Peyton Jones, *The Implementation of Functional Programming Languages. Computer Science* (Prentice-Hall, Englewood Cliffs, NJ, 1987).

[31] J. Rees and W. Clinger, Revised report^3 on the algorithmic language Scheme, *Sigplan Notices* **21**(12) (1986) 37-79.

[32] S.A. Romanenko, A compiler generator produced by a self-applicable specialiser can have a surprisingly natural and understandable structure, in: D. Bjõrner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 445-463.

[33] P. Sestoft, Automatic call unfolding in a partial evaluator, in: D. Bjõrner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 485-506.

[34] P. Sestoft, Replacing Function Parameters by Global Variables, Master's thesis, DIKU, University of Copenhagen, Denmark, Student Report 88-7-2, Oct. 1988.
[35] P. Sestoft, Replacing function parameters by global variables, *Proc. Fourth International Conference on Functional Programming and Computer Architecture*, London, UK, Sept. 1989 (ACM Press, New York and Addison-Wesley, Reading, MA, 1989) 39-53.
[36] V.F. Turchin, The concept of a supercompiler, *Trans. Program. Lang. Syst.* **8**(3) (1986) 292-325.
[37] V.F. Turchin, Semantic definitions in Refal and the automatic production of compilers, in: N.D. Jones, ed., *Workshop on Sematics-Directed Compiler Generation*, Århus, Denmark, Jan. 1980, Lecture Notes in Computer Science **94** (Springer, Berlin, 1980) 441-474.