Science of Computer Programming 16 (1991) 151-195
Elsevier

Automatic autoprojection of recursive equations with global variables and abstract data types

Anders Bondorf*

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

Olivier Danvy**

Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA

Communicated by J. Darlington
Received March 1990
Revised January 1991

Abstract

Bondorf, A. and O. Danvy, Automatic autoprojection of recursive equations with global variables and abstract data types, Science of Computer Programming 16 (1991) 151-195.

Self-applicable partial evaluation has been implemented for half a decade now, but many problems remain open. This paper addresses and solves the problems of automating call unfolding, having an open-ended set of operators, and processing global variables updated by side effects. The problems of computation duplication and termination of residual programs are addressed and solved: residual programs never duplicate computations of the source program; residual programs do not terminate more often than source programs.

This paper describes the automatic autoprojector (self-applicable partial evaluator) *Similix*; it handles programs with user-defined primitive abstract data type operators which may process global variables. Abstract data types make it possible to hide actual representations of data and prevent specializing operators over these representations. The formally sound treatment of global variables makes Similix fit well in an applicative order programming environment.

We present a new method for automatic call unfolding which is simpler, faster, and sometimes more effective than existing methods: it requires neither recursion analysis of the source program, nor call graph analysis of the residual program.

To avoid duplicating computations and preserve termination properties, we introduce an abstract interpretation of the source program, abstract occurrence counting analysis, which is performed during preprocessing. We express it formally and simplify it.

* This author 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 Informatik V, University of Dortmund, Germany.

** Most of this work was carried out when this author was employed by DIKU.

Similix has been implemented and self-applied. It has been used for a number of experiments such as compiler generation from interpretive specifications and generation of efficient pattern matchers from naive pattern matching programs.

This paper is a revision of [7].

1. Introduction

Partial evaluation transforms programs with incomplete input data: when given a *source* program p and a part of its input s, a partial evaluator *mix* generates a *residual* program p_s by *specializing p with respect to s.* Partial evaluation is also referred to as *program specialization.* When applied to the remaining input d, the residual program gives the same result as the source program would when applied to the complete input:

$$p(s, d) = p_s(d) \quad \text{where } p_s = mix(p, s).$$

For simplicity, we have identified programs with the functions they compute. For instance, p denotes a function in $p(s, d)$ and a program in $mix(p, s)$. Input s is called *static* and input d is called *dynamic.*

The main point in partial evaluation is one of efficiency: running the residual program p_s can be much faster than running the source program p. If p is being repeatedly applied to inputs with a fixed value for s, but each time with a new value for d, then it may be worthwhile first to generate p_s instead and then apply it to the different d inputs. The partial evaluator knows p and s and is therefore able to perform those of p's computations that only depend on s. Program p_s is thus (potentially) more efficient than program p: it need not perform the computations that depend only on s.

1.1. Self-application

Self-application means specializing the partial evaluator itself. This is also known as *autoprojection* [17]. Let us substitute *mix* for p, p for s, and s for d in the equation defining a residual program:

$$mix(p, s) = mix_p(s) \quad \text{where } mix_p = mix(mix, p).$$

Specializing p with respect to s may thus be achieved by running $mix_p(s)$ instead of the (potentially) slower $mix(p, s)$.

We may even go one step further: we can specialize the specilizer with respect to itself:

$$mix(mix, p) = mix_{mix}(p) \quad \text{where } mix_{mix} = mix(mix, mix).$$

We may thus generate mix_p by running $mix_{mix}(p)$ instead of the (potentially) slower $mix(mix, p)$. In the particular case where p is an *interpreter int*, these equations are known as the *Futamura projections* [1]; mix_{int} is then a compiler *comp* and mix_{mix} is a compiler generator *cogen.*

The first successfully implemented autoprojector was *Mix* [23]. The language treated by Mix was first-order statically scoped Lisp-type recursive equations over symbolic values, 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 [24]. Since then, autoprojectors for several languages have been implemented: for a subset of Turchin's Refal language [34], for an imperative flowchart language [18], and for pattern-matching-based programs in the form of restricted term rewriting systems [3].

1.2. Partial evaluation, operationally

Partial evaluation works by propagating the static input and reducing static operations. As an example, a conditional expression (`if` E_1 E_2 E_3) can be reduced 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 specializing the conditional is the result of specializing the branch chosen by evaluating the test E_1. If E_1 is dynamic, that is, if its value depends on some dynamic input, 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 obtained by specializing 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 [23, 24]. 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.

Polyvariant specialization achieves *sharing* in residual programs: in the residual program, there may be many calls to the same (residual) function [24]. Relying on the sharing property, it has been possible to generate residual programs equivalent to the Knuth–Morris–Pratt algorithm as well as compiled directed word acyclic graphs out of a brute-force (naive and quadratic) matching program. Self-application provides the corresponding matcher generators [13, 28].

1.3. Binding time analysis

Experience has shown 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 [23]. 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 source program, 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 yet).

The central preprocessing phase is *binding time analysis* [24]. Binding time analysis is usually implemented by *abstract interpretation* (and is therefore *safe* but *approximative*): the program is abstractly interpreted over a binding time lattice, in the simplest case the two-point lattice S⊑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 nonstatic" 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. During program specialization, static operations are reduced whereas residual code is generated for the dynamic ones.

1.4. This work

The main motivation for this work was a desire to design a user-friendly and automatic autoprojector and to use it for further experimenting and better understanding.

It is desirable to have *data abstraction* in the programs to be specialized [11]. This is necessary to have specialized versions of programs that do not account for the actual representation of data structures (environments, etc.). Experience from using Mix shows that having to specify everything as for example Lisp-type lists yields too low-level residual programs because even the processing of data structures has been specialized. Using abstract data types makes it possible to hide the representation and treatment of data from the partial evaluator. (Note: we only use the term "abstract data type" in this sense; we do not consider algebraic issues.)

For the sake of generality, it is advantageous to have an *open-ended* set of operators in the source language, as introduced in [10] and also described in [11]. This can be combined with the abstract data type approach, but it also introduces a new problem of coexistence: a program specialized with respect to another program (for instance by self-application) intermingles operators from both programs. Thus a general solution to handle possible homonymies between sets of operators is needed.

Existing methods for automating *call unfolding*, one of the central transformations performed by a partial evaluator, rely on particular knowledge about a fixed set of primitive operators—for instance that (car l) is a substructure of l [38]. To automate call unfolding in an autoprojector for a language with an open-ended set of operators, other methods are needed. We describe a new algorithm for classifying calls unfoldable/residual, and we also present an alternative *post-unfolding* method. Our post-unfolding method is significantly simpler and faster than the one described in [38] (no expensive *call graph analysis* is needed), and in some cases it is more effective.

There exist sound methods to detect and globalize *single-threaded* variables in functional programs [35, 40]. Many programs naturally use such global variables directly, for example the i/o from any existing, call-by-value oriented operating system. In interpretive specifications of programming languages and program transformation systems, a number of variables are single-threaded: variables holding stores, various counters, symbol generators, and so on. It is natural to keep them global, as done for example in action semantics [32].

Also in partial evaluators, some internal variables are single-threaded. Globalizing them reduces the size and improves the speed of the specializer and of its specialized instances obtained by autoprojection. In this paper, we show how to handle global variables safely during partial evaluation.

Computation *duplication* is a problem that may result in very bad residual programs: if a program runs in linear time, its residual version may sometimes run in exponential time due to computation duplication [23]. A closely related problem, call duplication, is addressed and solved in [38]; in Section 3.6 we give a simpler solution which does not intermingle the problems of call unfolding and duplication.

A related problem concerns preservation of termination properties: due to the call-by-name nature of unfolding, partial evaluators may sometimes produce residual programs that terminate more often than the source programs [24]: computations may be *discarded* due to unfolding. In this paper, we address and solve this problem. We almost get the solution for free: the analysis needed for avoiding duplication only needs a small modification to handle the discarding problem too.

1.5. Notation and prerequisites

Programs are written in a subset of the Scheme language [33]. Formal descriptions of algorithms are written in a conventional denotational semantics style using double brackets to surround syntactic objects. The conditional is written as $\_ \rightarrow \_ \, [] \, \_$.

Knowledge about partial evaluation is no requirement but will definitely be an advantage. Good introductions may be found in e.g. [23, 24].

1.6. Outline

In Section 2 we describe the language treated by Similix. Section 3 describes the central problems solved in Similix. Section 4 contains a formal description of the binding time analysis, and Section 5 discusses automatic call unfolding. In Section 6 we develop an analysis needed for avoiding duplicating and discarding computations. Section 7 contains a larger example of partial evaluation: we specialize an interpreter for an imperative language, thus compiling imperative programs into Scheme. Section 8 gives some benchmarks for Similix. Section 9 discusses related work and in Section 10 we conclude.

2. Language

Similix processes recursive equations expressed in a subset of Scheme [33] (see Fig. 1). Since programs follow the syntax of Scheme, they are directly executable in a Scheme environment.

```
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) |
          (begin L-E+) | (O L-E*) | (P L-E*)
```

Fig. 1. Abstract syntax of Scheme subset handled by Similix.

A source program is expressed by a set of user-defined procedures and a set of user-defined primitive operators. Following Scheme terminology, we use the term "procedure" rather than "function". Procedures are treated intensionally, whereas primitive operators are treated extensionally. The partial evaluator is given the definition 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 perform the operation or suspend it, generating residual code.

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: they are used to give a uniform description of annotations computed during preprocessing. Except for the labels, this abstract syntax is identical to the concrete one.

The user-defined primitive operators are defined in external modules located in files. These files are loaded by the `loadt` expressions; this is described in Section 2.2. Procedure definitions and `loadt` expressions from other files can be reused using `load`.

An expression can be a constant (boolean, number, string, or quoted construction), a variable, a conditional, a let-expression (unary for simplicity; L-E_1 is called the *actual parameter* expression and L-E_2 the *body* expression), a sequence (`begin`) operation (used for sequentializing side effecting expressions), a primitive operation (applying a user-defined operator), or a procedure call. The order of evaluation is applicative (strict, call-by-value, inside-out), and arguments are evaluated in an unspecified order.

2.1. Syntactic extensions

A number of built-in syntactic extensions [26] are treated by Similix. We mention the ones used in this paper. The form `cond` is expanded into (nested) `if`-expressions; a sequence of let-expressions can be abbreviated by `let*`:

(`let*` ((V_1 E_1) . . . (V_n E_n)) E) expands into

`(let ((`$V_1 E_1$`))...(let ((`$V_n E_n$`)) E)...).`

Implicit sequencing is allowed as in Scheme. E^+ thus expands into `(begin` E^+`)` in the syntactically sugared forms `(define (PV*)` E^+`)` and `(let ((VE`$_1$`))` E^+`)`.

2.2. *User-defined primitive operators*

User-defined primitive operators are defined according to the abstract syntax given in Fig. 2. Here are some examples of operator definitions:

```
(defprim-transparent (my-op x y) (cons x (cons x y))).
(defprim-transparent 1 my-car car).
(defprim-opaque 1 read read).
```

The form SE can be any Scheme expression and is thus not restricted to the expression subset allowed in procedures (Fig. 1). Similix never looks "inside" SE-expressions, but considers a primitive operation to be atomic. The representation of the data handled by primitive operators is completely hidden to the partial evaluator, thus providing abstract data type operators. SV-variables are variables defined at the Scheme top-level (such as read), or possibly operators O defined earlier in the file.

```
OD ∈ OperatorDefinition, O ∈ OperatorName,
SE ∈ SchemeExpression, SV ∈ SchemeVariable

OD     ::=  (Key (O V*) SE) | (Key Arity O SV)
Key    ::=  defprim-transparent | defprim-dynamic | defprim-opaque
Arity  ::=  0,1,2, ...
```

Fig. 2. User-defined primitive operators.

When a program is run ordinarily, the operator definitions correspond to ordinary Scheme definitions. The three above definitions would thus correspond to the definitions:

```
(define my-op (lambda (x y) (cons x (cons x y)))).
(define my-car car).
(define read read).
```

Notice that in Scheme, a definition such as `(define (read x) (read x))` is *not* equivalent to `(define read read)`: the former one redefines read to a nonterminating operation whereas the latter one binds read to its former value and thus has no effect.

When a program is partially evaluated, the additional information in the operator definitions becomes significant: arity information and transparency information.

The arity of operations of the form (Key (O V*) SE) is given by the number of arguments, but has to be specified explicitly for the form (Key Arity O SV) (there is no way to deduce the arity of a functional object in Scheme).

The key gives the transparency information: an operator can be referentially transparent or opaque, according to whether it uses global variables, or it can be dynamic (to be explained below). read is opaque since it accesses (and even updates) a global input stream.

In effect, the transparency information associates an abstract binding time function to each operator. Given the binding times of the arguments, this function computes the binding time of the result. Transparent operators thus have associated binding time functions that take the least upper bound of the binding times of the arguments; dynamic and opaque operators have more conservative associated binding time functions (see Section 4.1 for precise definitions). For instance, the following operator implements *generalization* [41]:

```
(defprim-dynamic (generalize x) x).
```

Generalization consists in raising a static value to be dynamic. Operationally, generalize acts as the identity, but its binding time function makes its result dynamic, even when the argument is static.

Generalization provides the user a way to prevent *infinite specialization* (generating infinitely many specialized versions f-x of a source procedure f). Generalization yields more conservative residual programs by delaying the evaluation of static expressions until run time.

It is always *possible* to avoid infinite specialization: by generalizing all arguments to all procedure calls; there will then be exactly one specialized version of each procedure, specialized with respect to no static values at all. Of course this trivial solution does not yield good results (although the residual program, because of unfolding, may still be an optimization of the source program). One should generalize only when necessary.

Generalization can also be used to increase sharing in residual programs, for example to obtain residual programs that are linear in size with respect to the static input [28].

In practice, generalization is rarely necessary, and finding the right generalization points is problem-dependent. For these reasons Similix does not try to find these points automatically. Instead, this is left to the user.

3. Overview of central problems and assumptions

3.1. Having an open-ended set of operators

The partial evaluator must ensure that programs always run with the right set of operators. For example, the residual program *target*, obtained by specializing an interpreter *int* with respect to a program *source*, runs with the set of operators of *int* and occasionally uses some of *source*'s operators. Similarly, the residual program *comp* (compiler), obtained by specializing the specializer *mix* with respect to *int*, runs with the set of operators of *mix*. However, since *comp* implements the specialization of *int* with respect to some program *source*, it can use the operators of *int* (though not those of *source*). Finally, the residual program *target'* obtained by applying *comp* to a program *source* runs with the set of operators of *int* and occasionally uses some of *source*'s operators (this was already stated, since *target* and *target'* are textually identical).

With a specializer having a universal and fixed set of operators, there is no problem of possible inconsistency between the different operator sets. But with a specializer having an open-ended set of operators, ensuring consistency is vital since different sets of operators may overlap (for example, they may name two different operators identically).

Similix provides this consistency in a transparent way. A program declares its own set of operators, which is loaded when the program is loaded. A residual program loads the same operators as the source program did. For instance, a compiler *comp* loads the operators coming from *mix*. The operators in *comp* coming from *int* are also loaded automatically, but these are stored separately to avoid name clashes with the operators coming from *mix*. Operators from *int* are evaluated in an indirect way, essentially by using Scheme's `apply`.

3.2. Termination of specialization: call unfolding

Unfolding, the replacement of a procedure call by the (partially evaluated) body of the procedure definition, increases the efficency of residual program. Let us review unfolding with the standard example, a recursive program appending two lists:

```
(define (append xs ys)
 (if (null? xs)
    ys
    (cons (car xs) (append (cdr xs) ys))))
```

Specializing this program with respect to a static value for `xs`, for instance the list `(7 8)`, yields the following—intermediate—residual program:

```
(define (append-0 ys) (cons 7 (append-1 ys)))
(define (append-1 ys) (cons 8 (append-2 ys)))
(define (append-2 ys) ys)
```

Static computations have been performed (the `null?`, `car`, and `cdr` operations) and the conditional has been reduced (since the test `(null? xs)` is static), but we have not yet unfolded calls. Unfolding simplifies the intermediate residual program to the following program:

```
(define (append-0 ys) (cons 7 (cons 8 ys))).
```

For another example, we could interchange the inputs and thus specialize append with respect to a dynamic first and a static second parameter. This would give this residual program:

```
(define (append-0 xs)
  (if (null? xs)
      '(7 8)
      (cons (car xs) (append-0 (cdr xs)))))
```

The test in the conditional is now dynamic and therefore cannot be reduced at partial evaluation time. Thus the conditional remains residual. Clearly, one cannot systematically unfold the append-0 calls (this would give infinite unfolding).

Sometimes one can unfold, sometimes not. Some way of controlling unfolding is necessary. In Similix, this is done automatically by combining pre- and postprocessing: some calls are classified "unfoldable" in preprocessing, and these are blindly unfolded at program specialization time. Then the residual program is simplified by post-unfolding some of the remaining calls. This approach was also used in [38], but the preprocessing algorithm described there relies on knowledge about a particular set of primitives (e.g. that car produces a substructure). To handle a language with an open-ended set of primitives, a new preprocessing algorithm is needed.

The postprocessing algorithm described in [38] is rather complex. We present a simpler and faster algorithm. Similix's call unfolding strategy (pre- and postprocessing) is covered in Section 5.

3.3. Termination of specialization: static computations

During program specialization, static computations are evaluated completely to exploit the static results. As an example, we evaluated (null? xs) when xs was static above.

This gives a termination problem of partial evaluation: such a complete evaluation may not terminate (a recursive procedure may be called with only static parameters). However, any standard evaluation of the same program, with input values that coincide with the static part of the input values to the partial evaluation, will not terminate in that case either (if the looping part is entered): the looping is controlled by the static part of the input only. We therefore accept the possibility of nontermination of static computations during program specialization (as is also done in [38]).

Note that if a dynamic test guards the looping part, partial evaluation will enter the loop (since both branches of the conditional need to be specialized). But standard evaluation may, depending on the test, not enter the looping branch. Standard evaluation may thus terminate more often that partial evaluation.

3.4. Computation duplication

Let us consider an intermediate residual program in which all static computations have been performed. Let-expressions with static actual parameter expression have thus been (beta) reduced (to make maximal use of static information at partial evaluation time), but let us further assume that all let-expressions with nonstatic parameter remain in the (intermediate) residual program.

One would often like to unfold such residual let-expressions, similarly to unfolding calls: the residual program becomes shorter and more efficient. But it is not always a good idea to unfold let-expressions. For example, let us consider the following residual let-expression:

```
(let ((n (foo ...)))
 (+nn))
```

If the let-expression were unfolded to `(+ (foo ...) (foo ...))`, the computation performed by the expression `(foo ...)` would be performed twice instead of once when running the code piece. This would be inefficient and even incorrect if `foo` had side effects (side effects are addressed in Section 3.9).

To avoid unfolding such a let-expression an *occurrence counting analysis* is needed. The analysis will detect that `n` is referenced twice in `(+nn)`, and thus the let-expression should not be unfolded. More precisely, the analysis must take all possible execution paths of the body of the let-expression into account. If any of these may have two or more occurrences of the formal let-parameter, then the let-expression is not unfolded.

3.5. Termination of residual programs

Unfolding may discard computations. To avoid discarding a possibly nonterminating computation (controlled by dynamic data) during partial evaluation, we adopt the following conservative strategy: any (nonconstant) residual expression must be present in the residual program. For example, the (intermediate) residual let-expression

```
(let ((n (foo ...)))
 33)
```

will *not* be unfolded since `(foo ...)` might not terminate. Keeping the let-expression of course yields a less reduced residual program.

To ensure that nonconstant expressions are never discarded, a second *occurrence counting analysis* (cf. Section 3.4) is required: if it is not guaranteed that a nonstatic formal parameter of a let-expression will occur at least once on any possible execution path of the let-body, then unfolding is unsafe.

Earlier partial evaluators [10, 36] contained rules for reducing combinations of primitive operations on nonstatic arguments. For example, the expression `(car (cons` E_1 E_2`))` was reduced to E_1, hence discarding E_2—even if evaluation of E_2 were possibly non-terminating. In Similix we take the purist view: such reductions are never performed.

3.6. Call unfolding should neither duplicate nor discard computations

Unfolding procedure calls may also duplicate or discard computations, just as unfolding let-expressions. Let us for example consider the call `(bar (foo ...))`

where bar is defined by

```
(define (bar n) (+ n n)).
```

It is, however, possible to avoid duplicating/discarding when unfolding calls by *inserting let-expressions* (an idea dating back to the work reported in [30]): the call `(bar (foo ...))` can be unfolded to

```
(let ((n (foo ...))) (+ n n)).
```

Technically, this effect can be achieved by *inserting identity let-expressions in the source program*: let-expressions are inserted for all formal procedure parameters. The definition of bar is thus automatically transformed into

```
(define (bar n) (let ((n n)) (+ n n))).
```

This relieves the program specializer from caring about inserting let-expressions (which it has to do in [30]). The inserted let-expressions are treated just like the user-defined ones. For example, one should not unfold the inserted let-expression in the bar definition.

Using inserted let-expressions, the decision of whether to unfold calls reduces to a problem of termination; duplication/discarding need not be considered at all.

3.7. Occurrence counting analysis

The duplication requirement is that nonstatic expressions are not duplicated, i.e. that they occur *at most* once on any possible execution path. The termination property requires that nonstatic expressions are not eliminated, i.e. that they occur *at least* once on any possible execution path.

Both requirements must be fulfilled, so let-expression unfolding can only take place in case of *exactly one* occurrence. Only one occurrence counting analysis is therefore necessary: one that distinguishes between "exactly one occurrence" and "anything else".

3.8. Abstract occurrence counting analysis

The occurrence counting analysis sketched so far reasons over intermediate residual expressions: one first constructs an intermediate residual expression (in which static computations have been performed), then one builds a new simplified residual expression by unfolding let-expressions.

We can, however, in many cases avoid building an intermediate expression: by performing an *approximate* (abstract) occurrence counting analysis reasoning over the *source* expressions, it can be stated that unfolding *any* residual version of a given (source) let-expression is *always* safe. Then there is no reason to build an intermediate residual let-expression first; the unfolded version can be built directly. When unfolding is *possibly unsafe*, it is necessary to build the residual let-expression—and then it can possibly be post-unfolded later.

The abstract occurrence counting analysis does, as we shall see, reason over binding time analyzed source expressions. We describe the abstract occurrence counting analysis in Section 6. Similix also contains a concrete occurrence counting analysis used for post-unfolding residual let-expressions. The concrete analysis is very similar to the abstract one, but it is simpler since it analyses *one* residual expression; on the contrary, the abstract analysis reasons over *any* possible residual version of a given source expression. We shall not go into details of the (relatively straightforward) concrete analysis.

3.9. Global side effects

In general one can consider three classes of side effects:

(1) side effects upon local bindings (with `set!`),
(2) side effects upon a construction (with `set-car!` and related), and
(3) side effects upon global variables (i/o operations, for example).

We shall only consider the third class.

The global variables are accessed and updated by opaque primitive operators (defined by `defprim-opaque`, see Section 2.2); the operations are sequentialized with let-expressions. The global variables are thus accessible everywhere in the programs, but textually they do not occur as parameters being passed around. An example is the input stream accessed by `read`. Hiding global variables is beneficial for the partial evaluator: it need not "worry" about global variables, except when they are actually addressed by the corresponding opaque primitives.

Trying to perform side effects statically (at partial evaluation time) would be problematic. I/o operations obviously need to be kept residual: if they were reduced statically, the residual program would not have the correct semantical behavior (since the residual program would not perform the i/o operations). It would also be difficult to attempt to reduce other side-effecting operations such as store accesses and updates. We can for instance consider a nonreducible conditional (i.e. with a nonstatic test) with update operations in both branches: it is wrong to perform both updates, but at partial evaluation time both branches need to be processed since one does not know which branch to choose.

We take the conservative approach simply to *suspend all side-effecting operations* until run time. This is, however, not sufficient to give side-effecting operations a semantically correct treatment: because of unfolding, there is a risk of duplicating or discarding possibly side-effecting parameter expressions (Sections 3.4 and 3.5). Furthermore, unfolding may *reverse evaluation orders*. This can be exemplified by the following program piece:

```
(let ((n (readport)))
  (let ((m (readport)))
   n))
```

Unfolding the first let-expression neither duplicates nor discards the read operation. But unfolding is still incorrect: the expression

```
(let ((m (readport)))
 (readport))
```

is *not* semantically equivalent to the original one. Side-effecting dynamic expressions are thus "more dangerous" than expressions that are just dynamic: the side-effecting ones must be treated more conservatively.

3.10. A new binding time lattice

Existing abstract interpretation-based binding time analyses, such as described in e.g. [24, 36], distinguish definitely static values (constants) from values which are possibly dynamic, i.e. which may be residual expressions. The static values are described by the binding time value S, the residual expressions by D (we do not consider structured binding time values [27, 30]). The ordering $S \sqsubseteq D$ indicates an asymmetry: a static value may be *lifted* into and treated as a residual expression (for instance `(1 . 2)` $\rightarrow$ `(quote (1 . 2))`), but residual expressions cannot be converted into values. Static values are thus safely abstracted by S (and by D), but residual expressions are only safely abstracted by D.

We also need to describe *possibly side-effecting* expressions. The binding time lattice is therefore extended with a new element X ("external"). The binding time lattice then becomes $(BTValue, \sqsubseteq)$, where

$$BTValue = \{S, D, X\}$$

and $S \sqsubseteq D \sqsubseteq X$.

The binding time value X abstracts residual expressions, which may have side effects on global variables. D thus now abstracts residual expressions, which definitely have no side effects on global variables.

During binding time analysis, the binding time value X is introduced when a primitive operator is declared opaque. For instance, any expression `(read ...)`, where `read` is user-defined as described in Section 2.2, gets binding time value X.

Section 4 describes Similix's binding time analysis in detail.

3.11 Let-expression parameters with global side effects

An earlier version of Similix contained an abstract so-called "evaluation order" analysis. This analysis was used in addition to the abstract occurrence counting analysis to decide when unfolding of a let-expression with external actual parameter expression (i.e. with binding time value X) was safe. The abstract evaluation order analysis would for instance detect that unfolding is unsafe for the expression

```
(let ((n (readport)))
 (let ((m (readport)))
  n))
```

from above (Section 3.9).

The binding time analysis is performed before the abstract occurrence counting and—for this early Similix—evaluation order analyses. It therefore does not know whether let-expressions are unfolded or kept residual, and therefore it has to be conservative to take both possibilities into account. The binding time value for n in the expression `(let ((n (read port))) ...)` depends on whether the let is unfolded or kept residual: in case of unfolding, n must get binding time value X (since it will be bound to the external expression `(read port)` at partial evaluation time), but if the let is kept residual, D suffices (since n will be bound simply to a residual variable, essentially "itself"). But the binding time analysis does not know whether the let is unfolded, so it has to classify n external (X).

All expressions depending on n also become external, and this may prevent later unfoldings. This problem can be clarified by the following example:

```
(let ((x (read port)))
 (let ((y (read port)))
  (let ((z (+ x 1)))
   (if (foo a)
       (+ (read port) z)
       (- z (read port))))))
```

We assume that a is static, so the conditional will be reduced to one of its branches. The evaluation order analysis would prevent unfolding all three let-expressions, also the one defining z which actually could be safely unfolded. The problem is that the actual parameter expression `(+ x 1)` must be classified external even though it definitely does not perform a side effect.

We therefore take another approach which at a first glance may seem very conservative: let-expressions with external actual parameter are *never* unfolded. The advantage of the approach is that formal let-parameters always get binding time value D (or S), but never X. In the above example, the let-expressions defining x and y immediately become non-unfoldable. But x now gets binding time value D, not X, and therefore the let-expression defining z becomes unfoldable. The apparently very conservative approach thus sometimes turns out to be more liberal, and experience has shown that this is the better approach—and further, no abstract evaluation order analysis is needed.

3.12 Maps and environments

Binding time and unfolding annotations will be represented as mappings from expression labels and variables.

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

$\mathscr{L}$: `Label` → *Label*,

$\mathscr{P}$: `ProcedureName` → *Label*,

$\mathscr{V}$: `Variable` → *Variable*.

$\mathscr{L}$ and $\mathscr{V}$ are used for "purity" to convert from syntactic to semantic domains (`Label` is, somewhat artificially, considered a syntactic domain since the labels are part of the extended abstract syntax). $\mathscr{P}$ associates a procedure name with the label of the procedure's body expression: when analyzing a procedure call, this gives access to information about the procedure body. The semantic domains are defined by:

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

$$l \in Label,$$

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

Formal parameters to a procedure P are identified as $(l, 1)$, $(l, 2)$, etc., where $l = \mathscr{P}[\![\mathrm{P}]\!]$. The formal parameter V of a let-expression is identified by some arbitrary unique value v in the domain *Variable*.

The binding time analysis produces two global mappings:

$$\mu_{bt} \in BTMap = Label \to BTValue,$$

$$\rho_{bt} \in BTEnv = Variable \to BTValue.$$

μ_{bt} maps labels (expression results) and ρ_{bt} maps variables to binding time values. A global mapping corresponds to what is called a *cache* in [19]: it associates a value to every expression in the program.

The preprocessing phase also produces two mappings for annotating procedure calls and let-expressions:

$$\mu_{def} \in DefMap = Label \to Annotation,$$

$$\mu_{let} \in LetMap = Label \to Annotation.$$

Here $Annotation = \{Unfold, Resid\}$.

μ_{def} classifies procedures: when $\mu_{def}(\mathscr{P}[\![\mathrm{P}]\!]) = Unfold$, all calls to procedure P are unfolded; when $\mu_{def}(\mathscr{P}[\![\mathrm{P}]\!]) = Resid$, all calls to P are kept residual (specialized). Note that this in effect annotates procedures rather than calls. Let-expressions are annotated by μ_{let}: when $\mu_{let}(\mathscr{L}[\![\mathrm{L}]\!]) = Unfold$ (where L is the label of a let-expression `L (let ((VL-E1)) L-E2))`, the let-expression is (always) unfolded. When $\mu_{let}(\mathscr{L}[\![\mathrm{L}]\!]) = Resid$, the let-expression is kept residual.

3.13. Overview of Similix

We end this section by giving an overview of the phases in Similix.

Partial evaluation is performed in three steps: the source program is preprocessed, then the preprocessed program is specialized, generating an intermediate residual program, and finally the intermediate residual program is postprocessed to produce the final residual program.

The input to the preprocessing phase is the source program (where identity let-expressions have been inserted for the formal procedure parameters) and binding time information about program inputs. The output is a heavily annotated program; the annotations guide the program specializer.

The program specializer is given the (preprocessed) annotated program and the static input values. It produces an intermediate residual program, which is then optimized in postprocessing.

Preprocessing consists of the following phases, performed in that order: binding time analysis (produces μ_{bt} and ρ_{bt}), a phase that adds call unfolding annotations (produces μ_{def}), and finally an abstract occurrence counting analysis (these phases are described in the following sections). Let-expressions are annotated (by μ_{let}) in the binding time and abstract occurrence counting analysis phases. Let-expressions with static actual parameter expression are classified as unfoldable, those with external actual parameter expression are classified as non-unfoldable. The remaining let-expressions (with dynamic parameter) are classified in the abstract occurrence counting phase.

An important point in Similix's preprocessing is that no iteration of the phases is necessary. The binding time analysis need *not* be redone after the annotation of procedure calls and let-expressions.

Specialization performs static computations, unfolds calls and let-expressions, and specializes program points (procedures).

Postprocessing performs additional unfolding of procedure calls and let-expressions.

4. Binding time analysis

This section describes the binding time analysis. Binding time analysis is performed by abstract interpretation and assigns a binding time value to all variables (ρ_{bt}) and all expressions (μ_{bt}, expressions are identified by the labels). The analysis initially assigns the binding time lattice's bottom value S to all variables and all expressions. The user specifies the binding time values of the parameters to the goal procedure. These values are then propagated through the program, updating the global mappings ρ_{bt} and μ_{bt} incrementally until a fixed point is reached.

4.1. Semantic domains and functions

The binding time lattice has already been given earlier:

$$b \in BTValue = \{\mathsf{S}, \mathsf{D}, \mathsf{X}\}$$

where $\mathsf{S} \sqsubseteq \mathsf{D} \sqsubseteq \mathsf{X}$.

The analysis computes a binding time map μ_{bt} and a binding time environment ρ_{bt}. These are updated by corresponding monotonic update functions. Map updating is performed by the function *upd*:

$$upd : Label \to BTValue \to BTMap \to BTMap,$$

$$upd\, l\, b\, \mu_{bt} = \mu_{bt} \sqcup [l \mapsto b] \perp_{BTMap}.$$

Environment updating has functionality

$$\mathit{Variable} \to \mathit{BTValue} \to \mathit{BTEnv} \to \mathit{BTEnv}$$

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:

$$\mu_{bt} \sqcup \mu'_{bt} = \lambda v . \mu_{bt}(v) \sqcup \mu'_{bt}(v),$$

$$\rho_{bt} \sqcup \rho'_{bt} = \lambda v . \rho_{bt}(v) \sqcup \rho'_{bt}(v),$$

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

The binding time values associated with primitive operators are defined by the arity and transparency information given by the user. The function $\mathcal{O}$ associates a binding time function with each operator:

$$\mathcal{O} : \mathtt{OperatorName} \to \mathit{BTValue}^* \to \mathit{BTValue}.$$

We assume that $\mathcal{O}$, for any given program, knows the arity n and transparency information for all operators. Here is the definition of $\mathcal{O}$:

$$\mathcal{O}[\![\mathrm{O}_n^{transparent}]\!][b_1, \ldots, b_n] = \bigsqcup_{i \in \{1, \ldots, n\}} b_i,$$

$$\mathcal{O}[\![\mathrm{O}_n^{dynamic}]\!][b_1, \ldots, b_n] = \bigsqcup_{i \in \{1, \ldots, n\}} b_i \sqcup \mathsf{D},$$

$$\mathcal{O}[\![\mathrm{O}_n^{opaque}]\!][b_1, \ldots, b_n] = \mathsf{X}.$$

4.2. *The analysis*

The function *BT* (see Fig. 3) takes a set of definitions and an initial binding time environment that contains the binding time values of the parameters to the goal procedure. In practice, the user does not provide an initial environment, but the name of the goal procedure and a list of binding time values for the parameters. The function *bt* processes expressions. Explicit quantification of indices has been avoided for readability (the quantification is clear from the context).

4.3. *Comments to the binding time analysis*

Let us now comment the binding time computations.

The binding time value of a constant is trivially S. Processing a constant does not involve any variables, so ρ_{bt} need not be updated.

The binding time value of a variable is the one given by (the current) ρ_{bt}.

For conditionals, we first process the subexpressions. Then, to compute the binding time value of the result of the conditional expression, the least upper bound of the binding time values of the subexpressions is taken. This gives the correct result: if all three subexpressions are guaranteed to specialize to static values, then so will the whole conditional expression. In that case, all three subexpressions will have

```
$BT$ : Definition$^+$ $\rightarrow BTEnv \rightarrow BTMap \times BTEnv$
$BT[\![$(define (P$_1$ ...) L$_1$E$_1$) ... (define (P$_n$ ...) L$_n$E$_n$)$]\!]\rho_{bt}^{init}$ =
  $fix(\lambda(\mu_{bt}, \rho_{bt}) \,.\, \bigsqcup_i bt[\![L_iE_i]\!]\mu_{bt}\rho_{bt} \sqcup (\bot_{BTMap}, \rho_{bt}^{init}))$

$bt$: LabeledExpression $\rightarrow BTMap \rightarrow BTEnv \rightarrow BTMap \times BTEnv$
$bt[\![$L E$]\!]\mu_{bt}\rho_{bt}$ =
  $let\ \ell = \mathcal{L}[\![L]\!]\ in$
    $case\ [\![E]\!]\ of$
      $[\![C]\!] : (upd\ \ell\ S\ \mu_{bt}, \rho_{bt})$
      $[\![V]\!] : (upd\ \ell\ \rho_{bt}(\mathcal{V}[\![V]\!])\ \mu_{bt}, \rho_{bt})$
      $[\![$(if L$_1$E$_1$ L$_2$E$_2$ L$_3$E$_3$)$]\!]$ :
        $let\ (\mu'_{bt}, \rho'_{bt}) = \bigsqcup_i bt[\![L_iE_i]\!]\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu'_{bt}(\mathcal{L}[\![L_i]\!])\ in$
          $(upd\ \ell\ (b_1 \sqcup b_2 \sqcup b_3)\ \mu'_{bt}, \rho'_{bt})$
      $[\![$(let ((V L$_1$E$_1$)) L$_2$E$_2$)$]\!]$ :
        $let\ (\mu'_{bt}, \rho'_{bt}) = \bigsqcup_i bt[\![L_iE_i]\!]\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu'_{bt}(\mathcal{L}[\![L_i]\!])\ in$
          $(upd\ \ell\ (b_1 \sqcup b_2)\ \mu'_{bt}, upd\ \mathcal{V}[\![V]\!]\ (b_1 = X \rightarrow D\ [\!]\ b_1)\ \rho'_{bt})$
      $[\![$(begin L$_0$E$_0$ L$_1$E$_1$ ... L$_n$E$_n$)$]\!]$ :
        $let\ (\mu'_{bt}, \rho'_{bt}) = \bigsqcup_i bt[\![L_iE_i]\!]\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu'_{bt}(\mathcal{L}[\![L_i]\!])\ in$
          $(upd\ \ell\ (\bigsqcup_i b_i)\ \mu'_{bt}, \rho'_{bt})$
      $[\![$(O L$_1$E$_1$ ... L$_n$E$_n$)$]\!]$ :
        $let\ (\mu'_{bt}, \rho'_{bt}) = (\mu_{bt}, \rho_{bt}) \sqcup \bigsqcup_i bt[\![L_iE_i]\!]\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu'_{bt}(\mathcal{L}[\![L_i]\!])\ in$
          $(upd\ \ell\ \mathcal{O}[\![O]\!][b_1, \ldots, b_n]\ \mu'_{bt}, \rho'_{bt})$
      $[\![$(P L$_1$E$_1$ ... L$_n$E$_n$)$]\!]$ :
        $let\ (\mu'_{bt}, \rho'_{bt}) = (\mu_{bt}, \rho_{bt}) \sqcup \bigsqcup_i bt[\![L_iE_i]\!]\mu_{bt}\rho_{bt}\ in\ let\ b_i = \mu'_{bt}(\mathcal{L}[\![L_i]\!])\ in$
          $(upd\ \ell\ (b_1 \sqcup \ldots \sqcup b_n \sqcup \mu'_{bt}(\mathcal{P}[\![P]\!]))\ \mu'_{bt}, \bigsqcup_i(upd\ (\mathcal{P}[\![P]\!], i)\ b_i\ \rho'_{bt}))$
    $end$
```

Fig. 3. Binding time analysis.

binding time value S, and then so will the least upper bound (which then safely abstracts the result of specializing the conditional expression).

If some subexpression specializes to a residual expression (thus abstracted by D), the residual version of the conditional is a residual expression and thus it must be abstracted by a binding time value greater than or equal to D. Further, if some subexpression is (possibly) side-effecting, this residual expression as a whole is (possibly) side-effecting and thus must be abstracted by the binding time value X.

For let-expressions, ρ'_{bt} is updated since the let-expression binds a variable. The formal parameter never gets binding time value X: let-expressions with external actual parameter are not unfolded, and so the formal parameter gets binding time value D (cf. the discussion in Section 3.11).

Every subexpression of a sequence expression is evaluated. The binding time value is therefore simply the least upper bound of the binding time values of the (results of the) subexpressions.

Primitive operations are handled by the function $\mathcal{O}$. Since primitive operations (and procedure calls as well) may be nullary, it is necessary to take the least upper bound with the old values of μ_{bt} and ρ_{bt}.

The most complex case is the one for procedure calls. The binding time value of the procedure call expression is the least upper bound of the arguments *and* the binding time value of the procedure body: both the arguments and the body are evaluated when evaluating a procedure call. The procedure call also influences the formal parameters of the procedure, and therefore ρ'_{bt} is updated. The ith parameter is influenced by the ith argument.

4.4. 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. The analysis is thus guaranteed to terminate.

4.5. Correctness

We will not give a correctness proof for the binding time analysis, but we do give a precise statement of correctness.

Certain *safety* criteria relating program specialization and binding time analysis must be fulfilled: the binding time analysis is correct if and only if μ_{bt} and ρ_{bt} safely abstract the values appearing during specialization (it was defined in Section 3.10 what the binding time values abstract). Assuming that the binding time values of the program input (specified in ρ_{bt}^{init}) safely abstract any input values with respect to which the program is specialized, correctness can be stated as follows:

- Safety of μ_{bt}: for any program expression L E, $\mu_{bt}(\mathcal{L}[\![\mathrm{L}]\!])$ safely abstracts any possible result of specializing L E.
- Safety of ρ_{bt}: for any program variable V, $\rho_{bt}(\mathcal{V}[\![\mathrm{V}]\!])$ safely abstracts any value that V may be bound to during program specialization.

A formal correctness proof would require a formal specification of program speciliz-ation. The proof would be done by structural induction, relating the specialization specification to the binding time analysis, much like we informally did above when explaining the binding time analysis (Section 4.3).

4.6. 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 efficiency point of view. Using sequential processing there is always only *one* active copy of μ_{bt} and of ρ_{bt}. The mappings are thus single-threaded and can be updated destructively; further, they can be implemented as global variables.

In practice, the mappings are not kept as separate variables. Instead, the information is kept as *attributes* (annotations) in the abstract syntax.

4.7. Independence of unfolding annotations

The binding time value of a let-expression with dynamic actual parameter is independent of the subsequent unfolding annotation, performed by the abstract occurrence counting analysis. The binding time analysis "knows" that a let-expression like

$$\texttt{(let ((x}\,E_{dynamic}\texttt{))}\,E_{static}\texttt{)}$$

will never be unfolded (in which case the result of the let-expression could be static); it immediately assigns D to the result of the let-expression—the least upper bound of the actual parameter and body expressions. Abstract occurrence counting eventually disallows unfolding because $E_{dynamic}$ cannot occur inside E_{static}.

A similar consideration holds for procedure calls: the binding time value of the result of a procedure call is independent of the subsequent annotation of the procedure call. The only procedure calls which have a static result are those with only static parameters and static body, and such calls are always unfolded anyway.

Because binding time analysis is independent of let and call annotations, it must not be redone after adding unfolding annotations.

5. Automatic call unfolding in Similix

5.1. Background: automatic call unfolding in Mix

The first Mix [23] required user-guided annotations for controlling call unfolding, but this was later automated [24]. The techniques used in the automatic version are described in detail in [38].

The process of automatic call unfolding in Mix is twofold: some function calls, annotated unfoldable in preprocessing, are unfolded "on the fly" during specialization. However, many trivial functions remain in the residual program, and some of these are reduced away in postprocessing by unfolding the calls to them.

It is obviously desirable to perform as much unfolding as possible already during program specialization. This produces the residual program piece directly: one avoids building intermediate specialized functions, which are removed again in postprocessing.

It is important that the decision on whether to unfold a call during program specialization has been taken purely on the basis of binding time information. The reason is self-application: the more decisons taken on the basis of only binding time information, the better self-application results (smaller and faster compilers) [8]. Decisions that depend only on binding time information can be performed in preprocessing and thus need not be performed during program specialization; this is the reason why Mix annotates calls in preprocessing. The idea in preprocessing is to find some calls which can be safely unfolded. No other calls will be unfolded during program specialization (although some may be post-unfolded later).

Mix uses both termination and duplication criteria to decide whether a call can safely be unfolded. A call may be safe to unfold from a termination point of view, but unfolding may duplicate an argument expression. In that case Mix will not unfold the call. In the description of Mix below, we do, however, not consider duplication. Only Mix's termination analysis is of interest for comparison with Similix (in Similix, call unfolding never duplicates, cf. Section 3.6).

Mix's *pre*processing annotates calls with completely static arguments as unfoldable (cf. Section 3.3). For all other calls, it detects primitive recursion loops (functions calling themselves) in which at least one static parameter becomes smaller for each recursion; such a parameter is called an *inductive variable.* Recursive calls with inductive variables can safely be unfolded during program specialization, provided the partial ordering on static values is well-founded (with no endless descending chains): eventually, the inductive variable will reach the smallest value, so infinite unfolding is impossible. The well-founded ordering used in Mix is the proper subterm ordering on acyclic S-expressions (Mix's only data structure): a term is greater than its proper subterms. Application of the primitives `car` and `cdr` produce smaller terms: `(car E)` and `(cdr E)` are always smaller than E (taking `car`/`cdr` of an atomic value gives an error).

Mix's *post*processing starts by performing a so-called *call graph analysis* of the (finite) residual program. The call graph is a directed graph representing all dependencies between calling functions and called functions. Nodes represent functions and arcs represent calls from one function to another one. Program loops are reflected by cycles in the graph. Unfolding is then performed in such a way that one does not go around in cycles: a *cut point* is chosen for each cycle. This guarantees a finite post-unfolding.

The automatic call unfolding of Similix described below is also based on annotations produced in preprocessing (the mapping μ_{def}) and additional unfolding performed in post-unfolding. However, the methods used in pre- and postprocessing are significantly simpler than those of Mix: preprocessing does not require any recursion analysis, but only relies on the binding time analysis; and postprocessing does not require an expensive call graph analysis.

5.2. Choosing dynamic conditionals as specialization points

Let us state our basic observation: any nontrivial loop contains at least one conditional for deciding whether to stop or continue looping. Loops without such a conditional never terminate; if a program contains such a loop, we do not take any responsibility (we then accept that partial evaluation does not terminate). (*Note*: all primitive operations are assumed terminating.)

If the test of the conditional is static, the loop is controlled statically: it is controlled statically whether to stop or continue looping. This makes it reasonable to unfold the recursion: the unfolding process will stop when the static test chooses the stop branch. An infinite unfolding loop can only be entered if caused by static data. But in that case, a standard evaluation of the program—with any value for the dynamic part of the input data—would also loop (if the looping part were entered), so we accept that partial evaluation loops too. Note the similarity with the argument for completely static computations used in Section 3.3

On the other hand, if the test of the conditional were dynamic, it would indeed be a bad idea to unfold the recursion: the specializer specializes both branches of a dynamic conditional, the "continue part" and the "stop part", and so can never reduce it to its "stop part". Infinite unfolding could result. Therefore, dynamic conditionals are chosen as specialization points to break unfolding. Let us concretize this idea now.

We first observe that it cannot immediately be deduced where a program contains a loop. Programs essentially consist of recursive equations, and there is no special syntactic loop construction. One could perform a static "loop detection" analysis, but we choose a simpler solution: insert specialization points for *all* dynamic conditionals, independently of whether they control a recursion.

One therefore sometimes gets many small specialized procedures in residual programs: these can, however, be removed in post-unfolding. Thus, by sacrificing some unfolding during program specialization and performing it in post-unfolding instead, a considerably simpler preprocessing is achieved. And as we shall see below, post-unfolding can be made very efficient, so we believe that the trade-off is worthwhile.

That dynamic conditionals are "dangerous" with respect to termination can also be understood in terms of *strictness*. Partial evaluation is stricter, and thus *less terminating*, than standard evaluation when—and only when—processing dynamic conditionals: standard evaluation evaluates only one of the branches, but partial evaluation specializes both (cf. Section 3.3).

Choosing dynamic conditionals as specialization points turns out to work surprisingly well, also for other conditionals than those controlling recursion. For example, dynamic conditionals are the ideal specialization points when considering string pattern matching [13]. Specialization yields residual programs that exhibit a considerably increased amount of sharing [28]. Interestingly enough, the idea of using dynamic conditionals as specialization points has been used in other language contexts, but never for autoprojectors for Lisp-like recursive equation languages. Gomard and Jones [18] thus report an autoprojector for an imperative assembly-style language using this idea. Turchin [41] and Bondorf [2] essentially use the same idea for functional languages based on pattern matching.

5.3. Insertion of new procedure calls

In partial evaluators for recursive equation languages, the specializable program points have traditionally been reduced to be only user-written procedures (functions). In Similix, we use a new idea: to be able to specialize *any* program point, we *replace* the expression in question by a call to a *new procedure* whose body is that expression. The parameters to the procedure are the free variables in the expression. These procedures then serve as the *only* specialization points: all calls to user defined procedures are unfolded.

Since we have chosen dynamic conditionals to be the (only) specialization points, we replace all dynamic conditional expressions by such new procedure calls, and corresponding new procedures are generated. To deal with nested dynamic conditional expressions, the process proceeds recursively for the new procedures. This is important when dealing with embedded dynamic conditionals where the inner ones depend on fewer static values than the outer ones: (residual versions of) the inner ones may be shared even though the outer ones are not.

Let us give a simple example of procedure call insertion. Suppose the program piece

```
... (if (foo a)
        (if (bar x)
            (car y)
            (cdr y))
        (car z))
```

has been binding time analyzed, and x, y, and z turn out to be dynamic (or external, that makes no difference) whereas a is static. This program piece is replaced by

```
... (if (foo a)
        (new x y)
        (car z))

(define (new x y)
  (let ((x x))
    (let ((y y))
      (if (bar x)
          (car y)
          (cdr z)))))
```

where new is a new name, which is annotated with *Resid* in μ_{def}.

Identity let-expressions are inserted for all formal parameters to the inserted procedures (just as for the user-written ones, cf. Section 3.6). Such a new let-expression is annotated as unfoldable if it has a static actual parameter variable; if the actual parameter is dynamic, the let-expression is annotated in the abstract occurrence counting phase just as any other let-expression. An external actual parameter variable would give rise to a non-unfoldable let-expression; it turns out, however, that the actual parameters cannot possibly be external because the actual parameters of the inserted procedure call are all variables.

An obvious optimization, implemented in Similix, is to avoid inserting new calls for dynamic conditionals occurring outermost (apart from the inserted let-expressions) in procedure bodies, for instance when processing the append program (Section 3.2) with dynamic xs and static ys. Here append itself is simply annotated with *Resid.*

5.4 Comparison of Mix and Similix preprocessing

Mix needs loop detection, whereas Similix does not. Detecting loops in Mix is rather primitive: only direct recursive calls are detected, not mutually recursive ones (procedures calling each other). Hence mutual recursion is treated very conservatively, with no unfolding during program specialization. For statically controlled structural induction-like loops such as the parsing performed in an "eval" loop of a typical interpreter, Mix will only unfold the primitive recursive "eval" calls; Similix will also unfold the mutually recursive ones. This makes a difference if the user has written the "eval" procedure such that it uses special procedures ("eval-if", "eval-while", etc.) for dealing with the different syntactic constructs.

Mix relies on knowledge about the fixed set of primitives used there, for instance that car reduces the size of a structure. Similix requires no knowledge of that kind.

As pointed out above, using dynamic conditionals as specialization points gives nice sharing properties in residual programs. This sharing is difficult to achieve in Mix.

The termination properties only vary slightly: Mix terminates in a few cases where Similix does not. This is the case for statically controlled loops where at least one dynamic parameter is passed around, but never nested: Mix never enters an infinite unfolding loop in such a case, but Similix may. For purely static computations, Mix and Similix both evaluate completely and so have identical termination properties. There seems to be no good reason why the termination of a statically controlled loop should depend on whether some dynamic parameter incidentally is carried around.

Neither Mix nor Similix guarantee termination; there is always the possibility of infinite specialization. Infinite specialization can always be avoided by generalization (forcing static expressions to become dynamic, cf. Section 2.2), but finding generalization points is in general undecidable. The best one can do is to make approximative analyses, which on the one hand do guarantee termination (safety), but on the other hand may be too conservative (generalizing unnecessarily much) in some cases. The problem is analyzed in great detail in [21]. Jones proposes different algorithms for ensuring safety; the algorithms pay special attention to specialization of interpreters, and some of the analyses correspond to the analysis performed by the Mix preprocessing (loop detection and detecting inductive static structures).

5.5. Postprocessing in Similix

Postprocessing unfolds calls to "trivial" procedures in the residual program. Post-unfolding is independent from and performed after program specialization.

Post-unfolding could, for instance, transform the (residual) program piece

```
(define (f x y)
 (if (foo x)
     (g y)
     176))

(define (g z)
 (if (h z)
     671
     716))
```

into the simpler program piece

```
(define (f x y)
 (if (foo x)
     (if (h y)
         671
         716)
     176))
```

Care must be taken so that post-unfolding does not enter a nonterminating unfolding loop. In Mix, this is handled by the call graph analysis, which finds cycles in the graph.

The Similix post-unfolding performs no graph analysis, but relies on the following simple observation: a loop will always contain at least one procedure, to which there are at least two calls. There is an intial call for entering the loop and one or more recursive calls. In graph terminology: any cycle contains at least one node towards which at least two arcs are directed. We note that the goal procedure, to which the initial call is performed, is a special case since the initial call is not explicitly present in the program. In graph terminology, the goal procedure corresponds to the root node.

The strategy for post-unfolding in Similix then follows: if a procedure is called only once, the call is unfolded. If a procedure is called more than once, none of the calls to it are unfolded. This guarantees termination of the post-unfolding.

This post-unfolding strategy implies that post-unfolding never duplicates code of procedure bodies, but keeps sharing. This has consequences when a procedure is called nonrecursively from two different places in the program: Similix does not unfold such calls (keeping sharing), but Mix only considers cycles and thus does unfold (duplicating function bodies and thereby destroying sharing).

The implementation of the Similix strategy is very simple: during specialization, a one-bit reference counter is associated with each residual procedure. During postprocessing, if a procedure is called only once, its call is post-unfolded. No call graph needs to be analyzed.

6. Abstract occurrence counting analysis

This section describes the abstract occurrence counting analysis (cf. Section 3.8) and the related *raising* of annotations (from *Unfold* to *Resid*) of let-expressions with actual parameter expression with binding time vale D. The analysis safely approximates the number of occurrences of a dynamic parameter on *any* possible execution path of *any* possible residual version of the analyzed source expression. If this number is "always exactly once", the let-expression can be unfolded safely during program specialization and thus it can be annotated with *Unfold.* Otherwise it must be annotated with *Resid* since duplication/discarding is possible.

The idea is to compute an abstract count for all nonstatic variables, let-parameters, and as well all procedure parameters. Let-expressions with dynamic actual parameter are initially all annotated unfoldable (*Unfold*); the annotations are then raised according to the following algorithm: compute an abstract occurrence count environment ρ_{oc} that contains abstract counts for all nonstatic variables; then, if there exists an unfoldable let-expression whose actual parameter expression is dynamic and whose formal parameter's abstract occurrence count is different from "exactly once", raise the annotation of the let-expression to *Resid* (i.e. update μ_{let}) and repeat (computing a new ρ_{oc} etc.), otherwise stop.

Abstract occurrence counts depend on other counts, so ρ_{oc} is defined recursively. Therefore it is necessary to compute counts not only for variables with binding time value D, but also for the ones with binding time value X: dynamic variables may be used in expressions that during program specialization become bound to variables with binding time value X (recall that $D \sqsubseteq X$), and the counts for these external variables influence the counts for the dynamic variables in question. On the other hand, there is no need to count occurrences of static variables: dynamic variables cannot possibly be used in expressions that during partial evaluation become bound to variables with binding time value S.

Counts for formal let-parameters depend on counts for procedure parameters. Therefore occurrence counts are computed not only for (nonstatic) let-bound variables, but also for (nonstatic) procedure parameters.

The algorithm for annotating dynamic let-expressions always terminates: annotatations are only raised, never lowered. In the worst case, all the dynamic let-expressions become residual.

The abstract count for a let-parameter is computed by analyzing the body of the let-expression. For procedure parameters, the procedure body is analyzed. The computation of ρ_{oc} depends on the current annotation of let-expressions and ρ_{oc} therefore has to be recomputed after each annotation raising. Because ρ_{oc} is defined recursively, every computation of it is itself a fixed point iteration.

It turns out that considerable simplification is possible. The computation of ρ_{oc} can be made independent of the current annotations, so recomputing it after each raising is not necessary. It also turns out that the recursive dependencies vanish, so no fixed point iteration is needed to compute ρ_{oc}.

6.1. *The abstract occurrence count lattice*

Abstract occurrence counting is computed over the lattice

$$(\mathit{AbstractCount}, \sqsubseteq),$$

where

$$c \in \mathit{AbstractCount} = \{\bot^{\#}, 0^{\#}, 1^{\#}, \mathit{any}^{\#}\}$$

and where the partial ordering is given by

$$\begin{tikzcd}
 & \mathit{any}^{\#} & \\
0^{\#} \arrow[ur, no head] & & 1^{\#} \arrow[ul, no head] \\
 & \bot^{\#} \arrow[ul, no head] \arrow[ur, no head] &
\end{tikzcd}$$

The lattice is an abstraction of a concrete domain, the lifted flat domain $Nat_{\bot} = \{0, 1, 2, 3, \ldots\}_{\bot}$. The values in the concrete domain count occurrences of a variable on an execution path; the concrete domain is lifted to account for nontermination, which corresponds to an infinite execution path.

The abstract values are related to the concrete ones in the following way: $0^{\#}$ abstracts 0, $1^{\#}$ abstracts 1, and $\mathit{any}^{\#}$ abstracts any natural number (including 0). $\bot^{\#}$ means "no value yet" (the initial value before fixed point iteration) and abstracts $\bot$.

One could have chosen a simpler lattice containing just the two values $1^{\#}$ and $\mathit{any}^{\#}$. However, 0 would then need to be abstracted by $\mathit{any}^{\#}$, and this would have given unnecessarily conservative results when analyzing compound expressions. For instance, as we shall see below, we sometimes add abstract counts. Adding $1^{\#}$ and $0^{\#}$ thus gives $1^{\#}$, whereas we would get $\mathit{any}^{\#}$ with the simpler domain (there we would need to add $1^{\#}$ and $\mathit{any}^{\#}$).

A let-expression is only unfolded when the abstract count is $1^{\#}$; this value precisely means "exactly one occurrence". Since $\bot^{\#} \sqsubseteq 1^{\#}$, it might be argued that we then also have to unfold when the abstract count is $\bot^{\#}$. However, in the simplified version of the analysis (presented later in this section), $\bot^{\#}$ never occurs in practice. The simplified analysis is slightly more conservative, and it may indeed happen that it gives $1^{\#}$ in situations where the fixed point analysis would give $\bot^{\#}$. However, since $\bot^{\#}$ abstracts computations that are always nonterminating, either at partial evaluation time or when running the residual program, it does not matter whether the let-expression is annotated with *Unfold* or *Resid* when the count is $\bot^{\#}$.

Let us define some operations over the counting lattice. To analyze compound expressions, we need an operator $+^{\#}$ to add abstract counts. A let-expression factors a value to avoid its multiple computation, so we need an operator $\times^{\#}$ to multiply abstract counts. Conditional expressions reduce to one expression out of two, so we need an operator $\sqcup$ to take the least upper bound of abstract counts. We therefore define:

$$+^{\#} : AbstractCount \times AbstractCount \rightarrow AbstractCount,$$

$$\times^{\#} : AbstractCount \times AbstractCount \rightarrow AbstractCount,$$

$$\sqcup : AbstractCount \times AbstractCount \rightarrow AbstractCount,$$

where $+^{\#}$ and $\times^{\#}$ are defined by the tables in Fig. 4. The precedence rules for $+^{\#}$ and $\times^{\#}$ are the ordinary ones. The least upper bound operator $\sqcup$ is defined by the partial ordering $\sqsubseteq$.

$+^{\#}$	$\perp^{\#}$	$0^{\#}$	$1^{\#}$	$any^{\#}$
$\perp^{\#}$	$\perp^{\#}$	$\perp^{\#}$	$\perp^{\#}$	$\perp^{\#}$
$0^{\#}$	$\perp^{\#}$	$0^{\#}$	$1^{\#}$	$any^{\#}$
$1^{\#}$	$\perp^{\#}$	$1^{\#}$	$any^{\#}$	$any^{\#}$
$any^{\#}$	$\perp^{\#}$	$any^{\#}$	$any^{\#}$	$any^{\#}$

$\times^{\#}$	$\perp^{\#}$	$0^{\#}$	$1^{\#}$	$any^{\#}$
$\perp^{\#}$	$\perp^{\#}$	$\perp^{\#}$	$\perp^{\#}$	$\perp^{\#}$
$0^{\#}$	$\perp^{\#}$	$0^{\#}$	$0^{\#}$	$0^{\#}$
$1^{\#}$	$\perp^{\#}$	$0^{\#}$	$1^{\#}$	$any^{\#}$
$any^{\#}$	$\perp^{\#}$	$0^{\#}$	$any^{\#}$	$any^{\#}$

Fig. 4. Definitions of $+^{\#}$ and $\times^{\#}$.

The operations $+^{\#}$, $\times^{\#}$, and $\sqcup$ can be verified to be commutative, associative, and monotonic (this is easily deduced from the tables). The operations $+^{\#}$ and $\times^{\#}$ abstract addition and multiplication over the domain $Nat_{\perp}$, i.e. for all natural numbers n_1 and n_2 the following relations hold:

$$abs(n_1 + n_2) \sqsubseteq abs(n_1) +^{\#} abs(n_2),$$

$$abs(n_1 \times n_2) \sqsubseteq abs(n_1) \times^{\#} abs(n_2).$$

$abs : Nat_{\perp} \rightarrow AbstractCount$ is the abstraction function. We note that abstraction is not defined on the powerdomain of $Nat_{\perp}$ but on $Nat_{\perp}$ itself. A similar approach to abstraction is found in [20].

The abstract occurrence count environment ρ_{oc} maps variables to occurrence counts:

$$\rho_{oc} \in OCEnv = Variable \rightarrow AbstractCount.$$

6.2. *Computing the abstract occurrence count environment*

The function *OC* computes ρ_{oc} as a fixed point (see Fig. 5). It uses the function *oc* to process expressions. The lattice *AbstractCount* is finite and *oc* is monotonic since $\sqcup$, $+^{\#}$, and $\times^{\#}$ are, so the fixed point will be reached after a finite number of iterations.

6.3. *Comments on the abstract occurrence counting analysis*

μ_{let} initially maps all let-expressions with actual parameter expression with static or dynamic result to *Unfold*. Those with external actual parameter expression are

```
$OC$ : Definition$^+$ $\rightarrow$ $LetMap$ $\rightarrow$ $DefMap$ $\rightarrow$ $OCEnv$
$OC[\![$PD+$]\!]\mu_{let}\mu_{def} =$
  $fix(\lambda\rho_{oc} . \lambda(\ell, i)$ .
    $((\exists$ $a$ $definition$ $[\![$(define (P V$_1$ ... V$_n$) L E)$]\!]$ $in$ $[\![$PD+$]\!]$ : $\ell = \mathcal{P}[\![$P$]\!] \wedge i \leq n)$   (1)
     $\vee$
     $(\exists$ $an$ $expression$ $[\![$L$_0$ (let ((V L$_1$E$_1$)) L E)$]\!]$ $in$ $[\![$PD+$]\!]$ : $(\ell, i) = \mathcal{V}[\![$V$]\!])$ )   (2)
    $\wedge$ $\rho_{bt}(\ell, i) \sqsupseteq$ D   (3)
    $\rightarrow$ $oc[\![$L E$]\!](\ell, i)\rho_{oc}\mu_{let}\mu_{def}$
    $\square$ $\bot^{\#})$   (4)

oc : LabeledExpression $\rightarrow$ $Variable$ $\rightarrow$ $OCEnv$ $\rightarrow$ $LetMap$ $\rightarrow$ $DefMap$
                                                                   $\rightarrow$ $AbstractCount$
$oc[\![$L E$]\!]v\rho_{oc}\mu_{let}\mu_{def} =$
  $let$ $\ell = \mathcal{L}[\![$L$]\!]$ $in$
    $\mu_{bt}(\ell) = $ S $\rightarrow 0^{\#}$
    $\square$ $case$ $[\![$E$]\!]$ $of$
        $[\![$C$]\!]$ : $0^{\#}$
        $[\![$V$]\!]$ : $\mathcal{V}[\![$V$]\!] = v \rightarrow 1^{\#} \square 0^{\#}$
        $[\![$(if L$_1$E$_1$ L$_2$E$_2$ L$_3$E$_3$)$]\!]$ :
          $oc[\![$L$_1$E$_1]\!]v\rho_{oc}\mu_{let}\mu_{def} +^{\#} (oc[\![$L$_2$E$_2]\!]v\rho_{oc}\mu_{let}\mu_{def} \sqcup oc[\![$L$_3$E$_3]\!]v\rho_{oc}\mu_{let}\mu_{def})$
        $[\![$(let ((V L$_1$E$_1$)) L$_2$E$_2$)$]\!]$ :
          $let$ $c_1 = oc[\![$L$_1$E$_1]\!]v\rho_{oc}\mu_{let}\mu_{def}$ , $c_2 = oc[\![$L$_2$E$_2]\!]v\rho_{oc}\mu_{let}\mu_{def}$ $in$
            $\mu_{let}(\ell) = Resid \vee \mu_{bt}(\mathcal{L}[\![$L$_1]\!]) = $ S $\rightarrow c_1 +^{\#} c_2$
            $\square$ $let$ $c_2' = oc[\![$L$_2$E$_2]\!](\mathcal{V}[\![$V$]\!])\rho_{oc}\mu_{let}\mu_{def}$ $in$
                $c_1 \sqsupseteq 1^{\#} \wedge c_2' \neq 1^{\#} \rightarrow c_1 +^{\#} c_2 \square c_1 \times^{\#} c_2' +^{\#} c_2$
        $[\![$(begin L$_0$E$_0$ L$_1$E$_1$ ... L$_n$E$_n$)$]\!]$ :
          $oc[\![$L$_0$E$_0]\!]v\rho_{oc}\mu_{let}\mu_{def} +^{\#} oc[\![$L$_1$E$_1]\!]v\rho_{oc}\mu_{let}\mu_{def} +^{\#} \ldots +^{\#} oc[\![$L$_n$E$_n]\!]v\rho_{oc}\mu_{let}\mu_{def}$
        $[\![$(O L$_1$E$_1$ ... L$_n$E$_n$)$]\!]$ : $oc[\![$L$_1$E$_1]\!]v\rho_{oc}\mu_{let}\mu_{def} +^{\#} \ldots +^{\#} oc[\![$L$_n$E$_n]\!]v\rho_{oc}\mu_{let}\mu_{def}$
        $[\![$(P L$_1$E$_1$ ... L$_n$E$_n$)$]\!]$ :
          $let$ $c = oc[\![$L$_1$E$_1]\!]v\rho_{oc}\mu_{let}\mu_{def} \times^{\#} \rho_{oc}(\mathcal{P}[\![$P$]\!], 1) +^{\#} \ldots +^{\#}$
                  $oc[\![$L$_n$E$_n]\!]v\rho_{oc}\mu_{let}\mu_{def} \times^{\#} \rho_{oc}(\mathcal{P}[\![$P$]\!], n)$
          $in$ $\mu_{def}(\mathcal{P}[\![$P$]\!]) = Unfold \rightarrow c$
               $\square$ $c \sqcup oc[\![$L$_1$E$_1]\!]v\rho_{oc}\mu_{let}\mu_{def} +^{\#} \ldots +^{\#} oc[\![$L$_n$E$_n]\!]v\rho_{oc}\mu_{let}\mu_{def}$
    $end$
```

Fig. 5. Abstract occurrence counting analysis. Line (1) in the definition of OC accounts for formal procedure parameters, line (2) for let-bound parameters. Only nonstatic variables are of interest (3), and only those actually occurring in the program (4).

mapped to *Resid* (cf. Section 3.13). μ_{def} maps procedure names to either *Unfold* or *Resid*: user-defined procedures are mapped to *Unfold*, the inserted ones to *Resid* (see Section 5.3).

A static expression always reduces to a constant in the residual program, so the (non-static) variable being counted can never occur in the residual version of a static expression. The count therefore is $0^{\#}$. The same holds for constants. This case is actually redundant since constants always get binding time value S.

The occurrence count for a variable is either $1^{\#}$ or $0^{\#}$, according to whether it is the variable v currently being counted.

If the test of a conditional expression is static, the conditional reduces to one of its branches. In that case, the count is the least upper bound of the two branch counts. If the test is nonstatic, a residual conditional is generated. When evaluating this residual conditional, the test and one of the branches are executed. The count therefore is the sum of the test count and the least upper bound of the branch counts. Since a static test implies that the test count is $0^{\#}$, this sum gives the correct count, also in the case of a static test.

The residual version of a let-expression annotated *Resid* contains a residual version of the parameter expression and a residual version of the body. The count therefore simply is the sum of the two counts. If the actual parameter expression E_1 is static, the let-expression will be unfolded (Section 3.13) and (the nonstatic) v cannot occur in E_1; the count is therefore simply c_2, but since c_1 is now $0^{\#}$, the count $c_1 +^{\#} c_2$ is still valid.

Unfolding a let-expression with nonstatic actual parameter expression gives two sources of occurrences: the "ordinary" ones in the (residual version of the) let-body and the indirect ones caused by occurrences of the actual parameter expression in the let-body. This gives the count $c_1 \times^{\#} c_2' +^{\#} c_2$.

In some cases, however, one can foresee that even though the let-expression currently is annotated *Unfold*, it must eventually be raised to *Resid*. This happens when the condition $c_1 \sqsupseteq 1^{\#} \wedge c_2' \neq 1^{\#}$ is fulfilled (notice that in that case the count $c_1 +^{\#} c_2$ is used, the same as for let-expressions annotated *Resid*). The condition states that if there is a possible occurrence of v in the parameter expression ($c_1 \sqsupseteq 1^{\#}$) and if the count for the formal parameter of the let-expression is different from "exactly once" ($c_2' \neq 1^{\#}$), then we already know that the let-expression—if c_2' remains different from $1^{\#}$—eventually will be raised to *Resid.*

The reason is that if $c_1 \sqsupseteq 1^{\#}$, then E_1 must have a nonstatic binding time value (when $c_1 \sqsupseteq 1^{\#}$, v occurs in E_1 which is therefore at least D), and since $c_2' \neq 1^{\#}$, the let-expression will be raised to *Resid.* It may also happen that c_2' eventually becomes $1^{\#}$ becuase of new annotations in the let-body; in that case the let-expression remains unfoldable, but since

$$c_1 \times^{\#} c_2' +^{\#} c_2 = c_1 +^{\#} c_2 \quad \text{when } c_2' = 1^{\#},$$

it is still correct to use the count $c_1 +^{\#} c_2$ for residual let-expressions.

The residual version of a sequence expression is a constant if the operation is reduced (this happens if all subexpressions are static), otherwise it is the sequence expression itself with residual versions of the subexpressions. In the former case, the count is $0^{\#}$, in the latter it is the sum of the counts of the subexpressions. This sum trivially reduces to $0^{\#}$ when all subexpressions are static, and so the sum can be used in both cases.

Like sequence operations, primitive operations are strict in all subexpressions. The count is therefore simply the sum of the counts of the subexpressions.

The treatment of an unfoldable procedure call is similar to the primitive operator case (summing over the arguments), but each actual parameter must be treated in a way similar to an unfoldable let (multiplying with the count for the formal procedure parameter). The residual version of a procedure call annotated *Resid* is a residual procedure call. In that case, the sum of the counts for the arguments is simply taken (as for primitive operations). This residual procedure call may, however, be post-unfolded. When that happens, the call must be treated as if it had been annotated *Unfold*. The abstract count for calls annotated *Resid* therefore is "$c \sqcup \ldots$".

The analysis does not consider *code* duplication [38], only computation duplication. For example, the variable y would get the abstract count $1^{\#}$ in `(if E y y)` (where y does not occur free in E). This may result in residual programs with duplicated code. It is straightforward to make a more conservative analysis that prevents such code duplication. This is done by changing the count for conditionals into

$$\begin{aligned} &\mathit{let}\ c_2 = oc[\![\mathrm{L}_2\mathrm{E}_3]\!]\, v\rho_{oc}\mu_{let}\mu_{def},\ c_3 = oc[\![\mathrm{L}_3\mathrm{E}_3]\!]\, v\rho_{oc}\mu_{let}\mu_{def}\ \mathit{in} \\ &\quad oc[\![\mathrm{L}_1\mathrm{E}_1]\!]\, v\rho_{oc}\mu_{let}\mu_{def} +^{\#} ((c_2 \sqcup c_3) \sqcup (c_2 +^{\#} c_3)). \end{aligned}$$

6.4. Correctness

As for the binding time analysis, we will not give a correctness proof for the abstract occurrence counting analysis, but we do give a precise statement of correctness.

The abstract occurrence counting analysis is correct if and only if the resulting abstract occurrence counting environment ρ_{oc} fulfills the following safety requirement: for any nonstatic program variable V defined either in a procedure definition

```
(define (P...V...)LE)
```

or in a let-expression

```
L0 (let ((V L1E1)) L E)
```

(where nonstatic means $\rho_{bt}(\mathcal{V}[\![\mathrm{V}]\!]) \sqsupseteq \mathrm{D}$), $\rho_{oc}(\mathcal{V}[\![\mathrm{V}]\!])$ safely abstracts the number of occurrences n of V on any possible execution path of any possible residual version of the expression E (i.e. $\rho_{oc}(\mathcal{V}[\![\mathrm{V}]\!]) \sqsupseteq abs(n)$).

A formal correctness proof would be quite complex, involving both reasoning over program specialization and over executing the generated residual code.

6.5. Simplifying the analysis

The counting analysis can be much simplified without losing significant precision. This will be done now.

Lemma 1. *oc (and consequently OC) can be made independent of* μ_{let}.

Proof. μ_{let} is only referred to in the *oc* rule for let-expressions. Here we observe that

$$\neg(c_1 \sqsupseteq 1^{\#} \wedge c_2' \neq 1^{\#}) \Leftrightarrow \neg(c_1 \sqsupseteq 1^{\#}) \vee c_2' = 1^{\#}$$
$$\Leftrightarrow c_1 = \bot^{\#} \vee c_1 = 0^{\#} \vee c_2' = 1^{\#}.$$

In that case it is easy to verify that

$$c_1 \times^{\#} c_2' +^{\#} c_2 \sqsubseteq c_1 +^{\#} c_2,$$

so it is safe to reduce the count for let-expressions to simply $c_1 +^{\#} c_2$.

This gives a loss of precision, but note that $c_1 \times^{\#} c_2' +^{\#} c_2 = c_1 +^{\#} c_2$ except for the rather uninteresting case $c_1 = 0^{\#} \wedge c_2' = \bot^{\#}$ (uninteresting since $\bot^{\#}$ abstracts nontermination). Practically, we therefore do not lose anything by approximating $c_1 \times^{\#} c_2' +^{\#} c_2$ with $c_1 +^{\#} c_2$. □

Lemma 2. *Abstract occurrence counts for formal procedure parameters are always* $1^{\#}$.

Proof. We know that all procedure bodies begin with the inserted let-expressions of the form `(let ((x x)) E)`. The only free occurrence of a formal procedure parameter is in the corresponding inserted let-expression (cf. Section 3.6). The count therefore trivially is $1^{\#}$, using Lemma 1 for counting when processing the inserted let-expressions. □

Lemma 3. *oc (and consequently OC) is independent of* μ_{def}.

Proof. μ_{def} is used when processing procedure calls. But using Lemma 2 and that $c \times^{\#} 1^{\#} = c$ (for any c), the count for procedure calls can be simplified to

$$oc[\![\mathrm{L}_1\mathrm{E}_1]\!] v\rho_{oc}\mu_{let}\mu_{def} +^{\#} \cdots +^{\#} oc[\![\mathrm{L}_n\mathrm{E}_n]\!] v\rho_{oc}\mu_{let}\mu_{def},$$

independently of the test. □

Lemma 4. *oc is independent of* ρ_{oc}.

Proof. In *oc*, ρ_{oc} is only used for finding counts for formal procedure parameters. But these are always trivially $1^{\#}$ (Lemma 2). □

Proposition 1. ρ_{oc} *need not be computed as a fixed point.*

Proof. Follows from Lemma 4: the recursive dependency has vanished. □

The following proposition is important: it formally justifies that the inserted identity let-expressions reduce the duplication/discarding problem to a question of unfolding let-expressions.

Proposition 2. *Computation duplication/discarding never occurs due to procedure call unfolding.*

Proof. Follows from Lemma 2: all formal procedure parameters have abstract occurrence count $1^{\#}$. □

Abstract occurrence counting thus does not affect procedure call unfolding (so μ_{def} need not be changed).

Proposition 3. *The resulting μ_{let} is independent of the order in which let-expressions are raised.*

Proof. Follows from Lemma 1: the computation of counts is independent of the current let-annotations. □

6.6 Simplified abstract occurrence counting analysis

The simplified abstract occurrence counting analysis is given in Fig. 6. We formalize the raising of let-annotations by defining a function $raise_{oc}$. It raises let-expressions until all dynamic (binding time value D) actual parameter expressions

```
$raise_{oc}$ : Definition$^+$ $\rightarrow$ $LetMap$ $\rightarrow$ $LetMap$
$raise_{oc}[\![$PD+$]\!]\mu_{let}$ =
  $fix(\lambda \mu'_{let}\ .$
    $\mu_{let} \sqcup (\exists$ an expression $[\![$L (let ((V L$_1$E$_1$)) L$_2$E$_2$)$]\!]$ in $[\![$PD+$]\!]$ :
            $\mu'_{let}(\mathcal{L}[\![L]\!]) = Unfold \wedge \rho_{bt}(\mathcal{V}[\![V]\!]) = D \wedge oc'[\![L_2E_2]\!]\mathcal{V}[\![V]\!] \neq 1^{\#}$
        $\rightarrow$ $upd\ \mathcal{L}[\![L]\!]\ Resid\ \mu'_{let}$
        $[\!]\ \mu'_{let}))$

$oc'$ : LabeledExpression $\rightarrow$ $Variable$ $\rightarrow$ $AbstractCount$
$oc'[\![$L E$]\!]v$ =
  $\mu_{bt}(\mathcal{L}[\![L]\!]) = S \rightarrow 0^{\#}$
  $[\!]$ case $[\![E]\!]$ of
      $[\![C]\!] : 0^{\#}$
      $[\![V]\!] : \mathcal{V}[\![V]\!] = v \rightarrow 1^{\#}\ [\!]\ 0^{\#}$
      $[\![$(if L$_1$E$_1$ L$_2$E$_2$ L$_3$E$_3$)$]\!]$ : $oc'[\![L_1E_1]\!]v +^{\#} (oc'[\![L_2E_2]\!]v \sqcup oc'[\![L_3E_3]\!]v)$
      $[\![$(let ((V L$_1$E$_1$)) L$_2$E$_2$)$]\!]$ : $oc'[\![L_1E_1]\!]v +^{\#} oc'[\![L_2E_2]\!]v$
      $[\![$(begin L$_0$E$_0$ L$_1$E$_1$ ... L$_n$E$_n$)$]\!]$ : $oc'[\![L_0E_0]\!]v +^{\#} oc'[\![L_1E_1]\!]v +^{\#} \ldots +^{\#} oc'[\![L_nE_n]\!]v$
      $[\![$(O L$_1$E$_1$ ... L$_n$E$_n$)$]\!]$ : $oc'[\![L_1E_1]\!]v +^{\#} \ldots +^{\#} oc'[\![L_nE_n]\!]v$
      $[\![$(P L$_1$E$_1$ ... L$_n$E$_n$)$]\!]$ : $oc'[\![L_1E_1]\!]v +^{\#} \ldots +^{\#} oc'[\![L_nE_n]\!]v$
  end
```

Fig. 6. Simplified abstract occurrence counting analysis.

of unfoldable let-expressions have the abstract occurrence count $1^{\#}$. We note that due to the various simplifications, the *OC* function has vanished.

7. A larger example: specializing an MP-interpreter

This section contains an example of compilation by specializing an interpreter. Interpreters for the toy language "MP" (introduced in [37]) have typically been used to test self-applicable partial evaluators [11, 30, 36]. MP is a small imperative untyped "while" language with Lisp data structures, assignments, conditionals, and loops. The abstract syntax of MP is given in Fig. 7.

There are two kinds of variables, declared by pars and vars. The "pars" are input parameters, the "vars" ordinary variables. The semantics is the straightforward one; notice that the empty list ( ) is identified with the boolean value *false*. The result of an execution is taken to be the entire store.

Figure 8 gives an example of an MP-program (coming from [37]). The program computes x to the yth symbolically; the numbers x, y, and the result out are

```
P ∈ Program,  B ∈ Block,  C ∈ Command,
E ∈ Expr,  V ∈ Variable,  Cst ∈ Constant

P  ::=  (program (pars V1*) (vars V2*) B)
B  ::=  (C*)
C  ::=  (:= V E) | (if E B1 B2) | (while E B)
E  ::=  Cst | V | (cons E1 E2) | (equal E1 E2) |
        (car E) | (cdr E) | (atom E)
```

Fig. 7. Abstract syntax of MP.

```
(program (pars x y) (vars out next kn)
  ((:= kn y)
   (while kn
     ((:= next (cons x next))
      (:= kn (cdr kn))))
   (:= out (cons next out))
   (while next
     ((if (cdr (car next))
        ((:= next (cons (cdr (car next)) (cdr next)))    then ...
         (while kn
           ((:= next (cons x next))
            (:= kn (cdr kn))))
         (:= out (cons next out)))
        ((:= next (cdr next))                             else ...
         (:= kn (cons '1 kn))))))))
```

Fig. 8. The MP-program power-MP.

represented as lists. It is not important here how the program actually works, it simply serves as an example.

7.1. MP-interpreter text

The interpreter uses an environment (env) and a store. An environment binds variables to locations, the store binds locations to values. Figure 9 gives the interpreter text. The interpreter uses a number of primitive operators, for instance for processing abstract syntax (for example P→V2*, isAssignment?, and C-Assignment→V) and environments. These operators are defined in the file "MP-int.adt".

The interesting point with this version of the MP-interpreter is the absence of an explicit store variable: the store is handled by primitive operations that only have

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

(define (run P value*)          ; Program × Value* → Undef
  (let* ((V2* (P->V2* P))
         (env (init-environment (P->V1* P) V2*)))
    (init-store! value* (length V2*))
    (evalBlock (P->B P) env)))

(define (evalBlock B env)       ; Block × Env → Undef
  (if (emptyBlock? B)
      "Finished block"
      (evalCommands (headBlock B) (tailBlock B) env)))

(define (evalCommands C B env)  ; Command × Block × Env → Undef
  (if (emptyBlock? B)
      (evalCommand C env)
      (begin (evalCommand C env)
             (evalCommands (headBlock B) (tailBlock B) env))))

(define (evalCommand C env)     ; Command × Env → Undef
  (cond
   ((isAssignment? C)
    (update-store! (lookup-env (C-Assignment->V C) env)
                   (evalExpression (C-Assignment->E C) env)))
   ((isConditional? C)
    (if (is-true? (evalExpression (C-Conditional->E C) env))
        (evalBlock (C-Conditional->B1 C) env)
        (evalBlock (C-Conditional->B2 C) env)))
   ((isWhile? C)
    (if (is-true? (evalExpression (C-While->E C) env))
        (begin (evalBlock (C-While->B C) env)
               (evalCommand C env))
        "Finished loop"))
   (else
    "Error - unknown command")))
```

Fig. 9. MP-interpreter.

```
(define (evalExpression E env)    ; Expr × Env → Value
  (cond
   ((isConstant? E)
    (constant-value E))
   ((isVariable? E)
    (lookup-store (lookup-env (E->V E) env)))
                 ; where lookup-env: Variable × Env → Value
   ((isPrim? E)
    (let ((op (E->operator E)))
      (cond
       ((is-cons? op)
        (cons (evalExpression (E->E1 E) env)
              (evalExpression (E->E2 E) env)))
       ((is-equal? op)
        (equal? (evalExpression (E->E1 E) env)
                (evalExpression (E->E2 E) env)))
       ((is-car? op)
        (car (evalExpression (E->E E) env)))
       ((is-cdr? op)
        (cdr (evalExpression (E->E E) env)))
       ((is-atom? op)
        (atom? (evalExpression (E->E E) env)))
       (else
        "Unknown operator"))))
   (else
    "Unknown expression form")))
```

Fig. 9 (continued).

locations (and values) as parameters, not the store itself. The store is implemented as a *global* variable which is updated destructively, and the store operators (defined in the file "`MP-int.adt`") are hence opaque (see Fig. 10). As can be seen from these definitions, the store is represented as a list, but this could be changed to any other representation; using a vector (array) is an obvious choice of a more efficient

```
(defprim-opaque (init-store! input-V1* length-V2*)
  (set! store
        (append
         input-V1*
         ((rec f (lambda (n) (if (equal? n 0) '() (cons '() (f (sub1 n))))))
          length-V2*))))

(defprim-opaque (update-store! location value)
  (set-car! (list-tail store location) value))

(defprim-opaque (lookup-store location)
  (list-ref store location))
```

Fig. 10. MP-interpreter, store operators.

implementation. We use a list to get a more faithful performance comparison with Mix (Section 8).

Most importantly, the store is global. Notice that the interpreter in case of successful evaluation always returns some dummy (or even undefined) value such as the string "`Finished loop`". The point is that the global variable `store` has been updated, so after the execution `store` contains the final values of the variables.

The above interpreter has been written with a globalized store from the beginning. However, globalizable variables can be detected in purely applicative programs. Schmidt has described a method for detecting such variables in denotational semantics definitions [35], and Sestoft has developed techniques for replacing function parameters by global variables [39]. One could imagine that the above interpreter had been generated automatically (or at least semi-automatically) from a purely applicative program.

One may note that not only the store, but also the environment could be globalized: after initialization, the environment never changes and thus it is definitely single-threaded and globalizable. There are, however, two good reasons not to globalize the environment. Firstly, globalizing environments is not possible in general: if the MP-language had been extended with local Algol-like variable declarations, there would be several active environments around at the same time. The environment would thus not be single-threaded and could not be globalized. A second reason for not globalizing the environment is related to Similix. Globalizing would make environment processing dynamic rather than static: all operations on global variables are treated as dynamic (cf. Section 3.9).

Finally, we note that no generalization point is needed in the MP-interpreter (cf. Section 2.2). This is usually the case in interpretive specifications of programming languages.

7.2. Specializing the MP-interpreter

Let us now use Similix to specialize the MP-interpreter with respect to the MP-program power-MP from above. This yields the Scheme target program given in Fig. 11. The structure of the target program is quite close to assembler code, although the code is not "flattened" (nested `begin` expressions have been flattened automatically by the postprocessor, but other nested expressions still exist). Notice that variable offsets have been computed and that *there are no parameters to the residual procedures.* There were only static parameters to `eval-command` in the source program, and therefore there are no parameters in the residual code. The residual procedure calls correspond closely to assembler instructions of the kind "jump subroutine".

Also notice that the two small `while`-loops both have been compiled into the same procedure, `eval-command-1`. This is of course possible since both while loops perform the same operations. The partial evaluator detects this because both loops are *textually* identical. They therefore correspond to identical static values for the parameter `C` to `eval-command`.

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

(define (run-0 value*_0)
  (init-store! value*_0 3)
  (update-store! 4 (lookup-store 1))
  (evalcommand-1)
  (update-store! 2 (cons (lookup-store 3) (lookup-store 2)))
  (evalcommand-2))

(define (evalcommand-2)
  (if (is-true? (lookup-store 3))
      (begin
        (if (is-true? (cdr (car (lookup-store 3))))
            (begin
              (update-store! 3 (cons (cdr (car (lookup-store 3)))
                                     (cdr (lookup-store 3))))
              (evalcommand-1)
              (update-store! 2 (cons (lookup-store 3) (lookup-store 2))))
            (begin (update-store! 3 (cdr (lookup-store 3)))
                   (update-store! 4 (cons 1 (lookup-store 4)))))
        (evalcommand-2))
      "Finished loop"))

(define (evalcommand-1)
  (if (is-true? (lookup-store 4))
      (begin (update-store! 3 (cons (lookup-store 0) (lookup-store 3)))
             (update-store! 4 (cdr (lookup-store 4)))
             (evalcommand-1))
      "Finished loop"))
```

Fig. 11. Compiled power-MP program.

7.3. *Generating an MP-compiler*

Similix generates an MP-compiler from the interpreter by self-application (cf. Section 1.1). Using the generated compiler, target programs are generated significantly faster than by specializing the interpreter (see the benchmarks in the Section 8). The compiler text is too large to show here. The interested reader can find fragments of automatically generated compilers in [6, 7].

8. Performance

Similix has been implemented in Scheme and self-applied successfully. Because source and residual programs follow the same syntax (our particular subset of Scheme), they can both be run directly in Scheme and specialized further. We have mainly used Similix to generate compilers from interpreters and to specialize pattern matching algorithms. Along the lines of earlier work in self-applicable partial evaluation, we reproduce benchmarks addressing the MP language.

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*. Figure 12 shows the speedups achieved by partial evaluation. It compares (1) running the MP-interpreter on the power-MP source program and running the power-MP target program, (2) specializing the MP-interpreter and running the MP-compiler, and (3, 4) specializing *mix* and using *cogen*.

job	time/s	speedup
$output = int(source, data)$	2.0	9.2
$output = target(data)$	0.2	
$target = mix(int^{ann}, source)$	0.4	4.8
$target = comp(source)$	0.1	
$comp = mix(mix^{ann}, int^{ann})$	3.4	4.4
$comp = cogen(int^{ann})$	0.8	
$cogen = mix(mix^{ann}, mix^{ann})$	18.4	3.9
$cogen = cogen(mix^{ann})$	4.7	

Fig. 12. Similix performance, MP-interpreter example.

Preprocessed programs are superscripted with *ann*. The first column identifies the job, and the run time figures are given in the second column. The figures are given in CPU seconds with one decimal, and they are for an implementation in Chez Scheme version 2.0.3 on a Sun 3/260. The figures exclude the time used for garbage collection (in the worst case 40% additional time, typically much less), but include time for postprocessing (post-unfolding). The third column shows the speedup ratios. More run time decimals than the ones given have been used in the computation of the ratios. The run time figures and ratios have the usual uncertainty connected to CPU measures.

Preprocessing *int* takes 0.7 seconds, and preprocessing *mix* takes 6.4 seconds. The size of the MP-interpreter is around 2 K, and that of the MP-compiler around 8 K. This gives an expansion factor 4. The size of *mix* is around 10 K, of *cogen* around 40 K, also giving an expansion factor of 4.

The figures compare very well with [24], to our knowledge the only other fully automatic partial evaluator (with automatic call unfolding) for a recursive equation language. We get smaller and faster programs, and better speedup ratios. One reason is that besides providing a stronger language, our use of abstract data type operators allows more conciseness and prevents the specialization of data structure processing. The figures are also comparable to the ones given in [11].

9. Related work

The book by Bjørner, Ershov and Jones [1] contains a thorough bibliography about other works involving partial evaluation.

9.1. Mix

Mix [23] was the first actual autoprojector. It processed programs expressed as collections of Lisp-type first-order recursive equations with a fixed set of primitive operators. Mix showed the need for binding time analysis in self-application partial evaluation, and many problems were identified while developing it: duplication, termination, and so on. An automatic version of Mix has been developed later [24].

9.2. Call duplication

The problem of *call duplication* is described and solved in [38]. Whereas computation duplication concerns duplicating any nonconstant residual expression, call duplication only concerns a subset of these, namely those containing function (procedure) calls. An additional abstract analysis operating on source programs. Sestoft's *call abstract interpretation*, is needed to detect such expressions. The analysis may need to be repeated during preprocessing. Sestoft's *duplication risk analysis* resembles our abstract occurrence counting analysis, but it is used differently: the language used there has no let-expressions, so duplication is avoided by raising the annotation of the surrounding call (that causes the duplication) into *Resid.*

9.3. Partially static structures

Mogensen developed an autoprojector treating partially static structures (using structured binding time values); to some degree, let-expressions were used to separate call/code duplication issues from call unfolding issues [30].

9.4. Arity reducing and arity raising

Moscow-Mix [34] is an autoprojector for RL ("Refal-Lisp") programs. It presents partial evaluation essentially as a two-phase process: arity reducing (specialization) and arity raising. Arity raising changes the functionalities of residual procedures from taking a list of *n* values to taking *n* arguments. Arity raising is referred to as *variable splitting* in the Copenhagen Mix work and *retyping* in [31].

9.5. Schism

Schism [10], an autoprojector for first-order Scheme programs, was the first to offer an open-ended set of primitive operators. The system uses hand-written filters to specify whether a procedure call should be unfolded or specialized as well as how arguments should be propagated if the call is specialized. As described in [11], Schism uses *polyvariant binding time analysis* and it also treats partially static structures.

In contrast, Similix's binding time analysis is *monovariant*: it only generates one binding time annotated version of each source procedure. If a procedure is called with different binding time patterns, the least upper bound is taken. This implies a possible loss of static information at program specialization time. In Consel's system, calls with different binding time patterns cause the binding time analyser to generate several annotated versions, one for each binding time pattern. This is similar to polyvariant partial evaluation, but the polyvariancy occurs already at binding time analysis time. In addition to this, Schism uses polyvariant specialization at partial evaluation time; the residual procedures are thus specialized versions of (binding time) specialized versions of the source procedures.

9.6. Compilation of binding times

In Schism, the interpretation of binding times is lifted away from the self-applicable specialization kernel, which allows to factor completely static and completely dynamic expressions out of the actual specialization [11, 14].

9.7. Synthesis

Similix has fulfilled and even gone beyond our initial expectations, in its underlying principles as well as in its actual realization: call unfolding is fully automatic (no user-added call unfolding annotations); it offers an open-ended abstraction of data structures compatible with the binding time analysis; it provides a sound interface with global variables (such as i/o); it guarantees not to duplicate computations in residual programs; it preserves termination properties; it specializes different program points than just user defined procedures; and it automatically maintains the consistency between different overlapping sets of user defined primitive operators.

As a direct consequence of Similix's open-ended design, arity raising (variable splitting) need no longer be particular for the operators cons, car, and cdr: it can be parameterized with respect to the user defined abstract data type operators. A prototype arity raiser based on this idea has been developed [29].

9.8. Higher-order partial evaluation

Similix has been extended to handle a *higher-order* subset of Scheme. This extension is described in [4] for a side-effect-free language. For a full description, also covering side effects on global variables, see [6]. This higher-order extension of Similix does provide arity raising through the higher-order constructs [4]. Other higher-order partial evaluators include Lambda-Mix [22] and a new version of Schism [12]. These systems are all based on monovariant binding time analyses and they offer various degrees of polyvariancy and automatism.

10. Conclusion and open problems

We have addressed and solved the partial evaluation problems of automating call unfolding, having an open-ended set of operators, and processing global variables updated by side effects. The problems of computation duplication and termination of residual programs have been addressed and solved: residual programs never duplicate computations of the source program; residual programs do not terminate more often than source programs.

We have presented a new method for automatic call unfolding which is simpler, faster, and sometimes more effective than existing methods: it neither requires recursion analysis of the source program, nor call graph analysis of the residual program.

To avoid computation duplication and preserve termination properties, we introduced an abstract interpretation of the source program, performed during preprocessing: abstract occurrence counting analysis.

Two important open problems remain: Similix's binding time analysis in monovariant (Section 9), and generalization points need to be inserted by hand in source programs (Section 2.2).

10.1. Applications

Applying partial evaluation in an active research area today. The applications include, among others, compiling pattern matching, [16, 25], compiling laziness [5], and compiling Algol-like programs [15].

Acknowledgement

This work has been accomplished in Neil D. Jones's Copenhagen Mix group, through many discussions with each member. We are most grateful to Torben Æ. Mogensen for stimulating discussions and for ideas and suggestions on inserting let-expressions to avoid duplication. We have benefited from insightful discussions with Charles Consel, especially concerning open-endedness in partial evaluation. Thanks are also due to Chris Hankin and the referees for many useful comments.

References

[1] D. Bjørner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation*, IFIP TC2 (North-Holland, Amsterdam, 1988); *Workshop Proceedings*, G. Avernæs, Denmark (1987).

[2] A. Bondorf, Towards a self-applicable partial evaluator for term rewriting systems, in: D. Bjørner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 27-50.

[3] A. Bondorf, A self-applicable partial evaluator for term rewriting systems, in: J. Diaz and F. Orejas, eds., *TAPSOFT'89, Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain*, Lecture Notes in Computer Science **352** (Springer, Berlin, 1989) 81-96.

[4] A. Bondorf, Automatic autoprojection of higher order recursive equations, in: N.D. Jones, ed., *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science **432** (Springer, Berlin, 1990) 70-87; also: *Sci. Comput. Programming* **17** (1991).

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

[6] A. Bondorf, Self-applicable partial evaluation (revised version), Ph.D. Thesis DIKU Report 90-17, DIKU, University of Copenhagen, Denmark (1990).

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

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

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

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

[11] C. Consel, Analyse de programmes, evaluation partielle et génération de compilateurs, Ph.D. Thesis, LITP, University of Paris 6 (1989).

[12] C. Consel, Binding time analysis for higher order untyped functional languages, in: *1990 ACM Conference on Lisp and Functional Languages*, Nice, France (1990) 264-272.

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

[14] C. Consel and O. Danvy, From interpreting to compiling binding times, in: N.D. Jones, ed., *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science **432** (Springer, Berlin, 1990) 88-105.

[15] C. Consel and O. Danvy, Static and dynamic semantics processing, in: *Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Orlando, FL (1991).

[16] O. Danvy, Semantics-directed compilation of nonlinear patterns, *Inform. Process. Lett.* 37(6) (1991) 315-322.

[17] A.P. Ershov, Mixed computation: potential applications and problems for study, *Theoret. Comput. Sci.* **18** (1982) 41-67.

[18] C.K. Gomard and N.D. Jones, Compiler generation by partial evaluation: a case study, in: *Proceedings Twelfth IFIP World Computer Congress* (1989).

[19] P. Hudak and J. Young, A collecting interpretation of expressions (without powerdomains), in: *Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Diego, CA (1988) 107-118.

[20] J. Hughes, Abstract interpretation of first-order polymorphically typed languages, in: C. Hall, J. Hughes and J.T. O'Donnell, eds., *1988 Glasgow Workshop on Functional Programming*, Research Report 89/R4, Computing Science Department, Glasgow University, Scotland (1989) 68-86.

[21] N.D. Jones, Automatic program specialization: a re-examination from basic principles, in: D. Bjørner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 225-282.

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

[23] 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.

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

[25] J. Jørgensen, Generating a pattern matching compiler by partial evaluation, in: S.L. Peyton Jones, G. Hutton and C. Kehler Holst, eds., *Functional Programming, Glasgow 1990. Workshops in Computing* (Springer, Berlin, 1990) 177-195.

[26] E.E. Kohlbecker, Syntactic extensions in the programming language Lisp, Ph.D. Thesis, Indiana University, Bloomington, IN (1986).

[27] J. Launchbury, Projection factorisations in partial evaluation, Ph.D. Thesis, Department of Computing, University of Glasgow, Scotland (1989).

[28] K. Malmkjær and O. Danvy, Preprocessing by specialization. Tech. Report, Kansas State University, Manhattan, KS (1991).

[29] M. Marquard and B. Steensgaard-Madsen, Parameter splitting in a higher order functional language, Student Report 90-7-1, DIKU, University of Copenhagen, Denmark (1990).

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

[31] T.Æ. Mogensen, Binding time aspects of partial evaluation, Ph.D. Thesis, DIKU, University of Copenhagen, Denmark (1989).

[32] P.D. Mosses and D.A. Watt, The use of action semantics, in: *IFIP TC2 Working Conference on Formal Descriptions of Programming Concepts* III (North-Holland, Amsterdam, 1986).

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

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

[35] D.A. Schmidt, Detecting global variables in denotational specifications, *Trans. Programming Languages Syst.* 7 (2) (1985) 299-310.

[36] P. Sestoft, The structure of a self-applicable partial evaluator, in: H. Ganzinger and N.D. Jones, eds., *Programs as Data Objects, Copenhagen, Denmark*, Lecture Notes in Computer Science **217** (Springer, Berlin, 1985) 236-256.

[37] P. Sestoft, The structure of a self-applicable partial evaluator, Tech. Report 85-11, DIKU, University of Copenhagen, Denmark (1985).

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

[39] P. Sestoft, Replacing function parameters by global variables, Master's Thesis, Student Report 88-7-2, DIKU, University of Copenhagen, Denmark (1988).

[40] P. Sestoft, Replacing function parameters by global variables, in: *Fourth Interational Conference on Functional Programming and Computer Architecture, London*, (ACM Press and Addison-Wesley, Reading, MA, 1989) 39-53.

[41] V.F. Turchin, The concept of a supercompiler, *Trans. Programming Languages Syst.* **8** (1986) 292-325.