Automatic Autoprojection of Higher Order Recursive Equations

Anders Bondorf
DIKU, University of Copenhagen*
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
e-mail: anders@diku.dk

Abstract

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* [Bondorf & Danvy 90] 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.

Given a source program and some, but not all of its inputs, partial evaluation produces a residual program. When applied to the rest of the inputs, the residual program yields the same result as the source program would when applied to all inputs. One important application of autoprojection is semantics directed compiler generation: given a denotational, interpretive specification of a programming language, it is possible automatically to generate a stand-alone compiler by self-applying the partial evaluator.

Efficient autoprojection is known to require *binding time analysing* source programs to be partially evaluated. Binding time analysis establishes in advance which parts of the source program that can be evaluated during partial evaluation and which parts that cannot. We describe a new automatic binding time analysis for higher order programs written in the Scheme subset. The analysis requires no type information. It is based on a *closure analysis* [Sestoft 88b], 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 three versions of an interpreter for a lambda calculus language: one written in direct style, one written in continuation passing style, and one implementing normal order reduction. The two latter are heavily based on higher order programming.

Keywords

Partial evaluation, self-application, binding time analysis, semantics directed compiler generation.

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 to all of the input. *Autoprojection* is a synonym for *self-applicable* partial evaluation, 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: specializ-

* This work was carried out while the author was staying at the University of Dortmund, Lehrstuhl Informatik V, Postfach 50 05 00, D-4600 Dortmund 50, Federal Republic of Germany. The work was partly supported by the Danish Research Academy.

ing a partial evaluator with static input being (the text of) an *interpreter* for some programming language S yields a *compiler* for S [Futamura 71] [Ershov 77] [Turchin 80]. 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* [Jones, Sestoft, & Søndergaard 85]. The language treated by Mix was a subset of statically scoped first order pure Lisp, and Mix was able to generate compilers out of interpreters written in this language. The experiment showed that autoprojection was possible in practice; an automatic version of Mix was developed later [Jones, Sestoft, & Søndergaard 89]. Since then, autoprojectors for several languages have been implemented: for a subset of Turchin's Refal language [Romanenko 88], for an imperative flowchart language [Gomard & Jones 89], for pattern matching based programs in the form of restricted term rewriting systems [Bondorf 89], and for first order functional languages with global variables [Bondorf & Danvy 90].

1.1 Autoprojecting higher order languages

The first autoprojector for a *higher order* functional language is (to our knowledge) *Lambda-mix* [Gomard 89] [JonGomBonDanMog 90]. Lambda-mix treats the untyped call-by-value lambda calculus. 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!). However, a strong limitation in Lambda-mix is that static parameters of recursive functions *must* be induction variables [Aho, Sethi, & Ullman 86]; non-inductive variables are always dynamic. Lambda-mix thus does not *specialize* (recursive) calls to equal functions with equal patterns of static argument values (known as *polyvariant* program specialization [Bulyonkov 84]). Specialization is a kind of *folding* and thus gives *sharing* of functions in residual programs. Since Lambda-mix does not specialize calls, it does not perform well for some applications. For instance, when using partial evaluation to generate string pattern matchers [Consel & Danvy 89], non-inductive recursive static variables are used.

In this paper we describe an implemented automatic autoprojector, *Similix-2*, that handles a higher order subset of the Scheme language [Rees & Clinger 86], essentially weakly (dynamically) typed, statically scoped, call-by-value recursive equations with lambda abstractions and applications. Similix-2 uses polyvariant program specialization and is, to our knowledge, the first fully automated and implemented autoprojector for a higher order language (Lambda-mix requires hand-written binding time annotations; an automatic version has been developed later [Gomard 90]).

Similix-2 has been developed and implemented by extending *Similix* [Bondorf & Danvy 90], an existing autoprojector for a first order Scheme subset. Similix-2 therefore has a number of features that have been inherited from Similix: side-effecting operations on global variables (such as i/o operations) are treated in a semantically correct way; primitives are user specified (as introduced in [Consel 88]), i.e. there is no fixed set of primitives; residual programs never duplicate computations (cf. *call duplication* [Sestoft 88a]); residual programs do not terminate more often than source programs; call unfolding is controlled automatically (by a simple strategy based on detecting *dynamic conditionals*). This paper does not cover these aspects which all come from Similix; we refer to [Bondorf & Danvy 90] (and to [Bondorf 90]).

Treating higher order languages opens new perspectives for using autoprojection for semantics directed compiler generation: Similix-2 treats interpreters written in continuation passing style and interpreters that implement (weak head) normal order reduction (outside-in, call-by-name) by "thunks" of the form `(λ () E)`. We exemplify this by specializing such interpreters. To our knowledge, this is the first time autoprojection has been used to generate compilers from interpreters of that kind.

1.2 Outline

The rest of the paper is organized as follows. Section 2 gives some background and introduces the main issues of this paper. In section 3, 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. The sections 4 and 5 contain the technical part: we develop/describe two pre-analyses needed for partial evaluation of higher order programs; both analyses are presented formally in a compositional denotational semantics style. In section 6, we exemplify partial evaluation of higher order programs: the Λ-interpreter is specialized, and we also specialize two other Λ-interpreters: one written in continuation passing style and one implementing normal order reduction. The performance of the system is shown in section 7. Section 8 is a discussion, and in section 9 we conclude and sketch some open problems.

1.3 Prerequisites

Some knowledge about partial evaluation is required, e.g. as presented in [Jones, Sestoft, & Søndergaard 85] or [Jones, Sestoft, & Søndergaard 89].

2. Background and Issues

2.1 Preprocessing

An essential component of an autoprojector is the *preprocessor*. Preprocessing is performed *before* program specialization; its purpose is to add *annotations* (attributes) to the source program [Jones, Sestoft, & Søndergaard 85]. The annotations guide the program specializer (which actually produces the residual program) in various ways: they tell whether variables are bound to static or dynamic values, whether operations such as + or `if` can be reduced away during program specialization, and whether certain expressions (function calls, let-expressions) should be *unfolded*. Annotations 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 [Bondorf, Jones, Mogensen, & Sestoft 90] (the essential reason: the static input to the program with respect to which the specializer is being specialized is not available). Without using annotations, the generated compilers would become unnecessarily general and hence large (in code size) and slow. All autoprojectors we know of use preprocessing and annotations.

The central preprocessing phase is *binding time analysis* [Jones, Sestoft, Søndergaard 89]. Binding time analysis is an *approximative* analysis that *abstractly interprets* the program over a binding time domain, in the simplest case the two-point lattice *Static* $\sqsubseteq$ *Dynamic*. *Static* is to be interpreted as "definitely static", i.e. it abstracts values that are available (*known*) at program specialization time. *Dynamic* means "possibly non-static" and abstracts values that are possibly not available (*unknown*) at program specialization time. Variables and operations are then classified according to their binding times. As a simple example, the operation + in the expression `(+ x 1)` is static (*eliminable*, *compile time*) if `x` is classified *Static*, and it is dynamic (*residual*, *run time*) if `x` is classified *Dynamic*. Static operations are evaluated during program specialization whereas residual code is generated for the dynamic ones. Static operations correspond to the overlined ones of [Nielson & Nielson 88], dynamic operations to the underlined ones.

2.2 Binding time analysing higher order programs

Nielson and Nielson have described an automatic binding time analysis for a *higher order* functional language [Nielson & Nielson 88]. Their analysis treats the *typed* lambda calculus. Mogensen has described an analysis for a polymorphically typed higher order functional language where programs are written in curried named combinator form [Mogensen 89b]. In these two papers, no autoprojector is developed; only binding time analysis is addressed. An automatic binding time analysis for Lambda-mix has been developed recently [Gomard 90].

The difficult point in binding time analysing higher order programs is to associate lambda abstractions with applications. If a program e.g. contains the application `(x y)`, and if `x` during (partial) evaluation may be bound to (the value of) some abstraction, say `(λ (z) (+ z 3))`, occurring elsewhere in the program, then the binding time value of `y` influences the one of `z`. If, for instance, `y` is classified *Dynamic*, then `z` cannot be *Static*. On the other hand, if `x` can never possibly be bound to (the value of) `(λ (z) (+ z 3))`, then it is unnecessarily conservative to let y influence 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 that y influences z? Nielson & Nielson use a *type inferencing* scheme (using the type information in the program); from the expression `(x y)`, it would identify that x had type *Dynamic* → ... (since y is *Dynamic*), and eventually this type would be unified (using least upper bounds) with the type of `(λ (z) (+ z 3))`. Its type is $Z \rightarrow \ldots$, where Z is the type of z. Unifying the types implies $Z \sqsupseteq$ *Dynamic*, i.e. z's binding time value is at least *Dynamic*. 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.

In this paper we present a different approach based on a variant of Sestoft's *closure analysis* [Sestoft 88b] [Sestoft 89]: a closure analysis is first performed, then the binding time analysis is performed:

bt-annotations = bt-analyse(P, 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) `(λ (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 (cf. the conservative extreme mentioned above).

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 particular, no *structured* binding time values (such as *Dynamic* → *Dynamic*) 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.

2.3 Programming language

Similix-2 processes higher order recursive equations. The language is an extension of the one treated by Similix [Bondorf & Danvy 90]: lambda abstractions and applications have been added to the allowed expression forms. As for Similix, programs follow the syntax of Scheme and are thus 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. A Scheme procedure corresponds to a *function* in references such as [Jones, Sestoft, & Søndergaard 89]. 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 the internal operations performed by a primitive operator. It can only do two things with a primitive operation: either evaluate the operation or suspend it generating residual code.

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

Abstract syntax of the Scheme subset treated by Similix-2

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

$\texttt{Pr ::= (loadt F)}^* \texttt{ (load F)}^* \texttt{ PD}^+$

$\texttt{PD ::= (define (P V}^*\texttt{) L-E)}$

$\texttt{L-E ::= L E}$

$\texttt{E ::= C | V | (if L-E}_1\texttt{ L-E}_2\texttt{ L-E}_3\texttt{)}$

```
| (let ((V L-E1)) L-E2)
| (O L-E*) | (P L-E*)
| (λ (V*) L-E1) | (L-E0 L-E*)
```

Notation: program texts and names of syntactic domains `are written in this font`. We write λ instead of `lambda` in program texts.

The primitive operators are defined in external files referred to by the `loadt` expressions. Definitions from other files can be reused using `load`. An expression is a constant (boolean, number, string, or quoted construction), a variable, a conditional, a let-expression (unary for simplicity), a primitive operation, a procedure call, a lambda abstraction, or an application; the latter two forms make the language higher order. The order of evaluation is applicative (strict, call-by-value, inside-out), and arguments are evaluated in an unspecified order.

We note that let-expressions are *not* considered syntactic sugar for applications of (higher order) lambda abstractions: this would not be beneficial since let-expressions are first order and thus simpler to deal with. Procedure calls are treated in another way than higher order applications; the two forms are therefore distinguished syntactically. Both procedure calls and higher order applications are, in turn, distinguished from applications of primitive operators. The distinctions are made during parsing. To keep the language simple, there is no `letrec` (nor any `rec`); recursion is expressed using named procedures.

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

2.4 Syntactic extensions

A number of built-in syntactic extensions are treated by Similix. We mention one which is used in the examples later: `cond`. It is expanded into nested `if` expressions. The system also treats user defined syntactic extensions following the syntax of [Kohlbecker 86] (only a subset of Kohlbecker's language is treated).

3. A sample interpreter

In this section we 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 this (abstract) syntax:

<u>Abstract syntax of Λ</u>

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

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

A program takes one input value, which initially is bound to all variables (for simplicity). For an example, this program computes the factorial function:

<u>Factorial program written in Λ</u>

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

The (arbitrary) variable name `input` is used to refer to the input value.

3.1 Denotational semantics

The denotational semantics of the language is specified below. We use the notation from [Schmidt 86]; the conditional is written $\_ \rightarrow \_ [] \_$, and $[v \mapsto w]r$ is shorthand for $\lambda v1.v{=}v1 \rightarrow w [] r(v1)$.

<u>Denotational semantics of Λ</u>

Semantic domains:

w ∈ Value, r ∈ Environment = `Var` → Value

Valuation functions:

run: $\texttt{Expr} \rightarrow \text{Value} \rightarrow \text{Value}$

$\text{run}[\![\texttt{E}]\!]\text{w} = \mathcal{E}[\![\texttt{E}]\!]\ \lambda\texttt{v}.\texttt{w}$

$\mathcal{E}$: $\texttt{Expr} \rightarrow \text{Environment} \rightarrow \text{Value}$

$\mathcal{E}[\![\texttt{c}]\!]\text{r} = \mathcal{C}[\![\texttt{c}]\!]$

$\mathcal{E}[\![\texttt{v}]\!]\text{r} = \text{r}([\![\texttt{v}]\!])$

$\mathcal{E}[\![\texttt{(B E}_1\ \texttt{E}_2\texttt{)}]\!]\text{r} = \mathcal{B}[\![\texttt{B}]\!]\ (\mathcal{E}[\![\texttt{E}_1]\!]\text{r})\ (\mathcal{E}[\![\texttt{E}_2]\!]\text{r})$

$\mathcal{E}[\![\texttt{(if E}_1\ \texttt{E}_2\ \texttt{E}_3\texttt{)}]\!]\text{r} =$
$\quad\mathcal{E}[\![\texttt{E}_1]\!]\text{r} \rightarrow \mathcal{E}[\![\texttt{E}_2]\!]\text{r} \mathbin{[\!]} \mathcal{E}[\![\texttt{E}_3]\!]\text{r}$

$\mathcal{E}[\![\texttt{(}\lambda\ \texttt{V E)}]\!]\text{r} = \lambda\texttt{w}.\mathcal{E}[\![\texttt{E}]\!][[\![\texttt{V}]\!]\mapsto\texttt{w}]\text{r}$

$\mathcal{E}[\![\texttt{(letrec V E}_1\ \texttt{E}_2\texttt{)}]\!]\text{r} =$
$\quad\mathcal{E}[\![\texttt{E}_2]\!]\ \mathit{fix}(\lambda\text{r1}.[[\![\texttt{V}]\!]\mapsto\mathcal{E}[\![\texttt{E}_1]\!]\text{r1}]\text{r})$

$\mathcal{E}[\![\texttt{(E}_1\ \texttt{E}_2\texttt{)}]\!]\text{r} = (\mathcal{E}[\![\texttt{E}_1]\!]\text{r})\ (\mathcal{E}[\![\texttt{E}_2]\!]\text{r})$

$\mathcal{C}$: $\texttt{Const} \rightarrow \text{Value}$ *unspecified*

$\mathcal{B}$: $\texttt{Binop} \rightarrow \text{Value} \rightarrow \text{Value} \rightarrow \text{Value}$ *unspecified*

No type checking is performed; this would require injection tags on values and has been omitted for simplicity.

3.2 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 6.3, we show an interpreter that defines a non-strict semantics.

To translate the semantics into Scheme, we first uncurry the functions run, $\mathcal{E}$, and $\mathcal{B}$; this is simple since the functions already are used in an uncurried way. Uncurrying is advantageous from a readability point of view (`(f x y)` contra `((f x) y)`), and it also sometimes gives better specialization (more about this in section 8.2).

We now give the interpreter text. $\mathcal{C}$ is just the identity function and has been omitted.

Direct style Λ-interpreter written in Scheme

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

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

(define (_E E r)
  (cond
   ((isCst? E)
    (cst-C E))
   ((isVar? E)
    (r (var-V E)))                           ;p
   ((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)
    (λ (w)                                   ;2
      (_E (lambda-E E)
          (upd (lambda-V E) w r))))          ;1,q
   ((isLetrec? E)
    (_E (letrec-E2 E)
        (fix (lambda (r1)                    ;4
          (upd (letrec-V E)                  ;3,r
               (_E (letrec-E1 E) r1)
               r)))))
   ((isApply? E)
    ((_E (apply-E1 E) r)                     ;s
     (_E (apply-E2 E) r)))
   (else
    (error '_E "unknown form: ~s" E))))

(define (fix f)
  (λ (x) ((f (fix f)) x)))                   ;5,t,u
```

The comments (`0-5` and `p-u`) are used for reference later (section 4.5).

Syntax accessors (such as `letrec-E1`), syntax predicates (such as `isLambda?`), and `ext` have been defined as primitive operations in the file `"lam-int.adt"`. The standard Scheme primitives `equal?` and `error` are defined in `"scheme.adt"`. The file `"lam-aux.sim"` defines environment updating as a syntactic extension:

Environment updating

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

3.3 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).

What can be expected from binding time analysing the interpreter? Λ is statically scoped, so e.g. environment operations should be classified eliminable: they can be performed at partial evaluation 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 eliminable.

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.

4. Closure analysis

In this section, we give a formal presentation of the closure analysis. The purpose of the analysis is for any application point to collect the set of possible (values of) lambda abstractions that may be applied at that point.

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 [Sestoft 88b] [Sestoft 89]. 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 also mentions this possible extension).

We describe the analysis in a different and more implementation suitable way than Sestoft's. Our approach is based on the idea of continuously updating global mappings: traversing a program expression does not result in a "value", but in updated global mappings. 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. A global mapping corresponds to what is called a *cache* in [Hudak & Young 88]: it associates every expression in the program with a value (this explains the need for expression labels in the abstract syntax).

4.1 Semantic domains and functions

We define some semantic domains and various utility functions used by the closure analysis. First, we need some (injective) functions for converting from syntactic to semantic domains:

$\mathcal{L}$: `Label` $\rightarrow$ Label

$\mathcal{V}$: `Variable` $\rightarrow$ Variable

$\mathcal{P}$: `ProcedureName` $\rightarrow$ Label

$\mathcal{P}$ associates a procedure name with the label of the procedure body. The semantic domains are defined like this:

Index = $\{1, 2, \ldots\}$

$k \in$ Label = *unspecified*

$v \in$ Variable = Label $\times$ Index

Formal parameters to a procedure `P` are identified as (k, 1), (k, 2), etc., where $k = \mathcal{P}[\![\texttt{P}]\!]$. Formal parameters to a lambda expression with body expression with label `L` are identified similarly ($k = \mathcal{L}[\![\texttt{L}]\!]$). Note that these identifications are unique. The formal parameter `V` of a let-expression is associated with some arbitrary unique value v.

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

Closure = Label

$c \in$ ClSet = $\wp$(Closure)

$\mu_{cl} \in$ ClMap = Label $\rightarrow$ ClSet

$\rho_{cl} \in$ ClEnv = Variable $\rightarrow$ ClSet

Maps and environments are updated by corresponding monotonic update functions. Map updating is performed by the function upd (which should not be confused with the `upd` in the Λ-interpreter):

upd: Label $\rightarrow$ ClSet $\rightarrow$ ClMap $\rightarrow$ ClMap

upd k c $\mu_{cl} = \mu_{cl} \sqcup [k \mapsto c]\bot_{ClMap}$

Environment updating has functionality

Variable $\rightarrow$ ClSet $\rightarrow$ ClEnv $\rightarrow$ 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 bounds on functions and cartesian products are defined pointwise:

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

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

Finally, we need a function for checking the arity of a closure:

arity: Closure $\rightarrow \{0, 1, 2, \dots\}$

4.2 The analysis

We now give the closure analysis rules. Given a set of procedure definitions, 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 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. A *case* expression is used for syntax dispatching.

<u>Closure analysis</u>

```
$\mathrm{Cl}$: Definition$^+ \rightarrow \mathrm{ClMap} \times \mathrm{ClEnv}$
$\mathrm{Cl}\llbracket$(define (...) $L_1E_1$) ... (define (...) $L_nE_n$)$\rrbracket = \mathit{fix}(\lambda(\mu_{cl}, \rho_{cl}) \,.\, \bigsqcup_i \mathrm{cl}\llbracket L_iE_i \rrbracket \mu_{cl}\rho_{cl})$
$\mathrm{cl}$: LabeledExpression $\rightarrow \mathrm{ClMap} \rightarrow \mathrm{ClEnv} \rightarrow \mathrm{ClMap} \times \mathrm{ClEnv}$
$\mathrm{cl}\llbracket L\ E \rrbracket \mu_{cl}\rho_{cl} =$
  let $k = \mathcal{L}\llbracket L \rrbracket$ in
    case $\llbracket E \rrbracket$ of
      $\llbracket C \rrbracket$: (upd k $\{\}$ $\mu_{cl}$, $\rho_{cl}$)
      $\llbracket V \rrbracket$: (upd k $\rho_{cl}(\mathcal{V}\llbracket V \rrbracket)$ $\mu_{cl}$, $\rho_{cl}$)
      $\llbracket$(if $L_1E_1$ $L_2E_2$ $L_3E_3$)$\rrbracket$: let $(\mu'_{cl}, \rho'_{cl}) = \bigsqcup_i \mathrm{cl}\llbracket L_iE_i \rrbracket \mu_{cl}\rho_{cl}$ in
        (upd k $(\mu'_{cl}(\mathcal{L}\llbracket L_2 \rrbracket) \sqcup \mu'_{cl}(\mathcal{L}\llbracket L_3 \rrbracket))$ $\mu'_{cl}$, $\rho'_{cl}$)
      $\llbracket$(let ((V $L_1E_1$)) $L_2E_2$)$\rrbracket$: let $(\mu'_{cl}, \rho'_{cl}) = \bigsqcup_i \mathrm{cl}\llbracket L_iE_i \rrbracket \mu_{cl}\rho_{cl}$ in
        (upd k $\mu'_{cl}(\mathcal{L}\llbracket L_2 \rrbracket)$ $\mu'_{cl}$, upd $\mathcal{V}\llbracket V \rrbracket$ $\mu'_{cl}(\mathcal{L}\llbracket L_1 \rrbracket)$ $\rho'_{cl}$)
      $\llbracket$(O $L_1E_1$ ... $L_nE_n$)$\rrbracket$: let $(\mu'_{cl}, \rho'_{cl}) = (\mu_{cl}, \rho_{cl}) \sqcup \bigsqcup_i \mathrm{cl}\llbracket L_iE_i \rrbracket \mu_{cl}\rho_{cl}$ in
        (upd k $(\bigsqcup_i \mu'_{cl}(\mathcal{L}\llbracket L_i \rrbracket))$ $\mu'_{cl}$, $\rho'_{cl}$)
      $\llbracket$(P $L_1E_1$ ... $L_nE_n$)$\rrbracket$: let $(\mu'_{cl}, \rho'_{cl}) = (\mu_{cl}, \rho_{cl}) \sqcup \bigsqcup_i \mathrm{cl}\llbracket L_iE_i \rrbracket \mu_{cl}\rho_{cl}$ in
        (upd k $\mu'_{cl}(\mathcal{P}\llbracket P \rrbracket)$ $\mu'_{cl}$, $\rho'_{cl} \sqcup \bigsqcup_i$(upd $(\mathcal{P}\llbracket P \rrbracket, i)$ $\mu'_{cl}(\mathcal{L}\llbracket L_i \rrbracket)$ $\rho'_{cl}$))
      $\llbracket$($\lambda$ ($V_1$ ... $V_n$) $L_1E_1$)$\rrbracket$: let $(\mu'_{cl}, \rho'_{cl}) = \mathrm{cl}\llbracket L_1E_1 \rrbracket \mu_{cl}\rho_{cl}$ in
        (upd k $\{\mathcal{L}\llbracket L_1 \rrbracket\}$ $\mu'_{cl}$, $\rho'_{cl}$)
      $\llbracket$($L_0E_0$ $L_1E_1$ ... $L_nE_n$)$\rrbracket$: let $(\mu'_{cl}, \rho'_{cl}) = \bigsqcup_i \mathrm{cl}\llbracket L_iE_i \rrbracket \mu_{cl}\rho_{cl}$ in
        let $c = \{k' \mid k' \in \mu'_{cl}(\mathcal{L}\llbracket L_0 \rrbracket) \wedge \mathrm{arity}(k') = n\}$ in
          (upd k $(\bigsqcup_{k' \in c} \mu'_{cl}(k'))$ $\mu'_{cl}$, $\rho'_{cl} \sqcup \bigsqcup_{i \geq 1, k' \in c}$(upd $(k', i)$ $\mu'_{cl}(\mathcal{L}\llbracket L_i \rrbracket)$ $\rho'_{cl}$))
    end
```

The rules for constants, variables, conditionals, and let-expressions are straightforward. For primitive operations, note that a closure occurring in an argument may possibly be returned, but no new closures may be introduced. For procedure calls, a closure returned by the procedure body may be returned; care must 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 "$(\mu_{cl}, \rho_{cl}) \sqcup$"). A lambda abstraction is the "source" of closures; note that (as mentioned earlier) the closure is identified by the label of the body.

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

4.3 Finiteness

For any given program, there is a finite number of closure sets. The mappings μ_{cl} and ρ_{cl} are updated monotonically, so they can only be updated a finite number of times. Fixed point iteration will therefore stabilize after a finite number of iterations. An implementation of the analysis is thus guaranteed to terminate.

4.4 Implementation issues

In the description, the subexpressions of a compound expression are processed in a parallel way. This simplifies the description, but sequential processing is better from an implementation point of view. Sequential processing means that there is always only *one* active copy of μ_{cl} as well as of ρ_{cl}; the mappings are *single-threaded* [Schmidt 85] and can therefore be implemented as global variables which are updated destructively.

In practice, the mappings are not kept as separate variables, but the information is kept as *attributes* (annotations) in the abstract syntax. This means that expression labels are not actually needed.

4.5 Application to the sample interpreter

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

The lambda abstractions are referred to by a number (0 to 5), the application points by a letter (p to u); see the comments in the interpreter text. Each use of upd is macro expanded into an expression containing a lambda abstraction ((λ (v1) ...)) and an application ((r v1)). t identifies the application of f to (fix f), u the application of (f (fix f)) to x. The closure analysis gives the following possible abstractions at the application points:

```
p, q, r: 0, 1, 5   s: 2  t: 4  u: 3
```

We see that at environment application points, p, q, and r, the environment closures 0, 1, and 5 (but not closure 3!) are the (only) possibilities. Closure 2, which implements lambda abstraction in the interpreted language, is the only one which may be applied at application point s; s implements application in the interpreted language. The only closure that the functional f may be bound to at point t is closure 4 that maps environments to environments. Finally, an "unrolled" recursive environment at point u can only be closure 3.

5. Binding time analysis

This section describes the binding time analysis that assigns a binding time value to all variables and all expressions (labels). The binding time uses the information collected in closure analysis.

5.1 The binding time domain

The binding time domain is a four value lattice:

$b \in \text{BtValue} = (\{\bot, S, CL, D\}, \sqsubseteq)$

The partial ordering is given by

$$\begin{tikzcd}
 & D & \\
S \arrow[ur, dash] & & CL \arrow[ul, dash] \\
 & \bot \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

S approximates ordinary first order static values (constants), *CL* approximates closure values, and *D* approximates dynamic values (residual code expressions). The value $\perp$ is needed because *S* and *CL* are incomparable. *S* corresponds to *Static* in a standard binding time analysis for first order programs, *D* to *Dynamic*.

At program specialization time, a *closure* is generated for *CL* annotated lambda expressions: a closure contains an identification of the lambda expression and values for its free variables. Closures are always eventually (beta) reduced away during program specialization, and *CL* is thus used for eliminable lambda expressions. For *D* annotated lambda expressions, a residual lambda expression (residual code) is generated. The lambda expression is thus suspended: no beta reduction is performed. The body of the residual lambda expression is a residual version of the body of the source lambda expression.

Let us consider an example:

```
((if (p x) (λ (y) y) (λ (z) (cdr z))) 1)
```

If the result of the test is static, i.e. the binding time value of (the label of) the expression `(p x)` is *S*, then the conditional expression always reduces to one of its branches. Consequently, beta reduction can always be performed during program specialization: it is safe to classify the two lambda expressions eliminable (*CL*). If, however, the test is dynamic, then residual code is generated for the conditional expression and beta reduction is not possible. The two lambda expressions are therefore annotated residual (*D*). We note that we do not consider more "exotic" (post-)reductions on residual code; for this particular example, the reduction ((if E_0 E_1 E_2) E_3) $\Rightarrow$ (if E_0 (E_1 E_3) (E_2 E_3)) would in fact enable beta reduction in case of a dynamic test.

5.2 Annotating lambda expressions

It is clearly desirable to classify eliminable as many lambda expressions as possible: this gives a more reduced residual program. On the other hand, a closure value must never be used in a context that makes it part of a residual code piece: residual code consists of expressions, not values internal to the program specializer. Therefore, if (the value of) a lambda expression may be used in such a context, it must be annotated residual. Otherwise it can safely be classified eliminable. (We note that one could imagine a program specializer that always generates a closure when processing a lambda expression (as proposed in [Mogensen 89a]). The specializer should then convert the closure into an expression if used in a residual code context. The method requires tagging and (re-)traversing residual values to find the closures. This is undesirable, especially for self-application.)

The value of a lambda expression E_λ may occur as part of a residual code piece in the following cases: (1) Some expression E has binding time value *D* (i.e. the result of specializing E is expected to be a residual code piece) and E may *itself* — according to the closure analysis — evaluate to (the value of) E_λ. (2) Some non-procedure call compound expression is suspended (a residual version of the expression is generated) and has an *argument* expression that may evaluate to (the value of) E_λ. (3) The body of the program's goal procedure (fixed for any particular program specialization) may evaluate to (the value of) E_λ. In these three cases, E_λ must be classified residual.

Case (1) implies that whenever an expression gets binding time value *D*, then all lambda expressions that E may evaluate to should be raised to be classified residual. Case (3) is needed because residual code is always generated for the body of the goal procedure, regardless of its binding time value. The point in case (2) is that suspending an operation requires generating residual versions of the argument expressions. Procedure calls are an exception: in the residual version of a suspended procedure call, the procedure name has been *specialized* with respect to the static arguments (this is the point in polyvariant program specialization). Closures are *partially static structures* [Mogensen 88] containing static and dynamic subparts; the dynamic parts become arguments to the residual procedure call. For the other compound expressions, some simplification is possible. A case analysis shows that case (1) covers case (2) for conditionals, let-expressions, and primitive operations. Conditionals are suspended in case of a dynamic test and let-expressions are possibly suspended in case of a dynamic actual parameter expression. The point now is that if any *other* argument expression, which because of the suspension 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 closure, then the closure analysis rules imply that the whole expression may return the same closure. Hence, the closure is "captured" by case (1) since the whole compound expression has binding time value *D*.

Binding time values for *formal parameters* of eliminable lambda expressions depend on the binding time values of the argument expressions at any relevant application point. The relevant application points are those where the lambda expression may possibly be applied (computed by the closure analysis). In the example from section 5.1, there is only one such point, and `y` and `z` get the same binding time value as `1`. In general, least upper bounding over all relevant application points is needed. Lambda expressions annotated residual are not beta reduced, and so the formal parameters all become dynamic.

One might think of introducing a binding time value *S-or-CL* lying above *S* and *CL*, but below *D*. This makes sense since Scheme is dynamically typed. Introducing *S-or-CL* gives additional precision in the description, but the program specializer is burdened in two ways: first, any value of the *S-or-CL* type needs to be tagged as either an *S*-value or a *CL*-value; second, the program specializer needs to type check such values. This is avoided by letting $S \sqcup CL = D$.

5.3 Domains and functions

The binding time value of an expression `(O L-E`$_1$ ... `L-E`$_n$`)` is typically $\bigsqcup_i$(the binding time value of `L-E`$_i$) $\sqcup$ *S*. Treating primitive operations working on higher order structures (such as Scheme's `procedure?`) introduces complications since the program specializer represents closures in its own way; this problem is inherent to the very idea of treating higher order operations intensionally. By least upper bounding the arguments with *S*, any primitive operation on a closure becomes dynamic whereby the problem is avoided (since no reduction takes place at partial evaluation time).

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* [Turchin 86], i.e. forcing a static 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 o:

o: `OperatorName` $\rightarrow \text{BtValue}^* \rightarrow \text{BtValue}$

In practice, a binding time function is user defined for each primitive [Bondorf & Danvy 90].

The binding time analysis computes two mappings:

$\mu_{bt} \in \text{BtMap} = \text{Label} \rightarrow \text{BtValue}$

$\rho_{bt} \in \text{BtEnv} = \text{Variable} \rightarrow \text{BtValue}$

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

The closure analysis identifies a 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 k2k:

k2k: Label $\rightarrow$ Label

Given the label of the body of a lambda expression, k2k returns the label of the lambda expression itself.

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

raise: Label$\rightarrow$ BtMap$\rightarrow$ BtEnv$\rightarrow$ BtMap $\times$ BtEnv

raise k μ_{bt} ρ_{bt} =

$\quad \bigsqcup_{k' \in \mu_{cl}(k)}(\text{upd k2k}(k')\ D\ \mu_{bt},$

$\qquad\qquad \bigsqcup_{\mathtt{i} \in \{1 \ldots \text{arity}(k')\}}(\text{upd}\ (k', \mathtt{i})\ D\ \rho_{bt}))$

$\quad \sqcup\ (\mu_{bt}, \rho_{bt})$

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

5.4 The analysis

We now give the binding time analysis rules. 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.

<u>Binding time analysis</u>

```
$\mathrm{Bt}\colon \texttt{Definition}^{+} \to \mathrm{Label} \to \mathrm{BtEnv} \to \mathrm{BtMap} \times \mathrm{BtEnv}$
$\mathrm{Bt}\llbracket\texttt{(define (...) L}_1\texttt{E}_1\texttt{) ... (define (...) L}_n\texttt{E}_n\texttt{)}\rrbracket k_{goal}\, \rho^{input}_{bt} =$
  $\mathit{fix}(\lambda(\mu_{bt}, \rho_{bt})\, .\, (\mu^{init}_{bt}, \rho^{init}_{bt}) \sqcup \bigsqcup_{\texttt{i}} \mathrm{bt}\llbracket\texttt{L}_\texttt{i}\texttt{E}_\texttt{i}\rrbracket\mu_{bt}\rho_{bt})$ where $(\mu^{init}_{bt}, \rho^{init}_{bt}) = \mathrm{raise}^3\, k_{goal} \perp_{\mathrm{BtMap}} \rho^{input}_{bt}$
$\mathrm{bt}\colon \texttt{LabeledExpression} \to \mathrm{BtMap} \to \mathrm{BtEnv} \to \mathrm{BtMap} \times \mathrm{BtEnv}$
$\mathrm{bt}\llbracket\texttt{L E}\rrbracket\mu_{bt}\rho_{bt} =$
  let $k = \mathcal{L}\llbracket\texttt{L}\rrbracket$ in $\mu^{\circ}_{bt}(k) = D \to \mathrm{raise}^1\, k\, \mu^{\circ}_{bt}\, \rho^{\circ}_{bt} \,\square\, (\mu^{\circ}_{bt}, \rho^{\circ}_{bt})$
    where $(\mu^{\circ}_{bt}, \rho^{\circ}_{bt}) =$
      case $\llbracket\texttt{E}\rrbracket$ of
        $\llbracket\texttt{C}\rrbracket$: $(\mathrm{upd}\ k\ S\ \mu_{bt}, \rho_{bt})$
        $\llbracket\texttt{V}\rrbracket$: $(\mathrm{upd}\ k\ \rho_{bt}(\mathcal{V}\llbracket\texttt{V}\rrbracket)\ \mu_{bt}, \rho_{bt})$
        $\llbracket\texttt{(if L}_1\texttt{E}_1\texttt{ L}_2\texttt{E}_2\texttt{ L}_3\texttt{E}_3\texttt{)}\rrbracket$: let $(\mu'_{bt}, \rho'_{bt}) = \bigsqcup_{\texttt{i}} \mathrm{bt}\llbracket\texttt{L}_\texttt{i}\texttt{E}_\texttt{i}\rrbracket\mu_{bt}\rho_{bt}$ in let $b_\texttt{i} = \mu'_{bt}(\mathcal{L}\llbracket\texttt{L}_\texttt{i}\rrbracket)$ in
          $(\mathrm{upd}\ k\ (b_1{=}D \to D \,\square\, b_2 \sqcup b_3)\ \mu'_{bt}, \rho'_{bt})$
        $\llbracket\texttt{(let ((V L}_1\texttt{E}_1\texttt{)) L}_2\texttt{E}_2\texttt{)}\rrbracket$: let $(\mu'_{bt}, \rho'_{bt}) = \bigsqcup_{\texttt{i}} \mathrm{bt}\llbracket\texttt{L}_\texttt{i}\texttt{E}_\texttt{i}\rrbracket\mu_{bt}\rho_{bt}$ in let $b_\texttt{i} = \mu'_{bt}(\mathcal{L}\llbracket\texttt{L}_\texttt{i}\rrbracket)$ in
          $(\mathrm{upd}\ k\ (b_1{=}D \to D \,\square\, b_2)\ \mu'_{bt}, \mathrm{upd}\ \mathcal{V}\llbracket\texttt{V}\rrbracket\ b_1\ \rho'_{bt})$
        $\llbracket\texttt{(O L}_1\texttt{E}_1\texttt{ ... L}_n\texttt{E}_n\texttt{)}\rrbracket$: let $(\mu'_{bt}, \rho'_{bt}) = (\mu_{bt}, \rho_{bt}) \sqcup \bigsqcup_{\texttt{i}} \mathrm{bt}\llbracket\texttt{L}_\texttt{i}\texttt{E}_\texttt{i}\rrbracket\mu_{bt}\rho_{bt}$ in let $b_\texttt{i} = \mu'_{bt}(\mathcal{L}\llbracket\texttt{L}_\texttt{i}\rrbracket)$ in
          $(\mathrm{upd}\ k\ (\mathcal{O}\llbracket\texttt{O}\rrbracket[b_1, \ldots, b_n])\ \mu'_{bt}, \rho'_{bt})$
        $\llbracket\texttt{(P L}_1\texttt{E}_1\texttt{ ... L}_n\texttt{E}_n\texttt{)}\rrbracket$: let $(\mu'_{bt}, \rho'_{bt}) = (\mu_{bt}, \rho_{bt}) \sqcup \bigsqcup_{\texttt{i}} \mathrm{bt}\llbracket\texttt{L}_\texttt{i}\texttt{E}_\texttt{i}\rrbracket\mu_{bt}\rho_{bt}$ in let $b_\texttt{i} = \mu'_{bt}(\mathcal{L}\llbracket\texttt{L}_\texttt{i}\rrbracket)$ in
          $(\mathrm{upd}\ k\ (\mathrm{some}\ b_\texttt{i}{=}D \to D \,\square\, \mu'_{bt}(\mathcal{P}\llbracket\texttt{P}\rrbracket))\ \mu'_{bt}, \rho'_{bt} \sqcup \bigsqcup_{\texttt{i}}(\mathrm{upd}\ (\mathcal{P}\llbracket\texttt{P}\rrbracket, \texttt{i})\ b_\texttt{i}\ \rho'_{bt}))$
        $\llbracket\texttt{(}\lambda\texttt{ (V}_1\texttt{ ... V}_n\texttt{) L}_1\texttt{E}_1\texttt{)}\rrbracket$: let $(\mu'_{bt}, \rho'_{bt}) = \mathrm{bt}\llbracket\texttt{L}_1\texttt{E}_1\rrbracket\mu_{bt}\rho_{bt}$ in
          $\mu''_{bt}(k) = D \to \mathrm{raise}^2\ \mathcal{L}\llbracket\texttt{L}_1\rrbracket\ \mu''_{bt}\ \rho'_{bt} \,\square\, (\mu''_{bt}, \rho'_{bt})$ where $\mu''_{bt} = \mathrm{upd}\ k\ CL\ \mu'_{bt}$
        $\llbracket\texttt{(L}_0\texttt{E}_0\texttt{ L}_1\texttt{E}_1\texttt{ ... L}_n\texttt{E}_n\texttt{)}\rrbracket$: let $(\mu'_{bt}, \rho'_{bt}) = \bigsqcup_{\texttt{i}} \mathrm{bt}\llbracket\texttt{L}_\texttt{i}\texttt{E}_\texttt{i}\rrbracket\mu_{bt}\rho_{bt}$ in let $b_\texttt{i} = \mu'_{bt}(\mathcal{L}\llbracket\texttt{L}_\texttt{i}\rrbracket)$ in
          let $c = \{k' \mid k' \in \mu'_{cl}(\mathcal{L}\llbracket\texttt{L}_0\rrbracket) \wedge \mathrm{arity}(k'){=}n\}$ in
            $\mu''_{bt}(\mathcal{L}\llbracket\texttt{L}_0\rrbracket) = D \to (\mu''_{bt}, \rho''_{bt}) \sqcup \bigsqcup_{\texttt{i}\geq 1}(\mathrm{raise}^2\ \mathcal{L}\llbracket\texttt{L}_\texttt{i}\rrbracket\ \mu''_{bt}\ \rho''_{bt}) \,\square\, (\mu''_{bt}, \rho''_{bt})$
            where $\mu''_{bt} = \mathrm{upd}\ k\ (\mathrm{some}\ b_\texttt{i}{=}D \to D \,\square\, \bigsqcup_{k' \in c}\mu'_{bt}(k'))\ \mu'_{bt}$
                  $\rho''_{bt} = \rho'_{bt} \sqcup \bigsqcup_{\texttt{i}\geq 1, k' \in c}(\mathrm{upd}\ (k', \texttt{i})\ \mu'_{bt}(\mathcal{L}\llbracket\texttt{L}_\texttt{i}\rrbracket)\ \rho'_{bt})$
      end
```

The applications of the function raise have been superscripted; the numbers refer to the cases (1)-(3) that cause lambda annotations to be raised (section 5.2).

5.5 Finiteness

There is a finite number of binding time values. Since the mappings μ_{bt} and ρ_{bt} are only updated monotonically, they can only be updated a finite number of times. Fixed point iteration will therefore stabilize after a finite number of iterations.

5.6 Application to the sample interpreter

When applied to the sample 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 eliminable. Dually, the applications `p`, `q`, `r`, `t`, and `u`, become eliminable: the expression to be applied in all cases gets binding time value *CL*. The lambda expression `2` and the application `s` become residual.

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

6. Results

In this section we use Similix-2 to specialize the direct style Λ-interpreter and two other Λ-interpreters: one written in continuation passing style and one implementing normal order reduction.

6.1 Direct style

Specializing the sample interpreter with respect to the factorial Λ-program yields the following Scheme target program:

<u>Machine produced factorial target program, generated from direct style Λ-interpreter</u>

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

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

(define (_E-1 r)
  (λ (w)
    (if (ext '= w 0)
        1
        (ext '*
             w
             ((_E-1 r) (ext '- w 1))))))
```

For readability, we have "cheated" by renaming some of the machine generated names (but this is a trivial conversion).

`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 14 times faster to run the target program than to interpret the source program (see the next section on performance).

Recursion is expressed by the procedure `_E-1`. The redundant variable `r` corresponds to the `input` variable in the factorial Λ-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 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.

6.2 Continuation passing style

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

<u>Continuation passing style Λ-interpreter</u>

```
(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) (λ (w) w)))

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

Binding time analysis (with static program and dynamic data input) classifies the environments eliminable (*CL*). The lambda expression `(λ (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 residual (*D*). The eta-conversions are then inserted to achieve that the binding time analysis classifies `_E`'s continuation parameter `c` eliminable rather than residual. This implies that the program specializer will beta reduce continuation applications at partial evaluation time, thus giving better, more reduced target programs.

The following target program is generated when specializing the interpreter with respect to the factorial program:

Machine produced factorial target program, generated from continuation style Λ-interpreter

```
(loadt "scheme.adt")
(loadt "lam-int.adt")
(define (run-0 w)
   ((_E-1 w) w (λ (w) w)))
(define (_E-1 r)
  (λ (w1 c1)
    (if (ext '= w1 0)
        (c1 1)
        ((_E-1 r)
         (ext '- w1 1)
         (λ (w) (c1 (ext '* w1 w)))))))
```

The target program is written in continuation passing style since the interpreter was.

6.3 Normal order reduction

The third interpreter is a variant of the direct style one, but it implements normal order reduction semantics. Normal order reduction is achieved by suspending the evaluation of arguments to applications. Instead of keeping values in environments, we thus now keep thunks of the form `(λ () ...)`.

Normal order reduction Λ-interpreter

```
(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) (λ (x) ((f (fix f)) x)))
```

Note that primitive operations are still call-by-value; only applications of lambda abstractions are call-by-name.

The following program 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.

<u>Even number program written in normal order Λ</u>

```
((λ lazy-cons
  ((λ lazy-car
    ((λ lazy-cdr
      (letrec first-n
        (λ n (λ l
          (if (= n 0)
              '()
              (cons
                (lazy-car l)
                ((first-n (- n 1))
                 (lazy-cdr l)))))))
        (letrec evens-from
          (λ n
            ((lazy-cons n)
             (evens-from (+ n 2))))
          ((first-n input)              ;main
           (evens-from 0))))))
     (λ x (x (λ a (λ d d))))))
   (λ x (x (λ a (λ d a))))))
 (λ x (λ y (λ z ((z x) y)))))
```

The Λ-language has no let-expressions and only unary lambda expressions, so the program looks somewhat clumsy.

Specializing the normal order interpreter with respect to the even number program yields a target program in which syntax analysis and environment operations have all been performed. The program contains lots of thunks and is rather hard to read (we do not include it here). It is, however, quite efficient: running the target program is around 25 times faster than interpréting.

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 thunks 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 siginificant interpretation overhead. But using partial evaluation, programs in the call-by-name language are compiled into efficient Scheme code (which is eventually itself compiled).

7. Performance

This section contains some benchmarks for Similix-2. The tables below show the speedups achieved by partial evaluation. Each table has four columns. The first one describes the result computed by the job in the second column. The third column shows the run time, the fourth column 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 40% additional time), but include postprocessing. The speedup ratios have been computed using more decimals than the ones given here; in some cases, the time has been computed by performing 10 successive runs and then dividing. The system is implemented in Chez Scheme [Dybvig 87] version 2.0.3, and the figures are for a Sun 3/160.

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 normal order example, the even number program is used; the figures are for 10 computations of "evens" of 20.

output	run	time/s	speedup
result	int(source, data)	5·7	14·3
	target(data)	0·40	
target	mix(int^{ann}, source)	0·53	7·8
	comp(source)	0·07	
comp	mix(mix^{ann}, int^{ann})	11·5	2·9
	cogen(int^{ann})	4·0	

<u>Direct style Λ-interpreter</u>

output	run	time/s	speedup
result	int(source, data)	6·1	17·1
	target(data)	0·36	
target	mix(int^{ann}, source)	1·2	7·8
	comp(source)	0·15	
comp	mix(mix^{ann}, int^{ann})	49·8	2·3
	cogen(int^{ann})	21·9	

Continuation passing style Λ-interpreter

output	run	time/s	speedup
result	int(source, data)	35·0	25·7
	target(data)	1·4	
target	mix(int^{ann}, source)	6·0	5·4
	comp(source)	1·1	
comp	mix(mix^{ann}, int^{ann})	16·3	3·2
	cogen(int^{ann})	5·1	

Normal order reduction Λ-interpreter

output	run	time/s	speedup
cogen	mix(mix^{ann}, mix^{ann})	82·3	3·0
	cogen(mix^{ann})	27·6	

Compiler generator

The first table shows that running the factorial target program is around 14 times faster than interpreting the factorial source program. Compiling by the stand-alone compiler is 8 times faster than by specializing the interpreter; this shows that the partial evaluator really is effectively self-applicable. Finally, generating the compiler by the mix-generated compiler generator cogen is 3 times faster than by specializing mix. The second and third tables are similar. The last table 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 an interpreter (includes closure and binding time analyses); preprocessing mix takes around 19 seconds. The size of mix is 2.5K cells (measured as the number of "cons" cells needed to represent the program as a list), cogen 13.9K cells, the interpreters 0.18K-0.26K cells, and the compilers 1.9K-7.6K cells. For mix, this gives an expansion factor of 5.5 (13.9/2.5), for the interpreters factors in the range 10-29.

The figures all in all compare well to similar published benchmarks for first order languages [Jones, Sestoft, & Søndergaard 89] [Bondorf & Danvy 90] [Consel 89], and also to those of Lambda-mix [Gomard 89].

8. Discussion

Partial evaluation is no panacea: some programs specialize well, but others do not. Program generators in general take some specification as input; in the case of partial evaluation, the specification is a program. The quality of a program generated by any program generator depends on the quality of the specification. For partial evaluation, the quality of the residual program depends on the quality of the source program supplied to the partial evaluator.

The "quality" of a source program does not necessarily mean its clarity or efficiency. It often happens that less efficient and/or less clear programs lead to better (more efficient, more clear) residual programs.

8.1 Exploiting static information

Programs have to be expressed carefully not to lose static information. A simple example: suppose `x` and `y` are static and `z` dynamic. Then `(+ (+ x y) z)` specializes better than `(+ x (+ y z))`: in the former case, the inner + is reduced, but in the latter no reduction takes place.

8.2 Currying

It was mentioned earlier (section 3.2) that uncurrying functions sometimes gives better specialization. When binding time analysing a curried expression such as `E = (λ (x) (λ (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 that procedure call unfolding and beta reduction of higher order applications *duplicates* or *discards* dynamic actual argument expressions, let-expressions are *inserted* for all formal parameters in source programs before preprocessing [Bondorf & Danvy 90]. The expression which is actually binding time analysed is therefore not E, but the semantically equivalent

```
(λ (x) (let ((x x)) (λ (y) (let ((y y))
  (+ (+ y y) x)))))
```

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. Hence, its parameter y becomes dynamic whereby static information is lost. It thus never happens that dynamic arguments are supplied before static arguments. This confirms the intuition in [Nielson & Nielson 88]: "early bindings before late bindings".

If a curried expression is always applied to all its arguments simultaneously, then it is advantageous 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.

9. Conclusion and open problems

We have presented an approach to treat a higher order subset of Scheme in autoprojection. We have implemented the ideas by extending the existing Similix autoprojector. 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 domain of binding time values 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 "thunks" to implement normal order reduction.

Several problems remain open. In the line of compiler generation, the system should be applied to bigger, more realistic examples. It would also be interesting to experiment with interpreters for real lazy (i.e. call-by-need rather than call-by-name) languages; we have made some promising experiments in this direction.

The autoprojector itself could also be improved. 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, then the least upper bound is taken. This implies 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.

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, Neil D. Jones, Jesper Jørgensen, Karoline Malmkjær, Torben Mogensen, Peter Sestoft, and Jörg Süggel.

References

[Aho, Sethi, & Ullman 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman: *Compilers: Principles, Techniques and Tools*, Addison-Wesley 1986.

[Bjørner, Ershov, & Jones 88] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones (eds.): *Partial Evaluation and Mixed Computation*, Gl. Avernæs, Denmark, October 1987, North-Holland 1988.

[Bondorf 89] Anders Bondorf: *A self-applicable partial evaluator for term rewriting systems*, TAPSOFT'89, Proceedings of the International Joint Conference on Theory and Practice of Software Development, J. Diaz and F. Orejas (eds.), Barcelona, Spain, Lecture Notes in Computer Science No 352 pp 81-96, Springer-Verlag 1989.

[Bondorf 90] Ph.D. thesis (forthcoming), DIKU, University of Copenhagen, Denmark.

[Bondorf & Danvy 90] Anders Bondorf and Olivier Danvy: *Automatic autoprojection of recursive equations with global variables and abstract data types*, Technical Report No 90-4, DIKU, University of Copenhagen, Denmark.

[Bondorf, Jones, Mogensen, & Sestoft 90] Anders Bondorf, Neil D. Jones, Torben Æ. Mogensen, and Peter Sestoft: *Binding time analysis and the taming of self-application*, submitted for publication, DIKU, University of Copenhagen, Denmark.

[Bulyonkov 84] Mikhail A. Bulyonkov: *Polyvariant mixed computation for analyzer programs*, Acta Informatica 21 pp 473-484, 1984.

[Consel 88] Charles Consel: *New insights into partial evaluation: the SCHISM experiment*, ESOP'88 (ed. Harald Ganzinger), Nancy, France, Lecture Notes in Computer Science No 300 pp 236-247, Springer-Verlag 1988.

[Consel 89] Charles Consel: *Analyse de programmes, Evaluation partielle et Génération de compilateurs*, Ph.D. thesis, LITP, University of Paris 6, France 1989.

[Consel & Danvy 89] Charles Consel and Olivier Danvy: *Partial evaluation of pattern matching in strings*, Information Processing Letters 30, No 2 pp 79-86, 1989.

[Dybvig 87] R. Kent Dybvig: *The SCHEME Programming Language*, Prentice-Hall, New Jersey 1987.

[Ershov 77] Andrei P. Ershov: *On the partial computation principle*, Information Processing Letters 6, No 2 pp 38-41, April 1977.

[Futamura 71] Yoshihiko Futamura: *Partial evaluation of computing process — an approach to a compiler-compiler*, Systems, Computers, Controls 2, 5, 45-50, 1971.

[Gomard 89] Carsten K. Gomard: *Higher Order Partial Evaluation — HOPE for the Lambda Calculus*, Master's thesis, DIKU student report 89-9-11, University of Copenhagen, 1989.

[Gomard 90] Carsten K. Gomard: *Partial Type Inference for Untyped Functional Programs*, submitted for publication, DIKU, University of Copenhagen, 1989.

[Gomard & Jones 89] Carsten K. Gomard and Neil D. Jones: *Compiler generation by partial evaluation*, Information Processing '89. Proceedings of the 11th IFIP World Computer Congress, G. X. Ritter (ed.), pp 1139-1144, North-Holland, 1989.

[Hudak & Young 88] Paul Hudak and Jonathan Young: *A collecting interpretation of expressions (without powerdomains)*, Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages pp 107-118, San Diego, California, January 1988.

[JonGomBonDanMog 90] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Æ. Mogensen: *A self-applicable partial evaluator for the lambda calculus*, IEEE Computer Society 1990 International Conference on Computer Languages, 1990.

[Jones, Sestoft, & Søndergaard 85] Neil D. Jones, Peter Sestoft, and Harald Søndergaard: *An experiment in partial evaluation: the generation of a compiler generator*, Rewriting Techniques and Applications (ed. J.-P. Jouannaud), Dijon, France, Lecture Notes in Computer Science No 202 pp 124-140, Springer-Verlag 1985.

[Jones, Sestoft, & Søndergaard 89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard: *MIX: a self-applicable partial evaluator for experiments in compiler generation*, International Journal LISP and Symbolic Computation 2, 1, pp 9-50, 1989

[Kohlbecker 86] Eugene E. Kohlbecker: *Syntactic Extensions in the Programming Language Lisp*, Ph.D. thesis, Indiana University, Bloomington 1986.

[Mogensen 88] Torben Æ. Mogensen: *Partially static structures in a self-applicable partial evaluator*, pp 325-347 of [Bjørner, Ershov, & Jones 88].

[Mogensen 89a] Torben Æ. Mogensen: *Binding Time Aspects of Partial Evaluation*, Ph.D. thesis, DIKU, University of Copenhagen, Denmark 1989.

[Mogensen 89b] Torben Æ. Mogensen: *Binding time analysis for polymorphically typed higher order languages*, TAPSOFT'89, Proceedings of the International Joint Conference on Theory and Practice of Software Development, J. Diaz and F. Orejas (eds.), Barcelona, Spain, Lecture Notes in Computer Science No 352 pp 298-312, Springer-Verlag 1989.

[Nielson & Nielson 88] Hanne R. Nielson and Flemming Nielson: *Automatic binding time analysis for a typed λ-calculus*, Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages pp 98-106, San Diego, Calinfornia, January 1988.

[Rees & Clinger 86] Jonathan Rees and William Clinger (eds.): *Revised^3 Report on the Algorithmic Language Scheme*, Sigplan Notices 21, 12, pp 37-79, December 1986.

[Romanenko 88] Sergei A. Romanenko: *A compiler generator produced by a self-applicable specialiser can have a surprisingly natural and understandable structure*, pp 445-463 of [Bjørner, Ershov, & Jones 88].

[Schmidt 85] David A. Schmidt: *Detecting global variables in denotational specifications*, ACM Transactions on Programming Languages and Systems 7, No 2 pp 299-310, April 1985.

[Schmidt 86] David A. Schmidt: *Denotational Semantics, a Methodology for Language Development*, Allyn and Bacon, Boston 1986.

[Sestoft 88a] Peter Sestoft: *Automatic call unfolding in a partial evaluator*, pp 485-506 of [Bjørner, Ershov, & Jones 88].

[Sestoft 88b] Peter Sestoft: *Replacing Function Parameters by Global Variables*, Master's thesis, DIKU student report 88-7-2, University of Copenhagen, 1988.

[Sestoft 89] Peter Sestoft: *Replacing function parameters by global variables*, Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture, London, UK, pp 39-53, ACM Press, September 1989.

[Turchin 80] Valentin F. Turchin: *Semantic definitions in Refal and the automatic production of compilers*, Proceedings of the Workshop on Semantics-Directed Compiler Generation, Neil D. Jones (ed.), Århus, Denmark, Lecture Notes in Computer Science No 94 pp 441-474, Springer-Verlag 1980.

[Turchin 86] Valentin F. Turchin: *The concept of a supercompiler*, ACM Transactions on Programming Languages and Systems 8, No 3 pp 292-325, July 1986.