Similix 5.0 Manual

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

May 17, 1993

Abstract

Similix is an autoprojector (self-applicable partial evaluator) for a large higher-order subset of the strict functional language Scheme. Similix treats source programs that use a limited class of side-effects, for instance input/output operations. Similix handles partially static data structures.

Similix is automatic: in general, no user annotations (such as unfolding information) are required; user assistance may in some cases be required to avoid looping, however. Similix gives certain guarantees concerning the residual programs it generates: computations are never discarded (partial evaluation thus preserves termination properties) and never duplicated.

Similix is well-suited for partially evaluating for instance interpreters that use environments represented as functions and interpreters written in continuation passing style. Since Similix is self-applicable, stand-alone compilers can be generated from interpreters.

Similix is highly portable. It conforms to the IEEE and R4RS Scheme standards, but it also runs under R3RS Scheme.

Similix 5.0 is based on the former Similix 4.0 (by Anders Bondorf and Olivier Danvy) [Bon91c]. A part of Similix 5.0 has been written jointly by Anders Bondorf and Jesper Jørgensen.

Relevant Similix references: [BD91, Bon91a, Bon90a, Bon91b, Bon92, BJ93a, BJ93b].

This manual summarizes some often used *binding-time improvements* (Section 7). These are needed in order to obtain good results of partial evaluation. The section is partly Similix-specific, but parts of it are of more general interest.

Please note Similix is an experimental system under development which may contain bugs and errors. You are encouraged to mail us about bugs, comments, suggestions and the like, but we cannot promise to give detailed answers to every communication.

Please direct any Similix communication to the author, preferably by e-mail.

Main differences between Similix 4.0 and Similix 5.0

New Similix 5.0 features

- Similix 5.0 is highly portable.
- A larger Scheme subset is handled, in particular internal definitions, `letrec`, and named (recursive) `let`.
- Partially static data structures are now available.
- User-defined constructors are now available (extension to Scheme).
- Pattern-matching facilities are now available (extension to Scheme).
- The preprocessing phase is much faster (e.g. binding-time analysis).
- A trace facility for tracking infinite specialization is now available.
- For binding-time debugging: the show-facility for inspecting preprocessed programs has been improved.
- User-control of specialization/memoization point insertion is now possible; this enables dynamic choice of static values.
- The file `scheme.adt` is now always automatically loaded, so user-programs need no longer contain the corresponding `loadt`-expression.

Similix 4.0 features not available in Similix 5.0

- Macros (`extend-syntax`) are no longer supported (because of portability problems).
- No binding-time debugger is available in Similix 5.0.

Contents

List of Figures

Acknowledgements

Similix 5.0 is based on Similix 4.0 which was joint work between Olivier Danvy and the author of this manual. The flow, binding-time, and evaluation-order dependency analyses of Similix 5.0 are joint work with Jesper Jørgensen.

A number of people have contributed to the system in many ways. Thanks are due to Lars Ole Andersen, Mikhail Bulyonkov, Charles Consel, Olivier Danvy, Hans Dybkjær, Harald Ganzinger, Robert Glück, Carsten K. Gomard, Chris Hankin, Fritz Henglein, Kristoffer Rose, Carsten Kehler Holst, Kristian Damm Jensen, Neil D. Jones, Jesper Jørgensen, John Launchbury, Karoline Malmkjær, Torben Æ. Mogensen, Christian Mossin, Peter Sestoft, and Jörg Süggel — and to those whom I may have undeliberately forgotten.

Overview of the paper

Section 1 contains a short general introduction to partial evaluation.

Section 2 introduces Similix by a small example session.

Section 3 describes the language treated by Similix.

Section 4 contains larger examples of Similix applications.

Section 5 gives a brief overview of the components of the Similix system.

Section 6 describes how to inspect annotated programs. Manual inspection of annotated programs is often necessary to obtain good results of partial evaluation.

Section 7 summarizes some often used *binding-time improvements*. These are needed in order to obtain good results of partial evaluation. The section is partly Similix-specific, but parts of it are of more general interest. To get full benefit of the section, you should also read at least Section 5 and Section 6.

Section 8 gives a systematic overview of the facilities available in Similix.

Readers familiar with Similix 4.0 are encouraged to read at least the sections 3, 6, and 7; these sections have changed significantly. Notice that the examples in Section 4.2 and Section 4.3 are new. The binding-time analysis domain has changed (Section 5.2.2). Finally, Section 8 contains some new and some improved facilities.

1 Short Introduction to Partial Evaluation

Partial evaluation [JGS93] 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:

$$p(s,d) = p_s(d) \ \textit{where} \ p_s = mix(p,s)$$

For simplicity, we have not distinguished programs from the functions they compute. For instance, p denotes a function in $p(s,d)$, but a program in $mix(p,s)$. Input s is called *static* and input d is called *dynamic*.

The main point in partial evaluation is efficiency: running the residual program p_s can be much faster than running the source program p. Instead of running $p(s,d)$, it may therefore be worthwhile first to generate p_s and then apply it to d. The partial evaluator knows p and s and is therefore able to perform those of p's computations that depend only 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 *auto-projection* [Ers82]. Let us insert mix for p, p for s, and s for d in the equation defining a residual program:

$$mix(p,s) = mix_p(s) \ \textit{where} \ mix_p = mix(mix,p)$$

Specializing p with respect to s may thus be done by running $mix_p(s)$ instead of the (potentially) slower $mix(p,s)$. Notice that mix_p is a *curried* version of p: program mix_p is a program which, when applied to s, generates a new program p_s which can then be applied to d. Program p thus takes its inputs s and d at the same time, but program mix_p takes them one at a time.

Program mix_p is often called a *generating extension* of p [Ers78]. The generating extension is a "specialized specializer": it is able to generate specialized versions of a particular program p whereas the general specializer mix can specialize any program. The advantage of generating a specialized specializer is efficiency: it is potentially faster to run a specialized specializer than to run the general one.

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

$$mix(mix,p) = mix_{mix}(p) \ \textit{where} \ mix_{mix} = mix(mix,mix)$$

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* [BEJ88]. Suppose that int takes two inputs, a source program written in the language specified by int and some data input to this source program:

$$output = int(source, data)$$

Then int_{source} is a *target* program $target$, written in the language that the residual programs generated by mix are written in. Program $target$ fulfills

$$output = int(source, data) = int_{source}(data) = target(data)$$

Self-application generates $comp = mix_{int}$. This program is a *compiler* since

$$comp(source) = int_{source} = target$$

Program mix_{mix} plays the role of a *compiler generator* $cogen$:

$$cogen(int) = comp$$

Here $cogen = mix_{mix}$. The three Futamura projections are:

1: $target = mix(int, source)$

2: $comp = mix(mix, int)$

3: $cogen = mix(mix, mix)$

We finally notice that mix_{mix} has an interesting feature — it is self-generating:

$$mix_{mix} = mix_{mix}(mix)$$

The first successfully implemented autoprojector was *Mix* [JSS85]. The language treated by Mix was a subset of statically scoped first-order pure Lisp, and Mix was able to generate compilers from interpreters written in this language. The experiment showed that autoprojection was possible in practice; an automatic version of Mix was developed later [JSS89].

1.2 Partial evaluation, operationally

Partial evaluation works by propagating the static input and performing statically reducible operations. As an example, a conditional expression `(if` E_1 E_2 E_3`)` can be reduced if 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 dynamic input, the conditional is not reducible and is therefore left residual: a residual expression `(if` RE_1 RE_2 RE_3`)` is produced. Here RE_i is the residual expression obtained by specializing E_i.

1.3 Preprocessing, annotations and binding-time improvements

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* to the source program [JSS85]. Partial evaluation is then done by specializing the annotated source program rather than the source program itself.

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, and they tell whether operations can be reduced during program specialization.

In addition to this, annotations are very useful for the user of a partial evaluator: by inspecting the annotated source program, the user can predict which operations will be reduced and which will appear in the residual programs generated by the specializer. Section 6 describes how to inspect preprocessed programs in Similix.

When inspecting annotated source programs, you will often experience that some operations are not annotated as you had expected them to be: typically, too much will have been annotated as "not reducible". By systematic rewritings of the source program, it is often possible to improve the annotations to make more operations reducible. Some common such *binding-time improvements* are described in Section 7.

2 Getting Started with Similix

This section contains an introduction to running Similix.

We assume that the Similix system has been installed as described in the `README` file. Position yourself in the `examples` directory and start Scheme. Boring typing work can be avoided by copying expressions to evaluate from the file `getting-started`.

Figure 1 contains the complete session; the numbers to the left are used for reference in the explanation below. We used Aubrey Jaffer's Scheme system "Scm" when running the session.

```
1 > (load "../system/sim-scm.scm")
  ;loading "../system/sim-scm.scm"

  Welcome to Similix 5.0
  Copyright (C) 1993 Anders Bondorf
  Contributions by Olivier Danvy and Jesper Joergensen

  util langext abssyn miscspec runtime front ......
  #<unspecified>
2 > (load "append.sim")
  #<unspecified>
3 > (append1 '(1 2 3) '(4 5 6))
  (1 2 3 4 5 6)
```

```
 4 > (similix 'append1 (list '(1 2 3) '***) "append.sim")
   front-end flow bt sp eod oc rl
   specializing
   ((define (append1-0 l2_0) (cons 1 (cons 2 (cons 3 l2_0)))))
 5 > (load-residual-program)
   ()
 6 > (append1-0 '(4 5 6))
   (1 2 3 4 5 6)
 7 > (define target (residual-program))
   #<unspecified>
 8 > (pp (showpall))
   ((define (_sim-goal l1:s l2:d -> d)
      (append1 l1 l2))
    (define (append1 l1:s l2:d -> d)
      (if (null? l1)
        l2
        (_cons (lift (car l1)) (append1 (cdr l1) l2)))))
   54
 9 > (cogen 'append1 '(static dynamic) "append.sim")
   front-end flow bt sp eod oc rl
   loading compiler generator
   generating compiler
   ()
10 > (comp (list '(1 2 3) '***))
   loading current compiler
   specializing
   ((define (append1-0 l2_0) (cons 1 (cons 2 (cons 3 l2_0)))))
11 > (define new-target (residual-program))
   #<unspecified>
12 > (equal? target new-target)
   #t
13 >
```

Figure 1: Getting started session

The session uses an example program, a program for appending two lists:

```
(define (append1 l1 l2)
  (if (null? l1)
      l2
      (cons (car l1) (append1 (cdr l1) l2))))
```

Figure 2: Program defined in file `append.sim`

The procedure has been named `append1` in order not to conflict with the standard Scheme procedure `append`. Now follows a description of the session.

1: Loading Similix:

The Similix system is loaded by loading the file `sim-scm.scm`. (Its location in general depends on the local installation.)

2–3: The example program *append*, located in `append.sim` and with goal procedure `append1`, is loaded (*2*) and applied to two lists (*3*).

4: Specialization:

The *append* program — with goal procedure `append1` — is now specialized with respect to a static value for its first parameter `l1`; the static value is the list `(1 2 3)`. No value is provided for the second parameter `l2`, so this input will be dynamic; dynamic input is denoted by the symbol `***`.

The residual program is a program that can append the list `(1 2 3)` to an arbitrary list: the residual program is a specialized version of the source program *append*.

Note:

Partial evaluation is done in several steps as indicated by the output

```
front-end flow bt sp eod oc rl
specializing
```

The *append* program is first parsed (`front-ending`). Partial evaluation is then done in two phases, as explained in further detail in Section 5. The first phase, the *preprocessing* phase, consists of several subphases (`flow`, `bt`, `sp`, `eod`, `oc`, and `rl`). The preprocessing generates an *annotated* version of the source program. The second phase (`specializing`) then specializes the annotated program, generating the residual program.

5: The residual program is loaded.

6: The residual program is run on an input list `(4 5 6)`, generating the expected output. The name `append1-0` of the goal procedure of the residual program is determined by the call to `similix` under point 4 (see Section 8.3.1 for details).

7: The residual program is saved in the variable `target` for later use.

8: Inspecting the annotated program:

The annotated program is inspected. Depending on the Scheme system, the pretty-printer must be called explicitly as done here: `(pp ...)`); the return value `54` is generated by the particular pretty-printer used here. Note that Similix automatically inserts an additional goal procedure `_sim-goal`; this is for internal technical reasons. The preprocessing phase has annotated `append1`'s first argument as static (`l1:s`) and its second argument as dynamic (`l2:d`). [This coincides with what we had specified under point 4. Sometimes the parameters of the user given goal procedure get a different annotation (more becomes dynamic) than the initial one provided by the user; this may happen when there are recursive calls to the goal procedure. Notice, however, that the additional goal procedure `_sim-goal` always has exactly the annotation the user specified for the (original) goal procedure.] The return value of `append1` is dynamic (`-> d`).

The annotated program also indicates which operations that will be reduced and which will occur in the residual program. The non-reducible ones are distinguished by a prefix `_` as exemplified by `_cons` above. Thus, the residual program will contain occurrences of this `cons`-operation as we also saw under point 4. The other operations are all reducible.

Notice the `lift`-form: it identifies where static values, computed during specialization, are dumped in residual code. The first argument to occurrences of `cons` in the residual program will thus (always) be a constant. The residual program generated under point 4 exemplifies this: in the expression `(cons 1 (cons 2 (cons 3 l2_0)))`, `1`, `2`, and `3` are all constants.

How to inspect annotated programs is described in detail in Section 6.

9: Generating a generating extension (currying):

We now use the Similix compiler generator to generate a generating extension (curried version) of *append* (cf. Section 1.1). The compiler generator is a general currifier, compiler generation being just one application. The generating extension of *append* is a program which, when applied to a list, generates a specialized version of *append*.

It is specified that the compiler generator should curry the *append* program with goal procedure `append1`. The *binding-time pattern* `(static dynamic)` tells `cogen` which way to curry *append*: here, `l1` will be the "early" parameter and `l2` will be the "late" one. If the binding-time pattern had been specified as, say, `(dynamic static)` instead, the program would have been curried the other way around.

10: Running the generating extension:

The generating extension is now applied to the same input that *append* was specialized with respect to above (line *4*), generating the specialized *append* program that we have already seen. Specializing by `comp` is faster (typically several times) than specializing by `similix`, so it will often be worthwhile first to generate a generating extension by `cogen`.

The procedure for running `cogen`-generated generating extensions is called `comp` because of the way generating extensions resemble compilers: when `cogen` is applied to an *interpreter int*, the resulting generating extension is a compiler mix_{int} (see Section 1.1).

11–12: That the two specialized *append* programs, generated by `comp` and `similix` respectively, are syntactically identical is verified by comparing them for equality.

The reader is strongly recommended to read (at least) Section 8 before experimenting further with Similix.

3 The Language Treated by Similix

Similix treats source programs written in an extension of a subset of Scheme [IEE90, CR91], see Figure 3 and Figure 4. The forms on lines marked with "♠$^{\text{Sec}\ \ldots}$" are thus extensions: these forms are not part of standard Scheme. The extensions are explained in the sections indicated by the superscripts.

Since programs follow the syntax of Scheme, they are directly executable in a Scheme environment. Notice that the Similix system needs to be loaded first if any of the extension forms are used. (Programs using the extension forms can be converted to stand-alone Scheme programs (independent of the Similix system) by the Similix procedure `sim2scheme`, see Section 8.3.5.)

The Similix front-end parser syntactically expands a number of the forms in Figure 3 and Figure 4 into simpler *core* forms. Thus, the programs that are actually partially evaluated are written in this more restricted core syntax. The core language and the expansion into it is described in Section 3.10.

The following standard Scheme syntax forms are *not* handled by Similix (this list may not be exhaustive):

- References to dynamically bound top-level variables; such variables are, however, accessible though user-defined primitive operators (see Section 4.1.1 for an example).
- The forms `(lambda` V E`)` and `(lambda (`V* `.` V`)` E`)`, that is, lambda-expressions with variable arity.
- The form `(set! ...)`. See Section 3.9 for how to simulate `set!` by other operations.
- The form `(case ...)`.
- The forms `(quasiquote ...)`, `` `(...)``, `(unquote ...)`, `,(...)`, `(unquote-splicing ...)`, and `,@(...)`.
- The forms `(define` V E`)` and `(define (`P V* `.` V`)` E`)`, that is, any `define`-form different from the D-form specified in Figure 3.
- The form `(begin (define ...)`$^+$`)`.
- Any `letrec`-form different from the kind specified in Figure 3.
- Any top-level form different from the form TLE specified in Figure 3.

$\Pi \in$ Program ; D $\in$ Definition ; TLE $\in$ TopLevelExpression ;
F $\in$ File ; B $\in$ Body ; E $\in$ Expression ; K $\in$ Constant ; V $\in$ Variable ;
Ofa $\in$ FixedArityPrimopName ; Ova $\in$ VariableArityPrimopName ;
C $\in$ ConstructorName ; S $\in$ SelectorName ; P $\in$ ProcedureName ;
MPat $\in$ CaseMatchPattern, CPat $\in$ CaseConstrPattern, WiC $\in$ WildCard
SE $\in$ SelfEvaluating ; Dat $\in$ Datum ; Bool $\in$ Boolean ; Num $\in$ Number ;
Char $\in$ Character ; Str $\in$ String ; Sym $\in$ Symbol ; Lis $\in$ List ; Vec $\in$ Vector ;

Π	::=	TLE* D TLE*	
TLE	::=	D | (load F)	
		| (loads F)	♠[Sec 3.8] loads-form
		| (loadt F)	♠[Sec 3.4] loadt-form
D	::=	(define (P V*) B)	procedure definition
B	::=	D* E+	body
E	::=	K | V	constant, variable
		| (if E E E)	conditional
		| (if E E)	one-armed conditional
		| (cond (E E*)+)	conditional
		| (cond (E E*)* (else E+))	conditional
		| (and E*) | (or E*)	and, or
		| (let ((V E)*) B)	parallel let
		| (let* ((V E)*) B)	sequential let
		| (let P ((V E)*) B)	named (recursive) let
		| (letrec ((P (lambda (V*) B))*) B)	letrec
		| (begin E+)	sequence
		| Ofa	fixed-arity prim. operator
		| (Ova E*)	variable-arity prim. operation
		| C	♠[Sec 3.6] constructor
		| S	♠[Sec 3.6] selector
		| C?	♠[Sec 3.6] constr. test-predicate
		| P	procedure name
		| (lambda (V*) B)	lambda-expression
		| (E E*)	application
		| (casematch E (MPat E+)*)	♠[Sec 3.8] casematch
		| (caseconstr E (CPat E+)*)	♠[Sec 3.8] caseconstr

Figure 3: Similix source language, part 1

MPat	::=	K | () | (MPat . MPat) | V | WiC	♠Sec 3.8
CPat	::=	(C CPat*) | V | WiC	♠Sec 3.8
WiC	::=	_ | else	♠Sec 3.8
V, P, C, S ::= Sym			
Ofa	::=	...	see Figure 5
Ova	::=	...	see Figure 6
K	::=	SE | (quote Dat) | 'Dat	
SE	::=	Bool | Num | Char | Str	
Dat	::=	SE | Sym | Lis | Vec	
Lis	::=	(Dat*) | (Dat+ . Dat) | 'Dat	
Vec	::=	#(Dat*)	

Figure 4: Similix source language, part 2

3.1 Restrictions on input to programs being partially evaluated

As we have seen in Section 2, a goal procedure must be specified when specializing a program. When specializing, all static input values to the goal procedure must be *first-order, acyclic* values; values constructed by user-defined constructors (cf. Section 3.6) are not allowed as static input.

The dynamic input, which is not specified when specializing, must also be *first-order* when running the residual program; values constructed by user-defined constructors are allowed in the input to the residual program.

The restrictions on static input and on input to residual programs are *not* checked by the system.

3.2 Programs relying on unspecified values

Similix gives no guarantee to preserve the semantics of programs that rely on unspecified values (example: the return value of a one-armed conditional if the test fails).

3.3 Primitive operators

Similix distinguishes (user-defined) *procedures* (P) from *primitive operators* (Ofa, Ova), see Figure 3 and Figure 4. Following Scheme terminology, we use the term "procedure" rather than "function". Primitive operators are also "procedures" in the Scheme sense [CR91], but they differ from user-defined procedures in the way the partial evaluator treats them. Primitive operations are "black box" operations: the specializer either reduces a primitive operation completely or it leaves the primitive operator in the residual program. On the

```
Ofa ::= abs | assoc | boolean?
      | caaaar | caaadr | caaar | caadar
      | caaddr | caadr | caar | cadaar
      | cadadr | cadar | caddar
      | cadddr | caddr | cadr
      | call-with-input-file | call-with-output-file
      | car | cdaaar | cdaadr | cdaar | cdadar | cdaddr
      | cdadr | cdar | cddaar | cddadr | cddar
      | cdddar | cddddr | cdddr | cddr | cdr | ceiling
      | char->integer | char-alphabetic? | char-ci<=?
      | char-ci<? | char-ci=? | char-ci>=? | char-ci>?
      | char-downcase | char-lower-case? char-numeric?
      | char-upcase | char-upper-case? | char-whitespace?
      | char<=? | char<? | char=? | char>=? | char>? | char?
      | close-input-port | close-output-port | complex? | cons
      | current-input-port | current-output-port | eof-object?
      | equal? | even? | exact? | floor | gcd
      | inexact? | input-port? | integer->char | integer?
      | lcm | length | list->string
      | list->vector | list-ref | list? | member
      | modulo | negative? | not | null? | number? | odd?
      | open-input-file | open-output-file | output-port?
      | pair? | positive? | procedure? | quotient | rational?
      | real? | remainder | reverse | round | string->list
      | string->symbol | string-ci<=? | string-ci<?
      | string-ci=? | string-ci>=? | string-ci>?
      | string-length | string-ref | string<=? | string<?
      | string=? | string>=? | string>? | string?
      | substring | symbol->string | symbol? | truncate
      | vector->list | vector-length
      | vector-ref | vector? | zero?
      | _sim-memoize                                   ♠Sec 7.2.3
      | user-defined primitive operator                Sec 3.4
```

Figure 5: Primitive operators, fixed arity

```
Ova ::= *  | +  | -  | /  | <  | <=  | =  | >  | >=
      | append  | display  | list  | make-string
      | make-vector  | max  | min  | newline
      | number->string  | peek-char  | read  | read-char
      | string  | string->number  | string-append
      | vector  | write  | write-char
      | _sim-error                                ♠Sec 3.3
      | user-defined primitive operator           Sec 3.4
```

Figure 6: Primitive operators, variable arity

contrary, the partial evaluator knows the code of a user-defined procedure; it can therefore elaborate a procedure call even if some arguments are dynamic at partial evaluation time.

Primitive operators are divided into two categories: those with fixed arity and those with variable arity. The only difference between these is that a fixed-arity primitive is an allowed expression form in itself whereas a variable-arity primitive must be in apply-position, cf. Figure 3; also, a fixed-arity primitive is arity checked by the Similix front-end parser when standing in apply-position.

A number of primitive operators are available, see Figure 5 and Figure 6. Except for the primitive operators `_sim-memoize` and `_sim-error`, these primitives constitute a subset of the standard Scheme "essential" procedures [CR91]. Primitive operators may also be user-defined as described in Section 3.4. The primitive `_sim-error` is used for aborting execution while printing a formatted error message; see Sections 4.1, 4.2, 4.3 for examples.

The only essential procedures from [CR91] not treated are: `eqv?`, `eq?`, `memq`, `memv`, `assq`, `assv`, `set-car!`, `set-cdr!`, `string-set!`, `vector-set!`, `apply`, `map`, `for-each`, `call-with-current-continuation`, and `load`. (However, see Section 3.5.)

No non-essential procedures from [CR91] are treated. However, such forms (except `string-fill!` and `vector-fill!`) can be defined by the user by following the guidelines in Section 3.4 where defining the non-essential `sqrt` as a primitive operator is exemplified.

3.4 User-defined primitive operators

User-defined primitive operators are useful for different purposes:

- For introducing side-effecting operations and operations that reference global variables.
- For introducing aborting operations.
- For controlling termination of specialization.

Finally, introducing primitives may speed up partial evaluation (see Section 7.3.4).

User-defined primitive operators are defined in separate files according to the syntax given in Figure 7. The `defconstr`-form for defining constructors is described in Section 3.6.

OCDs ∈ OpConstrDefinitions, OCD ∈ OpConstrDefinition,
SchE ∈ SchemeExpression, SchV ∈ SchemeVariable
Key ∈ KeyForm, Ari ∈ Arity,
Ss ∈ SelectorForm, S ∈ SelectorName

```
OCDs ::= OCD*
OCD  ::= (Key (Ofa V*) SchE)         primop. def. form 1f
       | (Key (Ova . V) SchE)        primop. def. form 1v
       | (Key Ari Ofa SchV)          primop. def. form 2f
       | (Key Ova SchV)              primop. def. form 2v
       | (defconstr (C Ss*)+)        constructor definition
       | (loadt F)
Key  ::= defprim-transparent | defprim
       | defprim-tin
       | defprim-dynamic
       | defprim-opaque
       | defprim-abort
       | defprim-abort-eoi
Ari  ::= 0 | 1 | 2 | ...
Ss   ::= S | *
```

Figure 7: User-defined primitive operators and constructors

A program using primitive operators in *file-name*`.adt` must contain the expression `(loadt` *file-name*`.adt)`, cf. Figure 3. A program may use primitive operators from many files; for each file, the program must contain an appropriate `loadt`-expression.

Notice from Figure 7 that a file containing primitive operator (and constructor) definitions may itself contain `loadt`-expressions. Such `loadt`-expressions make modularization easier; these `loadt`-expressions are syntactically (textually) expanded at load time, so the effect of using such `loadt`-expressions is the same as if the contents of referenced file had been textually copied into the referencing file.

Primitive operators defined within a file may call each other (also mutually recursively). Within an expression SchE there may thus be references to top-level defined names and to other primitives defined within the same file (or within files referenced by `loadt`-expressions as described in the previous paragraph).

Here are some examples of primitive operator definitions:

```
(defprim-transparent (my-op x y) (cons x (cons x y)))       1f
(defprim-transparent 1 my-car car)                          2f
(defprim-tin 1 sqrt sqrt)                                   2f
(defprim-opaque 1 read read)                                2f
(defprim list list)                                         2v
(defprim (my-list . x) x)                                   1v
(defprim-abort run-time-error _sim-error)                   2v
(defprim-abort-eoi syntax-error _sim-error)                 2v
```

Some other examples can be found in the file `scheme.adt` in the `system` directory. In fact, this file, which is always loaded automatically, defines all the primitives in Figure 5 and Figure 6.

When a program p using primitive operators is run, the primitive operator definitions correspond to ordinary Scheme definitions. The above definitions thus correspond to the definitions

```
(define (my-op x y) (cons x (cons x y)))
(define my-car car)
(define sqrt sqrt)
(define read read)
(define list list)
(define (my-list . x) x)
(define run-time-error _sim-error)
(define syntax-error _sim-error)
```

When the program p is partially evaluated, however, the additional information in the `defprim`-forms is used as described shortly.
The form SchE can be (almost) any Scheme expression (Section 3.5 describes some restrictions) and is thus not restricted to the expression subset allowed for procedure definitions (Figure 3–4). Similix never looks "inside" SchE-expressions: as mentioned above, Similix considers a primitive operation to be atomic. The SchV-variables are variables defined at the Scheme top-level (such as `read`), or possibly primitives Ofa/Ova defined earlier in the same primitive operator definition-file.

[Notice that in Scheme, due to its dynamic binding of top-level defined names, a top-level definition such as `(define (read x) (read x))` is *not* equivalent to `(define read read)`: the former one redefines `read` to a non-terminating primitive whereas the latter one binds `read` to its former value and thus has no effect. This is the reason for distinguishing the forms *1f*/*1v* from *2f*/*2v*.]

The forms *1f* and *2f* provide Similix with primitive operator *arity* information. These forms should be preferred as mentioned in Section 3.3. The forms *1v* and *2v* should only be used for primitives that must be af variable arity; for instance, the primitive `list` is defined as a primitive with variable arity in `scheme.adt`. The arity of primitives of the form *1f* is given by the number of arguments, but the arity has to be specified explicitly for the form *2f* (this is consequence of the fact that there is no way to deduce the arity of a procedure/closure object in Scheme).

The `Key` specifies properties of primitive operators needed by the partial evaluator.

`defprim-transparent` and `defprim`:

> These two forms are equivalent. They specify that the primitive is referentially transparent, that is, does not performs any side-effects. For example, primitive operator `car` is specified with `defprim` in the file `scheme.adt`.

`defprim-tin`:

> [You may want to ignore the `defprim-tin` form as you can always use `defprim` instead: only termination properties and appearance of the residual program is changed, not safety (correctness of the generated residual programs).] The `defprim-tin` form also specifies that the primitive operator is transparent. It is used for transparent primitives that may be applied repeatedly an infinite number of times. For example, primitive operator `+` is specified with `defprim-tin` in the file `scheme.adt`. Notice that for instance primitive operator `car` can only be applied repeatedly a finite number of times to a value (assuming the value is acyclic). Specifying a primitive with `defprim-tin` does in some cases increase termination of specialization, at the expense of less reductions being performed.

`defprim-dynamic`:

> This form specifies a transparent primitive operator that should never be reduced by the partial evaluator. For example, the following primitive operator implements *generalization* [Tur86]:

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

> Generalization forces possibly static values to become dynamic. Operationally, `generalize` acts as the identity during program execution. But during partial evaluation, `generalize` is not reduced. Hence, any expression `(generalize E)` becomes dynamic, even if the argument `E` is static. Generalization provides the user a way to prevent *infinite specialization* (generating infinitely many specialized versions of a source procedure): generalize the argument that may assume infinitely many static values during specialization. Termination and generalization is discussed more in Section 7.4.

`defprim-opaque`:

This form specifies a primitive operator that is evaluation-order dependent: either it performs a side-effect itself or it depends on some (global) entity that is side-effected by other primitives. For example, primitive operator `read` is specified with `defprim-opaque` in the file `scheme.adt`: primitive operator `read` accesses and updates a global input stream. As dynamic primitives, opaque primitives are never reduced by the partial evaluator. In addition to this, the partial evaluator is careful to preserve the order in which opaque primitives are evaluated (when running the residual program).

Sometimes it is possible to define a primitive operator as dynamic rather than opaque, even if it makes use of a global (external) variable (in which case the primitive operator must *never* be specified as transparent). For instance, if a program uses primitive operators that access but never update some global variable(s), it is perfectly safe to define the primitive operators dynamic: there is no evaluation-order dependency.

`defprim-abort`:

This form specifies a primitive operator that aborts execution. An example is the specification of `_sim-error` in the file `scheme.adt`. Such primitives are never reduced by the partial evaluator.

`defprim-abort-eoi`:

This form also specifies an aborting primitive operator, but the form is more liberal: "eoi" stands for "evaluation-order independent" which indicates that the partial evaluator is allowed to ignore evaluation orders for aborting primitives of this kind. An example is the primitive operator `err` (see Section 4.2) used by an interpreter for reporting syntax errors in a program being interpreted: it does not matter which syntax error is reported first, so `defprim-abort-eoi` is used instead of `defprim-abort`. This gives better results when specializing, but notice that using `defprim-abort-eoi` is less safe than using `defprim-abort`: the semantics of a program may actually be altered when using `defprim-abort-eoi`. The form `defprim-abort-eoi` should therefore be used with care.

3.5 Restrictions on user-defined primitive operators

There are some important restrictions on how primitive operators may behave. These restrictions will be described in this section.

3.5.1 Higher-order values

A primitive operation is not allowed to return any higher-order value which is not passed in as an argument; a primitive operation may thus *not create new higher-order values.* Additionally, a primitive operator is not allowed to *apply* a higher-order value which is passed in as an argument. The system does *not* check that these restrictions are fulfilled. Here are two examples of illegal definitions of primitive operators:

```
(defprim (f x) (lambda (y) x))
(defprim 2 map map)
```

The first definition defines a primitive operator that creates a new higher-order value; the definition should therefore be converted into a procedure definition:

```
(define (f x) (lambda (y) x))
```

The second definition defines a primitive operator that applies its argument; the definition should therefore be converted into an explicit definition (the syntax category D, cf. Figure 3). In order not to get a name clash with Scheme's built-in `map`, some other name should be chosen, e.g.:

```
(define (my-map f l)
  (if (null? l)
      '()
      (cons (f (car l)) (my-map f (cdr l)))))
```

The standard Scheme forms `apply` and `for-each` should be treated similarly: these should be defined explicitly as procedures (note: there is no obvious definition for `apply`).

3.5.2 Primitive operators testing pointer equality

Primitives are *generally* not allowed to perform "pointer-equality" tests on there arguments; this is the reason why the primitives `eqv?`, `eq?`, `memq`, `memv`, `assq`, and `assv` are not built-in (cf. Section 3.3). User-defined primitives must obey the same restrictions; these restrictions are *not* checked by the system.

There are some cases where these restrictions may be liberalized, however. Thus, primitive operators may pointer-equality test arguments if the values the arguments evaluate to are *only* created, inspected, and modified by primitive operators; these primitive operators must *all* be defined by `defprim-dynamic` (or `defprim-opaque`). The values may also be (part of) of dynamic input to the program being specialized.

For example, we might specialize the program

```
(loadt "my-primitives.adt")
(define (f y)
  (let ((x (make-value))
    (my-op (cdr (cons y x)) (cdr (cons y x)))))
```

with `y` being dynamic. Supposing that the file `my-primitives.adt` contains the definitions

```
; These definitions are not ok:
(defprim 2 my-op eq?)
```

```
(defprim (make-value) '(1 2))
```

the residual program would become

```
(loadt "my-primitives.adt")
(define (f-0 y_0)
  (my-op (cdr (cons y_0 '(1 2)))
         (cdr (cons y_0 '(1 2)))))
```

This residual program is *incorrect* as for instance `(f 89)` evaluates to `#t` whereas `(f-0 89)` evaluates to `#f`. Another example yielding an incorrect result is obtained by replacing the call to `(make-value)` by the constant expression `'(1 2)`: now a value being pointer-equality tested is created by a quoted construction, thus it is neither created by a primitive operator defined with `defprim-dynamic`, nor does it come from the dynamic input (through the variable `y`) to the program.

However, we may change the definition of `make-value`:

```
; These definitions are ok:
(defprim 2 my-op eq?)
(defprim-dynamic (make-value) '(1 2))
```

Now the residual program becomes

```
(loadt "my-primitives.adt")
(define (f-0 y_0)
  (let ((x_1 (make-value)))
    (my-op (cdr (cons y_0 x_1)) (cdr (cons y_0 x_1)))))
```

and `(f-0 89)` now correctly evaluates to `#t`. (Note: if you in one session want to try to run these examples, both the incorrect ones and the correct one, you need the `loadt!`-form, see Section 8.3.5).

3.5.3 Side-effecting primitive operators

Primitives are *generally* not allowed to side-effect any of their arguments; this is why the primitives `set-car!`, `set-cdr!`, `string-set!`, and `vector-set!` are not built-in (cf. Section 3.3). The only kind of side-effects generally allowed by primitives are side-effects on *global* entities such as top-level bound variables or input/output (example: the primitive `read`); recall from Section 3.4 that primitives that either perform side-effects or depend on entities that are side-effected must all be defined by `defprim-opaque`. User-defined primitives must obey the same restrictions; these restrictions are *not* checked by the system.

There are some cases where these restrictions may be liberalized, however. Thus, primitive operators may side-effect arguments if the values the arguments evaluate to are *only* created, inspected, and modified by primitive operators; these primitive operators must *all* be defined by `defprim-opaque`. The values may also be (part of) of dynamic input to the program being specialized.

For example, Similix correctly specializes programs that manipulate "boxed" values by the following primitives only:

```
(defprim-opaque (box x) (cons x 'dummy))
(defprim-opaque (unbox x) (car x))
(defprim-opaque (set-box! x v) (set-car! x v))
```

Similix also correctly specializes programs that handle vectors (arrays) if these are only manipulated by primitives like the following ones:

```
(defprim-opaque my-make-vector make-vector)
(defprim-opaque 3 my-vector-set! vector-set!)
(defprim-opaque 2 my-vector-ref vector-ref)
(defprim-opaque 1 my-vector-length vector-length)
```

3.6 Scheme extension: user-defined constructors

User-defined constructors are defined in the same files as primitive operators, see Figure 7. A `defconstr`-form specifies a number of constructors, selectors, and constructor test-predicates which can then be used in expressions (cf. Figure 3). It is through user-defined constructors that Similix offers partially static data structures.

A `defconstr`-form defines a family of constructors. Such a family corresponds to a "disjoint sum of product" type in statically typed languages such as ML. You should always group constructors together in a family if they logically belong to the same sum type! *In particular, notice that at partial evaluation time Similix makes values dynamic when constructors from different families are mixed!*

For each constructor in a family, a selector name must be specified for each argument field of the constructor. The symbol * may be supplied instead of a name; then the selector is given a default name `C.`i where i is the position of the field (the fields are numbered $0, 1, \ldots$). The *-form is particularly relevant in connection to the `caseconstr`-form, see Section 3.8. *When using the *-form, beware that the automatically defined selector names do not clash with other names!*

A constructor test-predicate is defined automatically for each constructor specified. For each constructor `C`, the name of the test-predicate is `C?`. *Beware that automatically defined test-predicate names do not clash with other names!*

Here are some examples of constructor definitions:

```
(defconstr (makepair fst snd))
```

```
(defconstr (mynil) (mycons mycar mycdr))
```

We can now implement association lists (environments) in the following way:

```
(define (init-env) (mynil))

(define (upd-env name value alist)
  (mycons (makepair name value) alist))

(define (lookup-value name alist)
  (let loop ((a alist))
    (cond
      ((mynil? a)
       (_sim-error 'lookup-value "Name ~s not bound" name))
      ((mycons? a)
       (let ((p (mycar a)))
         (if (equal? name (fst p)) (snd p) (loop (mycdr a)))))
      (else
       (_sim-error 'lookup-value "Ill-formed environment")))))
```

Such association lists may become partially static when specializing a program using them; see Section 4.2 for an example.

It is important to notice that the *only* way to obtain partially static data structures with Similix is by using user-defined constructors. In particular, notice that the built-in `cons` (see Figure 5) is a primitive operator, *not* a constructor: primitive operator `cons` *cannot* be used for creating partially static data. The same applies to the primitive operator `list`.

Also, recall from Section 3.1 that values constructed by user-defined constructors are not allowed in static input when specializing. This in effect means that Similix *does not handle partially static input.*

3.7 Primitive operator and constructor name clashes

If a name is defined (either as primitive operator, constructor, selector or constructor test-predicate) more than once within a file or within different files loaded by the same program (by `loadt`), only the *last* definition counts.

3.8 Scheme extension: pattern matching

Similix provides two pattern-matching extensions to Scheme, `casematch` and `caseconstr` (see Figure 3). The `casematch`-form is for matching ordinary Scheme S-expressions constructed by `cons` (which is considered a primitive operator by Similix); the `caseconstr`-form is for matching values constructed by user-defined constructors (cf. Section 3.6).

The semantics of both `casematch` and `caseconstr` is standard: the expression is evaluated to a value which is then matched against the patterns (starting from the top) until a match is found. The variables in the matched pattern are bound to the appropriate components in the value, and the expression(s) corresponding to the pattern is (are) evaluated.

For `casematch`, a pattern MPat (see Figure 4) is either a constant K, the empty list `()` (that is, no `quote` is needed for the empty list), a pair `(MPat . MPat)`, a variable V, or a wildcard pattern (either `_` or `else`: the two forms are fully equivalent). Using `casematch` is exemplified in Section 4.2.1.

For `caseconstr`, a pattern CPat (see Figure 4) is either a constructor pattern (C CPat*), a variable V, or a wildcard pattern (either `_` or `else`: the two forms are fully equivalent). When `caseconstr` is expanded into the Similix core language (described in Section 3.10), the default selector names C.i (cf. Section 3.6) are used. *That is,* `caseconstr` *can only be used for constructors defined with the *-form for all argument fields:*

```
(defconstr (C * ...) ...)
```

The restriction that `caseconstr` is only used for constructors specified with the *-form is *not* checked by the system. Using `caseconstr` is exemplified in Section 4.2.1.

Notice that MPat is entirely for use with `casematch` whereas CPat is entirely for use with `caseconstr`. Both `casematch` and `caseconstr` are exemplified in Section 4.2.

The form `(loads F)` (cf. Figure 3) is similar to the ordinary Scheme form `(load F)` except that it expands the forms `casematch` and `caseconstr` into simpler forms that do not use pattern matching. The form `(loads F)` can be used both at Scheme top-level to load programs that use the pattern-matching forms and it can be used in program files, cf. Figure 3.

3.9 Simulating `set!`

Similix does not handle assignment by `set!`, but `set!` can be simulated by transforming the source program. There are two cases, assignment to top-level bound variables and assignment to locally bound variables.

3.9.1 Top-level bound variables

In the Similix Scheme subset, top-level bound variables (top-level defined procedures being an exception) are only accessible through user-defined primitive operators which should be defined by `defprim-opaque`. Suppose you would like to write a program containing the fragment

```
... topvar ... (set! topvar ...) ...
```

where `topvar` is top-level defined. To convert this fragment into the Scheme subset treated by Similix, write for instance

```
... (get-topvar) ... (set-topvar! ...) ...
```

where `get-topvar` and `set-topvar!` are defined as primitive operators:

```
(defprim-opaque (get-topvar) topvar)
(defprim-opaque (set-topvar! value) (set! topvar value))
```

The store-operations in the interpreter in Section 4.1.1 give another example of global variable handling in the Similix Scheme subset.

3.9.2 Locally bound variables

Assignment to locally bound variables can be done by using the user-defined primitive operations for boxed values given in Section 3.5.3. Suppose the local variable is defined in a let-expression

```
(let ((localvar ...)) E)
```

where `E` contains operations of the forms `(set! localvar ...)`.

To convert into the Similix subset, replace the let-binding by

```
(let ((localvar (box ...))) E)
```

Then, in `E`, replace all references to (uses of) `localvar` by `(unbox localvar)`, and replace all forms `(set! localvar ...)` by `(set-box! localvar ...)`. Remember to include definitions of the boxed-value operations in some file referred to by a `loadt`-expression.

If `localvar` is a parameter to a procedure or anonymous lambda-expression (and thus not defined in a let), insert a let-expression around the body `B` of the procedure/anonymous lambda-expression:

```
(let ((localvar (box localvar))) B)
```

Then perform the same changes regarding occurrences of `localvar` in `B` as we did in `E` above.

3.10 Similix core language

The Similix front-end parser expands a number of the forms in the full Similix language from Figure 3 and Figure 4 into simpler core forms. The core language, specified in Figure 8, is a subset of the full Similix language. Preprocessed programs are written in an annotated variant of the core language (Section 6).

The expansion into core form is done as follows:

- `(load F)` and `(loads F)` are replaced by the text in the referenced file `F`.
- The form `B` is expanded to a `letrec`-form which in turn is expanded further (see below). If there are no "internal definitions" (D^* is empty), E^+ is expanded to a `begin`-form if there is more than one expression `E`.

$\Pi \in$ Program ; D $\in$ Definition ; E $\in$ Expression ; K $\in$ Constant ; V $\in$ Variable ;
O $\in$ PrimopName ; C $\in$ ConstructorName ; S $\in$ SelectorName ; P $\in$ ProcedureName ;
SE $\in$ SelfEvaluating ; Dat $\in$ Datum ; Bool $\in$ Boolean ; Num $\in$ Number ;
Char $\in$ Character ; Str $\in$ String ; Sym $\in$ Symbol ; Lis $\in$ List ; Vec $\in$ Vector ;

Π	::=	TLE* D TLE*	
TLE	::=	D | `(loadt F)`	
D	::=	`(define (P V*) B)`	procedure definition
E	::=	K	constant
	|	V	variable
	|	`(if E E E)`	conditional
	|	`(let ((V E)) E)`	let-expression
	|	`(begin E E)`	sequence
	|	`(O E*)`	primitive operation
	|	`(C E*)`	constructor application
	|	`(S E)`	selector application
	|	`(C? E)`	constr. test-predicate application
	|	`(P E*)`	procedure call
	|	`(lambda (V*) E)`	lambda-expression
	|	`(E E*)`	application
V, P, C, S	::=	Sym	
O	::=	Ofa | Ova	
Ofa	::=	...	see Figure 5
Ova	::=	...	see Figure 6
K	::=	SE | `(quote Dat)` | `'Dat`	
SE	::=	Bool | Num | Char | Str	
Dat	::=	SE | Sym | Lis | Vec	
Lis	::=	`(Dat*)` | `(Dat+ . Dat)` | `'Dat`	
Vec	::=	`#(Dat*)`	

Figure 8: Similix core language

- One-armed conditionals `(if E E)` are expanded into ordinary conditionals with a dummy-value inserted in the else-branch.
- `cond`-conditionals are expanded into nested `if`-expressions.
- The `and`- and `or`-forms are expanded into `if`-forms (`or` is actually expanded into a combination of `let` and `if`).

- Parallel and sequential let-expressions are expanded into nested simple let-expressions, each with only one binding.
- Named (recursive) let-expressions are expanded into `letrec` which in turn is expanded further (see below).
- Recursive let-expressions (`letrec`) are expanded by lambda-lifting [Joh85]: variables that are free in the bodies of the defined procedures are added as parameters and the definitions are lifted out to the top-level.
- Sequences are expanded to nested simple sequences with only two expressions in each `begin`-form.
- Fixed-arity primitive operators that are not in apply-position are eta-expanded: the form Ofa expands into (lambda ($V_1 \ldots V_n$) (Ofa $V_1 \ldots V_n$)).
- Constructors, selectors, constructor test-predicates, and procedure names that are not in apply-position are also eta-expanded.
- Pattern-matching forms are expanded into appropriate expressions for testing matches (conditionals) and binding variables (let-expressions).

4 Examples

This section contains some examples of Similix-applications. The examples are all interpreters: by specializing the interpreters, compilation into Scheme is obtained. The programs shown in this section can all be found in the `examples` directory. The associated job files (`MP-job`, `mw-job`, `com-job`) may be used to reproduce the results; we recommend that you do so while reading this section.

4.1 Specializing an MP-interpreter

We first specialize an interpreter for the toy language "MP" (introduced in [Ses85]).

4.1.1 The MP-interpreter

MP is a small imperative untyped "while" language with Lisp data structures, assignments, conditionals, and while-loops. The interpreter is given in Figure 9 and Figure 10.

```
; P ::= (program (pars V1*) (vars V2*) B)
; B ::= (C*)
; C ::= (:= V E)
;     | (if E B1 B2)            # first branch iff exp not ()
;     | (while E B)             # loop iff Exp not ()
```

```
; E ::= (quote Value)
;     | V
;     | (car E)
;     | (cdr E)
;     | (cons E1 E2)
;     | (atom E)               # () iff not atom
;     | (equal E1 E2)          # () iff not equal
;
; value: Value = ...
; env:   Env   = Var -> Loc
;        Store = Loc -> Value

(loadt "MP-int.adt")

(define (run P value*)
  (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)
  (if (emptyBlock? B)
      "Finished block"
      (evalCommands (headBlock B) (tailBlock B) env)))

(define (evalCommands C B env)
  (if (emptyBlock? B)
      (evalCommand C env)
      (begin (evalCommand C env)
             (evalCommands (headBlock B) (tailBlock B) env))))

(define (evalCommand C env)
  (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
     (err 'evalCommand "Unknown command: ~s" C))))

(define (evalExpression E env)
  (cond
```

```
    ((isQuote? E)
     (E->E1 E))
    ((isVariable? E)
     (lookup-store (lookup-env E env)))
    ((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)
          (eval-equal (evalExpression (E->E1 E) env)
                      (evalExpression (E->E2 E) env)))
         ((is-car? op)
          (car (evalExpression (E->E1 E) env)))
         ((is-cdr? op)
          (cdr (evalExpression (E->E1 E) env)))
         ((is-atom? op)
          (eval-atom (evalExpression (E->E1 E) env)))
         (else
          (err 'evalExpression "Unknown operator: ~s" op)))))
    (else
     (err 'evalExpression "Unknown expression: ~s" E))))
```

Figure 9: MP-interpreter (file `MP-int.sim`*)*

```
; Syntax:

(defprim 1 P->V1* cdadr)
(defprim 1 P->V2* cdaddr)
(defprim 1 P->B cadddr)

(defprim 1 emptyBlock? null?)
(defprim 1 headBlock car)
(defprim 1 tailBlock cdr)

(defprim 1 C-Assignment->V cadr)
(defprim 1 C-Assignment->E caddr)
(defprim 1 C-Conditional->E cadr)
(defprim 1 C-Conditional->B1 caddr)
(defprim 1 C-Conditional->B2 cadddr)
(defprim 1 C-While->E cadr)
(defprim 1 C-While->B caddr)

(defprim (isAssignment? c) (and (pair? c) (equal? (car c) ':=)))
(defprim (isConditional? c) (and (pair? c) (equal? (car c) 'if)))
(defprim (isWhile? c) (and (pair? c) (equal? (car c) 'while)))
```

```
(defprim (isQuote? e) (and (pair? e) (equal? (car e) 'quote)))
(defprim (isVariable? e) (not (pair? e)))
(defprim (isPrim? e)
  (and (pair? e) (member (car e) '(cons equal car cdr atom))))

(defprim (is-cons? op) (equal? op 'cons))
(defprim (is-equal? op) (equal? op 'equal))
(defprim (is-car? op) (equal? op 'car))
(defprim (is-cdr? op) (equal? op 'cdr))
(defprim (is-atom? op) (equal? op 'atom))

(defprim 1 E->operator car)
(defprim 1 E->E1 cadr)
(defprim 1 E->E2 caddr)

;-------------------------------------
; True and false ---
; the empty list () counts as false in MP:

(defprim (eval-equal v1 v2) (if (equal? v1 v2) #t '()))
(defprim (eval-atom v) (if (pair? v) '() #t))

(defprim (is-true? value) (not (null? value)))

;-------------------------------------
; Environment:

(defprim (init-environment v1* v2*) (append v1* v2*))
(defprim (lookup-env v env)
  (let f ((env env) (n 0))
    (if (equal? v (car env)) n (f (cdr env) (+ 1 n)))))

;-------------------------------------
; Store:

(defprim-opaque (init-store! input-V1* length-V2*)
  (set! store
        (append
         input-V1*
         (let f ((n length-V2*))
           (if (= n 0) '() (cons '() (f (- n 1))))))))
(defprim-opaque (update-store! location value)
  (set-car! (list-tail store location) value))
(defprim-opaque (lookup-store location) (list-ref store location))

;-------------------------------------
; Error:

(defprim-abort-eoi err _sim-error)
```

Figure 10: The file `MP-int.adt`

As it can be seen from the syntax of MP (defined in Figure 9), there are two kinds of variables, declared by `pars` and `vars`. The "pars" are input parameters, the "vars" are ordinary variables. The semantics is the straightforward one; notice that the empty list `()` counts as "false". The result of an execution is taken to be the entire store.

This is an example of an MP-program (coming from [Ses85]):

```
(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)))))))))
```

Figure 11: The MP-program power (file `power.MP`*)*

The program computes `x` to the `y`'th, where numbers are represented as lists (unary representation). It is not important here how the program actually works, it simply serves as an example of a program to be compiled.

The interpreter uses an environment (`env`) and a store. The environment binds variables to locations; it is processed by the primitive operations `init-environment` and `lookup-env`. The store binds locations to values. An 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 locations (and values) as parameters, not the store itself. The store is implemented as a global variable which is updated destructively, and the store primitives (defined in the file `MP-int.adt`) are hence defined by `defprim-opaque` (see Figure 10).

As it can be seen from the definitions of store primitives, 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 implementation.

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

4.1.2 Specializing the MP-interpreter

Let us now specialize the MP-interpreter with respect to the MP-program from Figure 11. This yields the following Scheme target program:

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

(define (run-0 value*_0)
  (define (evalcommand-0-2)
    (let ((g_0 (lookup-store 3)))
      (if (is-true? g_0)
          (begin
            (let* ((g_1 (lookup-store 3))
                   (g_2 (car g_1))
                   (g_3 (cdr g_2)))
              (if (is-true? g_3)
                  (let* ((g_4 (lookup-store 3))
                         (g_5 (car g_4))
                         (g_6 (cdr g_5))
                         (g_7 (lookup-store 3))
                         (g_8 (cdr g_7))
                         (g_9 (cons g_6 g_8)))
                    (update-store! 3 g_9)
                    (evalcommand-0-1)
                    (let* ((g_10 (lookup-store 3))
                           (g_11 (lookup-store 2))
                           (g_12 (cons g_10 g_11)))
                      (update-store! 2 g_12)))
                  (let* ((g_13 (lookup-store 3))
                         (g_14 (cdr g_13)))
                    (update-store! 3 g_14)
                    (let* ((g_15 (lookup-store 4))
                           (g_16 (cons 1 g_15)))
                      (update-store! 4 g_16)))))
            (evalcommand-0-2))
          "Finished loop")))
  (define (evalcommand-0-1)
    (let ((g_0 (lookup-store 4)))
      (if (is-true? g_0)
          (let* ((g_1 (lookup-store 0))
                 (g_2 (lookup-store 3))
                 (g_3 (cons g_1 g_2)))
            (update-store! 3 g_3)
            (let* ((g_4 (lookup-store 4))
                   (g_5 (cdr g_4)))
              (update-store! 4 g_5)
```

```
              (evalcommand-0-1)))
          "Finished loop")))
  (init-store! value*_0 3)
  (let ((g_1 (lookup-store 1)))
    (update-store! 4 g_1)
    (evalcommand-0-1)
    (let* ((g_2 (lookup-store 3))
           (g_3 (lookup-store 2))
           (g_4 (cons g_2 g_3)))
      (update-store! 2 g_4)
      (evalcommand-0-2))))
```

Figure 12: Compiled power program

The structure of the target program is quite close to assembler code. 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 (both program syntax and environments were completely static), 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, `evalcommand-0-1`. This is of course possible since both while loops perform the same operations. The specializer is lucky to detect this because both loops are *textually* identical. They therefore correspond to identical static values for the parameter `C` to `eval-command`.

4.1.3 Generating an MP-compiler

The Similix compiler generator can generate an MP-compiler from the interpreter. Using the generated compiler, target programs are generated significantly faster than by specializing the interpreter. The compiler text is too large to show here, but you may generate the compiler by running the `MP-job` in the `examples` directory.

4.2 Specializing a Mixwell-interpreter

We now specialize an interpreter for the Mixwell language of [JSS89].

4.2.1 The Mixwell-interpreter

Mixwell is a first-order Lisp-like functional language. The interpreter is given in Figure 13 and Figure 14.

```
; P ::= (D1 D2 ... Dn)
; D ::= (F (V1 ... Vn) = E)
```

```
; E ::= V | (quote C)
;     | (car E) | (cdr E) | (atom E) | (cons E E) | (equal E E)
;     | (if E E E) | (call F E1 ... En)
;------------------------------------
(loadt "mw-int.adt")
;------------------------------------
(define (run-mixwell P vals)
  (casematch P
    (((_ Vs '= E) . _)
     (ev E
         (let ((arity (length Vs)))
           (let loop ((i 0))
             (if (= i arity)
                 (init-env)
                 (upd-env (list-ref Vs i)
                          (list-ref vals i)
                          (loop (+ 1 i))))))
         P))
    (else
     (err 'run-mixwell "Illegal program syntax: ~s" P))))

(define (ev E r P)
  (if (symbol? E)
      (lookup-env E r) ; E = variable V
      (casematch E
        (('quote C)
         C)
        (('car E)
         (car (ev E r P)))
        (('cdr E)
         (cdr (ev E r P)))
        (('atom E)
         (not (pair? (ev E r P))))
        (('cons E1 E2)
         (cons (ev E1 r P) (ev E2 r P)))
        (('equal E1 E2)
         (equal? (ev E1 r P) (ev E2 r P)))
        (('if E1 E2 E3)
         (if (ev E1 r P) (ev E2 r P) (ev E3 r P)))
        (('call F . Es)
         (let ((D (lookup-function F P)))
           (casematch D
             ((F Vs = E)
              (ev E
                  (let loop ((Vs Vs) (Es Es))
                    (if (null? Vs)
```

```
                        (init-env)
                        (upd-env (car Vs)
                                 (ev (car Es) r P)
                                 (loop (cdr Vs) (cdr Es)))))
                  P))
             (else
              (err 'ev "Illegal definition syntax: ~s" D)))))
        (else
         (err 'ev "Illegal expression syntax: ~s" E)))))

(define (init-env) (bindings-nil))

(define (upd-env V val r) (bindings-cons (binding V val) r))

(define (lookup-env V r)
  (let loop ((bs r))
    (caseconstr bs
      ((bindings-nil)
       (err 'lookup-env "Name ~s not bound" V))
      ((bindings-cons (binding V1 val) bs)
       (if (equal? V V1) val (loop bs)))
      (else ; no bs argument:
       (err 'lookup-env "Internal error: illegal environment")))))
```

Figure 13: Mixwell-interpreter (file `mw-int.sim`*)*

```
(defprim 2 lookup-function assoc)

(defconstr (binding * *))
(defconstr (bindings-nil) (bindings-cons * *))

(defprim-abort-eoi err _sim-error)
```

Figure 14: The file `mw-int.adt`

The Mixwell-interpreter illustrates using `casematch` for syntax dispatch; notice that the MP-interpreter used primitive operators. One could also have used (user-defined) procedures. What to use is mainly a matter of taste (however, see Section 7.3.4).

The environment is represented as a list generated by user-defined constructors. Defining it by user-defined constructors is crucial to make it partially static: the names become static and the values become dynamic. In the MP-interpreter, both names and values (which were location there) were static, so there we could successfully process the environment by primitive operators. If we had done so here, the environment would have become completely dynamic with bad residual programs as a consequence. Notice that in the MP-interpreter, we could have used the environment operations from the Mixwell-interpreter.

The Mixwell-interpreter also illustrates using `caseconstr` for testing and decomposing values constructed by user-defined constructors. Notice that the constructors (`binding`, `bindings-nil`, and `bindings-cons`) are defined using the *-form for argument fields, not by specifying selector names: otherwise, the code which `caseconstr` expands into would not be correct.

Here is an example of a Mixwell-program for appending two lists:

```
(goal (x y) = (call app x y))

(app (x y) =
     (if (equal x '())
         y
         (cons (car x) (call app (cdr x) y))))
```

Figure 15: The Mixwell-program app (file `app.mw`*)*

4.2.2 Specializing the Mixwell-interpreter

Let us now specialize the Mixwell-interpreter with respect to the Mixwell-program from Figure 15. This yields the following Scheme target program:

```
(loadt "mw-int.adt")

(define (run-mixwell-0 vals_0)
  (define (ev-0-1 r_0 r_1)
    (if (equal? r_1 '())
        r_0
        (cons (car r_1) (ev-0-1 r_0 (cdr r_1)))))
  (ev-0-1 (list-ref vals_0 1) (list-ref vals_0 0)))
```

Figure 16: Compiled app program

The procedure `ev-0-1` is identical to the "standard" append-program in Scheme. The "overhead" is some initialization caused by the fact that the input to the residual program is packed into a list.

4.2.3 Generating a Mixwell-compiler

The Similix compiler generator can generate a Mixwell-compiler from the interpreter. Using the generated compiler, target programs are generated significantly faster than by specializing the interpreter. The compiler text is too large to show here, but you may generate the compiler by running the `mw-job` in the `examples` directory.

4.3 Specializing a $\frac{\mathcal{LA}}{\mathcal{ZY}}$-interpreter

We finally specialize an interpreter for $\frac{\mathcal{LA}}{\mathcal{ZY}}$, a lazy functional curried named combinator language [Bon91b].

4.3.1 The $\frac{\mathcal{LA}}{\mathcal{ZY}}$-interpreter

The language $\frac{\mathcal{LA}}{\mathcal{ZY}}$ is a lazy functional curried named combinator language. The interpreter is given in Figure 17, Figure 18, and Figure 19.

```
; P ::= D*
; D ::= (F V* = E)
; E ::= C | V | F | (B E1 E2) | (if E1 E2 E3) | (E1 E2)

; Parsed form:
; P ::= (D*)
; D ::= (F (V*) E)
; E ::= (cst C) | (var V) | (fct F) | (binop B E1 E2)
;     | (if E1 E2 E3) | (apply E1 E2)

;------------------------------------
(loadt "com-int.adt")
(loadt "thunk.adt")

;------------------------------------
; Values are delayed for two resons:
; (1) Environment updating is done by strict functions; therefore,
;     the value argument is delayed (and then forced at lookup-time).
; (2) The interpreted language is lazy so arguments to applications
;     are delayed.

(define (init-fenv)
  (lambda (name)
    (err 'init-fenv "Unbound function: ~s" name)))
(define (upd-fenv name value r)
  (lambda (name1)
    (if (equal? name name1)
        (value) ; force value
        (r name1))))

(define (init-venv)
  (lambda (name)
    (err 'init-venv "Unbound variable: ~s" name)))
(define (upd-venv name value r)
  (lambda (name1)
    (if (equal? name name1)
        (value) ; force value
        (r name1))))
```

```
;-------------------------------------
(define (_P P F v) (((fix (lambda (phi) (_D* P phi))) F) (lambda () v)))

(define (_D* D* phi)
  (casematch D*
    (()
     (init-fenv))
    (((F V* E) . D*)
     (upd-fenv F
               (lambda () (_V* V* E (init-venv) phi)) ; delay value
               (_D* D* phi)))
    (else
     (err '_D* "Illegal program syntax: ~s" D*))))

(define (_V* V* E r phi)
  (casematch V*
    (()
     (_E E r phi))
    ((V . V*)
     (lambda (s) (_V* V* E (upd-venv V (lambda () s) r) phi))) ; delay value
    (else
     (err '_V* "Illegal parameter syntax: ~s" V*))))

(define (_E E r phi)
  (casematch E
   (('cst C)
    C)
   (('var V)
    ((r V))) ; force value
   (('fct F)
    (phi F))
   (('binop B E1 E2)
    (ext B (_E E1 r phi) (_E E2 r phi)))
   (('if E1 E2 E3)
    (if (_E E1 r phi)
        (_E E2 r phi)
        (_E E3 r phi)))
   (('apply E1 E2)
    ((_E E1 r phi)
     (casematch E2
       (('cst C) (lambda () C))
       (('var V) (r V))
       ;;(('fct F) (lambda () (phi F)))
       (else
        (save (lambda () (_E E2 r phi))))))) ; delay value
   (else
    (err '_E "Illegal expression syntax: ~s" E))))
```

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

Figure 17: ${}^{\mathcal{LA}}_{\mathcal{ZY}}$-interpreter (file `com-int.sim`*)*

```
(defprim (ext binop value1 value2)
  (case binop
    ((cons) (cons value1 value2))
    ((hack-car) (car value1))
    ((hack-cdr) (cdr value1))
    ((equal?) (equal? value1 value2))
    ((+) (+ value1 value2))
    ((-) (- value1 value2))
    ((*) (* value1 value2))
    ((/) (/ value1 value2))
    ((=) (= value1 value2))))

(defprim-abort-eoi err _sim-error)
```

Figure 18: The file `com-int.adt`

```
(defprim-dynamic (save s)
  (let ((v '())
        (tag #t))
    (lambda ()
      (if tag
          (begin
            (set! v (s))
            (set! tag #f)))
      v)))
```

Figure 19: The file `thunk.adt`

The interpreter uses “delay and force”, a classical way of implementing laziness (call-by-need) in strict (call-by-value) languages. Notice that environments are represented by *functions* here, not data structures. We could have used similar representations in the MP- and Mixwell-interpreter examples.

Below is an example of a ${}^{\mathcal{LA}}_{\mathcal{ZY}}$-program. The program computes a list of even numbers; the input to the goal function `goal` specifies how long the list should be. Notice that the program utilizes laziness in the definition of `evens-from`. Also notice that since the interpreter implements `cons` (which is a “binop”) eagerly (call-by-value), a special `lazy-cons` is used to construct the infinite list specified by `evens-from`; the definition of `lazy-cons` is the standard functional λ-calculus one.

```
(first-n n l  = if (= n 0)
                '()
                (cons (lazy-car l) (first-n (- n 1) (lazy-cdr l))))

(evens-from n = lazy-cons n (evens-from (+ n 2)))

(lazy-cons x y z = z x y)
(lazy-car x      = x 1st)
(lazy-cdr x      = x 2nd)

(1st x y = x)
(2nd x y = y)

(goal input = first-n input (evens-from 0))
```

Figure 20: The $\substack{\mathcal{LA}\\\mathcal{ZY}}$-program evens (file `evens.com`*)*

4.3.2 Specializing the $\substack{\mathcal{LA}\\\mathcal{ZY}}$-interpreter

Let us now specialize the $\substack{\mathcal{LA}\\\mathcal{ZY}}$-interpreter with respect to the $\substack{\mathcal{LA}\\\mathcal{ZY}}$-program from Figure 20. This yields the following Scheme target program:

```
(loadt "com-int.adt")

(loadt "thunk.adt")

(define (_p-0 v_0)
  (define (_v*-0-16)
    (lambda (s_0)
      (let ((s_2 (save
                   (lambda ()
                     ((_v*-0-16)
                      (save (lambda () (ext '+ (s_0) 2))))))))
        (lambda (s_3) (((s_3) s_0) s_2)))))
  (define (_v*-0-2)
    (lambda (s_0)
      (lambda (s_1)
        (if (ext '= (s_0) 0)
            '()
            (ext 'cons
                 ((s_1)
                  (save (lambda ()
                          (lambda (s_3) (lambda (s_4) (s_3))))))
                 (((_v*-0-2) (save (lambda () (ext '- (s_0) 1))))
```

```
                (save
                 (lambda ()
                   ((s_1)
                    (save (lambda ()
                            (lambda (s_6)
                              (lambda (s_7) (s_7)))))))))))))
(((_v*-0-2) (lambda () v_0))
 (save (lambda () ((_v*-0-16) (lambda () 0))))))
```

Figure 21: Compiled evens program

The program looks quite complicated at a first sight, but it turns out that it actually closely corresponds to the source program in Figure 20, the main differences being syntax and the explicit delay/force operations.

4.3.3 Generating a $\frac{\mathcal{LA}}{\mathcal{ZY}}$-compiler

The Similix compiler generator can generate a $\frac{\mathcal{LA}}{\mathcal{ZY}}$-compiler from the interpreter. Using the generated compiler, target programs are generated significantly faster than by specializing the interpreter. The compiler text is too large to show here, but you may generate the compiler by running the `com-job` in the `examples` directory.

5 System Overview

This section is quite technical and gives an overview of the Similix system.

In Similix, partial evaluation is done in two phases. First, the source program is preprocessed, then the preprocessed program is specialized (see Figure 22). The residual program is generated in the specialization phase. We use phrases like "at specialization time" and "during specialization" to refer to operations done in the specialization phase.

5.1 The front-end

The front-end is a parser: it expands programs written in the language of Figure 3 and Figure 4 into the core language of Figure 8. The resulting code is represented in an internal abstract syntax format; this format is an acyclic Scheme data structure representation, so the code may for instance be printed.

Conceptually, the front-end may be seen as part of the preprocessor, but it is sometimes useful to run the front-end alone: the front-end performs various syntax checks such as arity checks on primitive operations and procedure calls. The front-end may therefore be used for program debugging, even if partial partial evaluation is not intended.

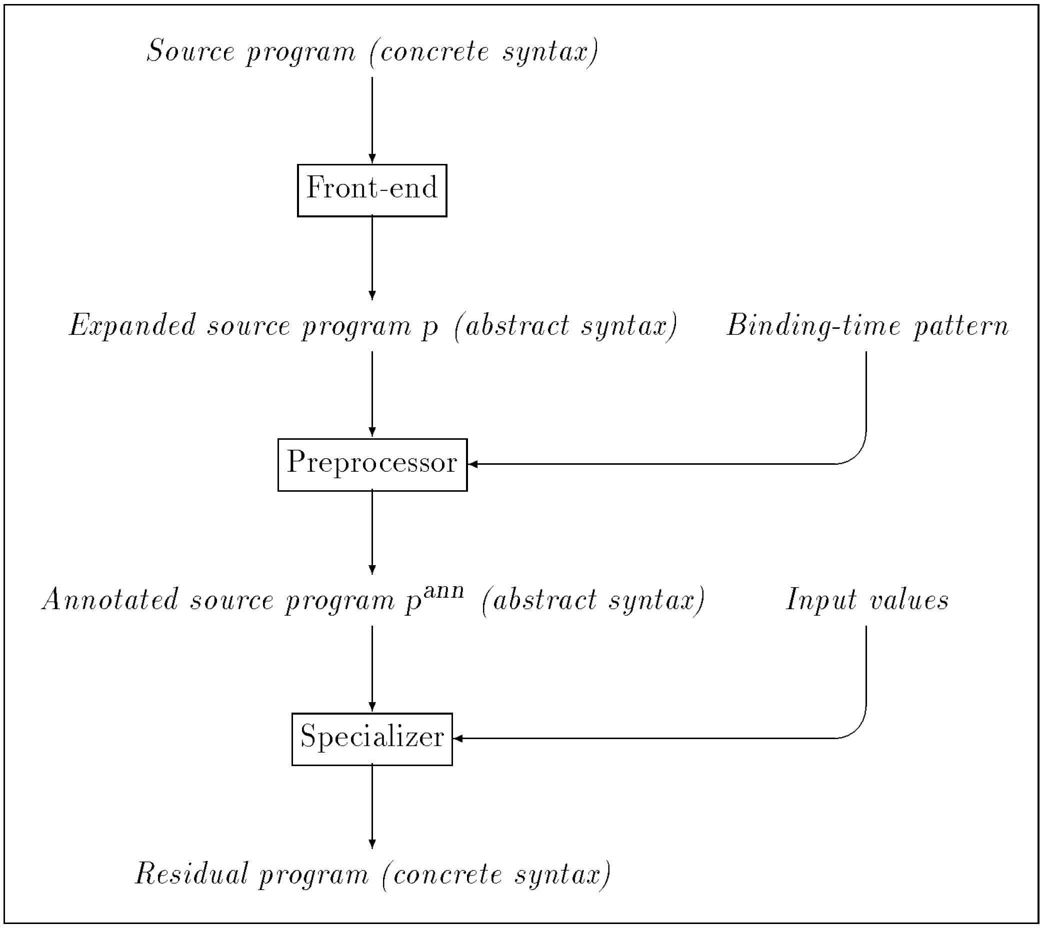


Figure 22: Similix system

5.2 The preprocessor

The preprocessor consists of several subphases as seen in Figure 23. Each phase updates the analysed program destructively while computing source program annotations.

The result of preprocessing is a heavily annotated source program which is used as input to the specializer.

5.2.1 Flow analysis

This phase determines possible value flow between constructor applications and selector/predicate applications, and it determines possible value flow between lambda-expressions and application points. The flow analysis is described in [BJ93b].

5.2.2 Binding-time analysis

This phase propagates binding-time information about the program input — the binding-time pattern in Figure 23 — through the program. Each program expression and each

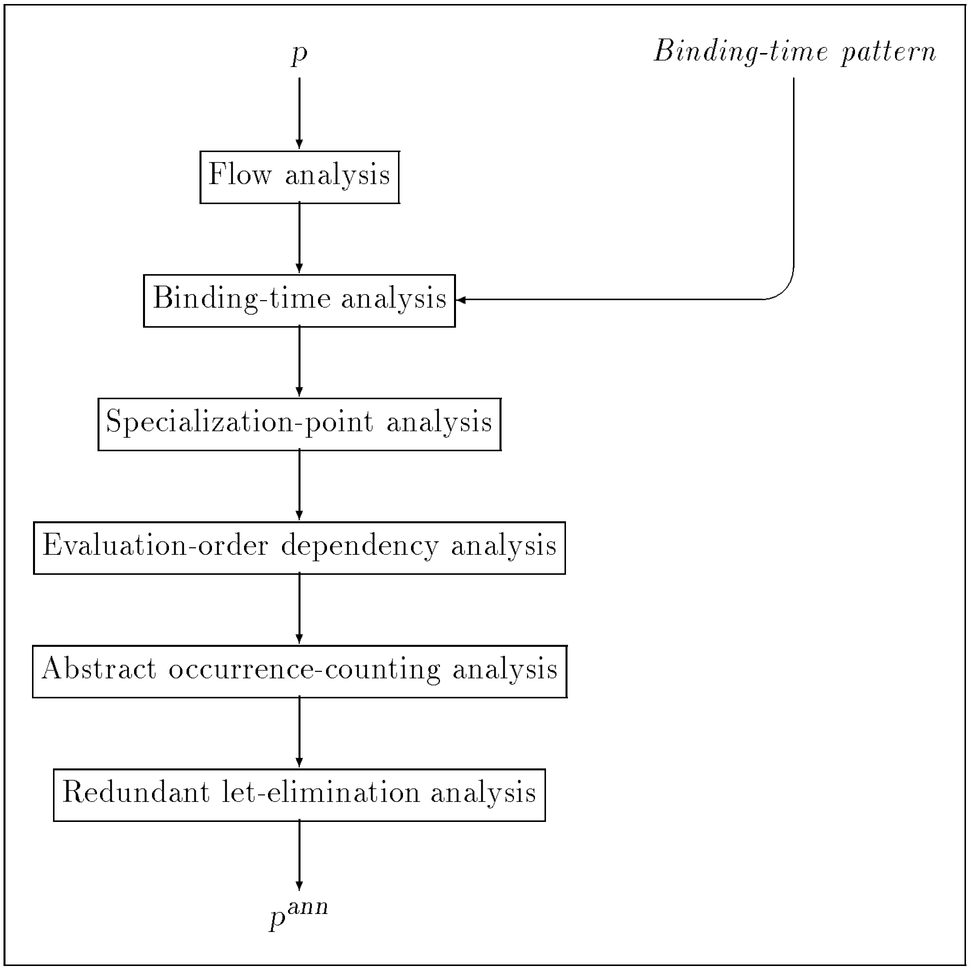
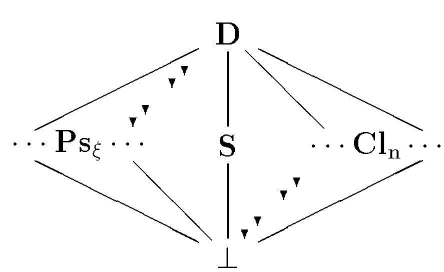


Figure 23: Similix preprocessor

variable gets a binding-time value. The information may be read as type information about specialization-time expression return-values and specialization-time values bound to variables.

The domain of binding-time values is the following lattice:



- The binding-time value $\mathbf{S}$ describes first-order static values.
- The binding-time values $\mathbf{Ps}_\xi$ describe partially static values (generated by user-defined constructors; there is one binding-time value $\mathbf{Ps}_\xi$ for each constructor family ξ used in

the analysed program p).

- The binding-time values $\mathbf{Cl}_\mathrm{n}$ describe static higher-order values (there is one binding-time value $\mathbf{Cl}_\mathrm{n}$ for each function/procedure arity n used in the analysed program p).

- The binding-time value $\mathbf{D}$ describes residual expressions (dynamic values).

- The binding-time value $\perp$ means "no value yet". If a program has occurrences of $\perp$ in a program fragment after binding-time analysis, the fragment is either never used or definitely always non-terminating.

As indicated by the binding-time domain, *the binding-time analysis makes higher-order values dynamic if they flow together with other higher-order values with a different arity.* Similarly, *if constructed values from different constructor families flow together, the values become dynamic rather than partially static.*

The *specializer* does not distinguish between different $\mathbf{Cl}_\mathrm{n}$-values, nor does it distinguish between different $\mathbf{Ps}_\xi$-values; the annotations given to the specializer (and the ones shown to the user, see Section 6) are therefore collapsed into **Ps** and **Cl**.

The binding-time analysis is described in [BJ93b].

5.2.3 Specialization-point analysis

This phase finds specialization/memoization points. The Similix specializer is memoizing (as was for instance Mix [JSS85]): if it during specialization encounters the same specialization point expression E more than once, it checks whether the non-dynamic parts of the values of the free variables of E have been seen before. If yes, a call to the previously generated code is generated. It is the memoization that generates residual definitions in the residual program.

Where memoization points have been inserted can be inspected by the user (see Section 6); the user can also explicitly control insertion of memoization points (see Section 7.2.3). The built-in strategy inserts specialized points at dynamic conditionals (conditionals with dynamic test) and at dynamic lambda-expressions (lambda-expressions which do not get beta-reduced); this strategy is described in [BD91, Bon90b].

5.2.4 Evaluation-order dependency analysis

The evaluation-order dependency analysis finds expressions that may possibly be evaluation-order dependent. Such expressions arise from opaque primitive operations and are always dynamic (cf. Section 3.4: opaque operations are always kept residual). The analysis is used to prevent unfolding let-expressions when the actual parameter is potentially evaluation-order dependent. The analysis is described in [BJ93a].

5.2.5 Abstract occurrence-counting analysis

This phase is used to prevent unfolding let-expressions when this could lead to duplicating or discarding residual code. The analysis is described in [BD91, Bon90b].

5.2.6 Redundant let-elimination analysis

This is a tidy-up phase that removes some let-expressions which have been inserted automatically by the front-end.

5.3 Postprocessing residual code

The residual code generated by the specializer can often be improved substantially by some simple last-minute optimizations; this is done by the postprocessor. Postprocessing is an integrated part of the specializer from a system point of view (this is why no separate postprocessing phase was shown in Figure 22), but it operates on residual code, not on (annotated) source code. Among other reductions, postprocessing for example post-unfolds some residual procedure calls and it post-unfolds some residual let-expressions.

6 Inspecting preprocessed/annotated programs

In Similix, partial evaluation is done by specializing a preprocessed program (see Figure 22). The specializer follows the annotations in the preprocessed program, so if partial evaluation does not give expected results, it is the annotated program which should be inspected. Annotated programs are internally represented in a (for humans) unreadable abstract syntax form, so a facility is provided to display annotated programs in a readable way. A systematic description of the facilities (called `showp`, `showpall`, `show`, and `showall`) is given in Section 8.3.3.

The information displayed can summarized as follows:

- At the definition point of any variable `V`, the binding time of `V` is displayed: `V`:*bt-value* where *bt-value* is $\perp$ (dead code or infinite loop), **S** (first-order static value), **Ps** (partially static data structure), **Cl** (higher-order static value), or **D** (value not known at specialization time, i.e. residual code). The definition points of variables are let-expressions, procedure definitions, and lambda-expressions.

- The binding times of return values of procedure definitions and lambda-expressions are shown as `->` *bt-value*.

- Every expression form is annotated as either "reducible" or "not reducible". A form is non-reducible if an underscore `_` has been added. For instance, `(_if .........)` denotes a non-reducible conditional whereas `(if .........)` denotes a reducible one.

- The new form `lift` identifies where constants are dumped in the residual code. If very large constants are accidentally dumped in some residual code when specializing a program p, you should look for occurrences of `lift` in the annotated version of p to locate where the constants origin.
- The form (memo-*name* ...) identifies specialization/memoization points (cf. Section 5.2.3). The *name* identifies the particular memoization point in the program; this may used in connection to tracing infinite loops, see under `verbose-spec` in Section 8.3.1. Also, the names of the procedures in the residual program are generated by extending the *name*-forms.

Since programs have been expanded into the Similix core language before preprocessing, the annotated programs are also in the (annotated) core language. For example, `cond`-forms will have been expanded into `if`-forms. However, two non-core forms are displayed to help the user: named (recursive) let-forms and letrec-forms. As these forms are expanded in non-local way (by lambda-lifting which moves code to a completely different place in the program), it would be quite hard to read annotated programs otherwise.

The session in Figure 24 illustrates the use of `showpall`: we inspect the annotated version of the MP-interpreter from Figure 9.

```
> (load "../system/sim-scm.scm")
;loading "../system/sim-scm.scm"

Welcome to Similix 5.0
Copyright (C) 1993 Anders Bondorf
Contributions by Olivier Danvy and Jesper Joergensen

util langext abssyn miscspec runtime front ......
#<unspecified>
> (preprocess! 'run '(s d) "MP-int.sim")
front-end flow bt sp eod oc rl
done
> (showpall)
((define (_sim-goal p:s value*:d -> d)
   (run p value*))

 (define (run p:s value*:d -> d)
   (let ((v2*:s (p->v2* p)))
     (let ((env:s (init-environment (p->v1* p) v2*)))
       (_begin
         (_init-store! value* (lift (length v2*)))
         (evalblock (p->b p) env)))))

 (define (evalblock b:s env:s -> d)
   (if (emptyblock? b)
       (lift "Finished block")
```

```
      (evalcommands (headblock b) (tailblock b) env)))
(define (evalcommands c:s b:s env:s -> d)
  (if (emptyblock? b)
      (evalcommand c env)
      (_begin
        (evalcommand c env)
        (evalcommands (headblock b) (tailblock b) env))))
(define (evalcommand c:s env:s -> d)
  (if (isassignment? c)
      (_update-store!
       (lift (lookup-env (c-assignment->v c) env))
       (evalexpression (c-assignment->e c) env))
      (if (isconditional? c)
          (memo-evalcommand-1
           (_if (_is-true?
                 (evalexpression (c-conditional->e c) env))
                (evalblock (c-conditional->b1 c) env)
                (evalblock (c-conditional->b2 c) env)))
          (if (iswhile? c)
              (memo-evalcommand-0
               (_if (_is-true? (evalexpression (c-while->e c) env))
                    (_begin
                      (evalblock (c-while->b c) env)
                      (evalcommand c env))
                    (lift "Finished loop")))
              (_err (lift 'evalcommand)
                    (lift "Unknown command: ~s")
                    (lift c))))))
(define (evalexpression e:s env:s -> d)
  (if (isquote? e)
      (lift (e->e1 e))
      (if (isvariable? e)
          (_lookup-store (lift (lookup-env e env)))
          (if (isprim? e)
              (let ((op:s (e->operator e)))
                (if (is-cons? op)
                    (_cons (evalexpression (e->e1 e) env)
                           (evalexpression (e->e2 e) env))
                    (if (is-equal? op)
                        (_eval-equal
                         (evalexpression (e->e1 e) env)
                         (evalexpression (e->e2 e) env))
                        (if (is-car? op)
                            (_car (evalexpression (e->e1 e) env))
                            (if (is-cdr? op)
```

```
                                (_cdr (evalexpression (e->e1 e) env))
                                (if (is-atom? op)
                                    (_eval-atom (evalexpression
                                                  (e->e1 e) env))
                                    (_err (lift 'evalexpression)
                                          (lift "Unknown operator: ~s")
                                          (lift op))))))))
              (_err (lift 'evalexpression)
                    (lift "Unknown expression: ~s")
                    (lift e))))))
)
```

Figure 24: Session inspecting annotated MP-interpreter

How the result of the call `(showpall)` is actually pretty-printed is Scheme-system dependent; you may have to call the pretty-printer explicitly, i.e. `(pp (showpall))`.

You may want to compare Figure 24 with Figure 12 which contained an example of a residual program obtained by specializing the MP-interpreter. Notice that the forms in Figure 12 are instances of the non-reducible forms in Figure 24.

7 How to Obtain Good Results when Using Similix

In an ideal world, we would write a source program (for instance an interpreter), partially evaluate it with respect to some static input, and then get a "good" residual program. In practice, life is more complex. Partial evaluation is no panacea: some programs specialize well, but others do not. Program generators in general take some specification as input; in the case of partial evaluation, the specification is a program. The quality of a program generated by any program generator depends on the quality of the specification. For partial evaluation, the quality of the residual program depends on the quality of the source program supplied to the partial evaluator.

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

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

In practice, many *binding-time improvements* [HH90] are needed to get good results. For example, one may convert `(+ x (+ y z))` into `(+ (+ x y) z)` when `z` is dynamic and `x` and `y` static. This section summarizes a number of well-known binding-time improvements. Some of them are of particular interest to Similix, others are more general.

7.1 Monovariancy of binding-time analysis

Binding-time analysis is *monovariant*: only one annotated version of each procedure is generated. Thus, if the same procedure is used with different binding times for the arguments, the "most dynamic" annotated version will be used in all cases. For example, one might have a program

```
(define (foo x y) (+ (bar x) (bar y)))
(define (bar z) ...)
```

with `x` being static and `y` dynamic. The binding-time analysis will then classify `z` as dynamic, and thus some possible reductions in the call `(bar x)` will be lost.

The problem can be solved by defining two versions (copies) of `bar`:

```
(define (foo x y) (+ (bar-1 x) (bar-2 y)))
(define (bar-1 z) ...)
(define (bar-2 z) ...)
```

Now `z` in `bar-1` will be static.

7.2 Some "classical" binding-time improvements

7.2.1 Static copies of dynamic data

Consider an expression

```
(if (equal? x E₁)
    (f x)
    ...)
```

with `x` dynamic and E_1 static. Since `x` is dynamic, `f` will be given a dynamic parameter. However, expression E_1 is static and we know that `x` and E_1 are equal at the point where `f` is applied. Therefore we can improve the binding times by rewriting into

```
(let ((y E₁))
  (if (equal? x y)
      (f y)
      ...))
```

Now `f` is applied to the static `y` rather than the dynamic `x`.

Negative knowledge may also be exploited. For instance, one knows that the dynamic `x` definitely does not have the value of E_1 in the false branch of the conditional above. Improvements of these kinds were of great importance in [CD89]. Some systems automate such improvements [Tur86, FN88].

7.2.2 Dynamic choice of static values

Consider an expression

```
(car (if E1 E2 E3))
```

where E_1 is dynamic, but E_2 and E_3 static. The dynamic test will make the result of the conditional expression dynamic and hence no reduction of the `car` operation will take place. This is the classical problem of a conditional with dynamic test and static branches (mentioned in for instance [Mog89]).

If the program piece is rewritten into

```
(if E1 (car E2) (car E3))
```

the `car` operations will be reduced. Notice, however, that `car` has been duplicated. Had the operation been a complex expression E rather than a simple operation such as `car`, such a duplication may not be desirable. This can be solved by using a let-expression:

```
(let ((f E)) (if E1 (f E2) (f E3)))
```

Here `f` must be a fresh variable not occurring free in any of E_1, E_2, E_3. Another kind of dynamic conditional with static branches is the following one with E_1 being dynamic:

```
((if E1 E2 E3) E)
```

If E_2 and E_3 evaluate to static function values at specialization time, no reduction of the applications will take place. Again, the problem is fixed by a simple rewriting:

```
(let ((x E)) (if E1 (E2 x) (E3 x)))
```

Now the applications `(E2 x)` and `(E3 x)` can be beta-reduced.

A more complex well-known example is the problem of dynamic indexing in a set of static values (described in for instance [Dyb85, GJ91]; the binding-time improvement is sometimes referred to as "the trick"). This problem can be illustrated by the following expression E:

```
(f (my-list-ref values index))
```

Here `index` is assumed to be dynamic and `values` is assumed to be static.

Suppose `my-list-ref` had been defined as a procedure:

```
(define (my-list-ref values index)
  (let loop ((values values) (index index))
    (cond
      ((null? values)
       (_sim-error 'my-list-ref "Whoops"))
```

```
    ((= index 0)
     (car values))
    (else
     (loop (cdr values) (- index 1))))))
```

Procedure `my-list-ref` acts just like the standard Scheme primitive `list-ref`, but we need an explicit definition to illustrate the rewriting to be performed. Since the test `(= index 0)` is dynamic, the result of the call `(my-list-ref values index)` will also be dynamic. Thus `f` is given a dynamic argument even though there is a static set of possible values `f` can be applied to (namely the elements in `values`).

We can make `f`'s argument static by rewriting `E` into

```
(my-list-ref-f values index)
```

where `my-list-ref-f` is defined by

```
(define (my-list-ref-f values index)
  (let loop ((values values) (index index))
    (cond
      ((null? values)
       (_sim-error 'my-list-ref-f "Whoops"))
      ((= index 0)
       (f (car values)))
      (else
       (loop (cdr values) (- index 1))))))
```

More generally, we may pass a function like `f` as a third parameter to `my-list-ref`. The function can then be viewed as a *continuation* `c`. Expression `E` would now take form

```
(my-list-ref-c index values f)
```

and `my-list-ref-c` would be defined by

```
(define (my-list-ref-c values index c)
  (let loop ((values values) (index index))
    (cond
      ((null? values)
       (_sim-error 'my-list-ref-c "Whoops"))
      ((= index 0)
       (c (car values)))
      (else
       (loop (cdr values) (- index 1))))))
```

Rewriting program fragments into continuation passing style is a general way to allow static information to “escape” out of a dynamic conditional expression. Introducing continuation passing style for improving binding times is discussed in [Dan91] and [HG91], and the idea is put into a more general framework in [CD91]. See also [Bon92].

7.2.3 Specialization points and dynamic choice of static values

There are two user-controlled features regarding specialization/memoization points in Similix (controlling memoization manually is quite subtle, so it cannot be recommended to the inexperienced user). There are two advantages of controlling memoization manually:

1. To obtain code sharing (less in-lining), it may be useful to manually *add* a specialization point somewhere where the automatic strategy does not insert one.
2. To speed up specialization and to obtain more in-lined residual code, it may be useful to *avoid* the automatically inserted specialization points.

1. Inserting specialization points manually is done by the primitive operator `_sim-memoize` (cf. Figure 5). For example, specializing

```
(define (f x)
  (define (g y) (+ 1 (_sim-memoize (* y y))))
  (+ (g x) (g x)))
```

with `x` being dynamic yields

```
(define (f-0 x_0)
  (define (g-0-1 y_0) (* y_0 y_0))
  (+ (+ 1 (g-0-1 x_0)) (+ 1 (g-0-1 x_0))))
```

Primitive operator `_sim-memoize` acts like the operator `generalize` from Section 3.4 in that it makes its argument dynamic. In addition to this, operator `_sim-memoize` forces insertion of a specialization point. Notice that `_sim-memoize` does not appear in the specialized program (an operator like `generalize` would appear in residual code): the specializer specifically does not generate residual `_sim-memoize` operations.

We could have specialized the above program without `_sim-memoize`:

```
(define (f x)
  (define (g y) (+ 1 (* y y)))
  (+ (g x) (g x)))
```

Specialization then gives:

```
(define (f-0 x_0)
  (+ (+ 1 (* x_0 x_0)) (+ 1 (* x_0 x_0))))
```

Notice that the *-operation occurs twice now. Thus, less code sharing is obtained as more in-lining has been performed. Sometimes code sharing is preferable, some times in-lining is preferable. It is an open research problem to design an automatic strategy that decides where to insert specialization points on the basis of an analysis of code sharing in the residual program.

2. Similix only contains a very simple mechanism for avoiding automatically inserted specialization points: automatic specialization points insertion can be switched off completely (by `standard-memoization-off`, cf. Section 8.3.2). When this is done, the *only* specialization points inserted are those specified by `_sim-memoize`! That is, there is full user-control of specialization point insertion.

The automatic strategy inserts specialization points in case of dynamic conditionals or dynamic lambdas. When the automatic strategy is switched off, dynamic choice of static values is enabled: source program rewriting is no longer necessary to obtain this. For example, no rewriting of `my-list-ref` in Section 7.2.2 is necessary to make `f`'s argument static when the automatic specialization point insertion strategy is switched off: specializing the `my-list-ref` example then gives an equally good result as specializing the `my-list-ref-f`/`my-list-ref-c` examples.

7.2.4 Eta-expansion

Consider an expression

```
(let ((f (lambda (x) ...)))
  (+ (f ...) (g f)))
```

where `g` is dynamic and hence the application `(g f)` is dynamic. The occurrence of `f` in the application `(g f)` is known as a *residual code context* in [Bon91a]: it causes the lambda-expression to become dynamic whereby no beta-reduction of the application `(f ...)` will take place either.

The problem can be solved by *eta-expansion*:

```
(let ((f (lambda (x) ...)))
  (+ (f ...) (g (lambda (w) (f w)))))
```

Now `f` no longer occurs in a residual code context due to the application `(f w)` which definitely can be beta-reduced during specialization. The new lambda-expression becomes dynamic, but that does not influence the application `(f ...)` which therefore can be reduced during specialization.

Eta-expansion can also be used in situations like those described in Section 7.1. Let us look at the program from there again:

```
(define (foo x y) (+ (bar x) (bar y)))
```

```
(define (bar z) ...)
```

Now suppose `x` and `y` are function parameters with `x` being static (binding-time value **Cl**, cf. Section 5.2.2) and `y` dynamic. The program can be rewritten into

```
(define (foo x y) (+ (bar x) (bar (lambda (w) (y w)))))
(define (bar z) ...)
```

Now both calls to `bar` have static (**Cl**) actual parameter, and thus `z` becomes static (**Cl**). Notice that no copying of the `bar` definition is necessary in this case.

In programs written in continuation passing style, the continuations will typically become static (**Cl**). However, the continuations will be built under *dynamic control*. The depth of the (partial evaluation time closures representing the) continuations will therefore not have a static bound, and thus this is a typical example of "construction of static values under dynamic control" [Jon88]. The consequence is, in case of recursion, infinite specialization.

A solution is to use the generalization primitive `generalize` (cf. Section 3.4): continuations must be forced to become dynamic. However, one is still interested in performing continuation reductions during specialization, so "most of the time" continuations should still be static. The following example illustrates how this done.

Suppose `f` is a recursive procedure defined by

```
(define (f ... c) ...)
```

where `c` is the (static) continuation. A way to generalize while keeping `c` static is to rewrite the definition into

```
(define (f ... c)
  (let ((c (collapse c))) ...))
```

where `collapse` is defined as follows:

```
(define (collapse c)
  (eta-expand-s (generalize (eta-expand-d c))))
(define (eta-expand-d c) (lambda (x) (c x)))
(define (eta-expand-s c) (lambda (x) (c x)))
```

The somewhat strange rewriting (`generalize`, `eta-expand-d`, and `eta-expand-s` all act like identity operators) ensures that the partial evaluation time closures bound to `c` during specialization never grow infinitely. Reduction of `f`'s formal parameter `c` is forced by `collapse`.

Calling `eta-expand-d` prevents `c` from occurring in the residual code context caused by `generalize`: the argument position to `generalize` is a residual code context. This eta-expansion is thus of the first kind of those described above. Calling `eta-expand-s` makes the new `c` static (**Cl**). This eta-expansion is of the second kind of those described above.

Eta-expansion has been used for binding-time improvements in a number of papers [Bon91a, BP93, Mos93]. The kind of reasoning in this section is central in the derivation of exact, one-pass continuation passing style transformers [DF89, DF91].

7.3 Some general advice on how to write source programs

7.3.1 Mixing arities

Higher-order values of different arities should not be mixed as this makes them dynamic (cf. Section 5.2.2 and Section 8.3.2). Notice that such mixings would create type errors in strongly typed languages. If you need to express that an expression evaluates to a functional value of either one arity or another arity, use user-defined constructors to wrap up the functional values. For example, when specializing the program

```
(define (f)
  ((if #f (lambda () 3) (lambda (x) x)) 8))
```

the two lambda-expressions and the application to argument `8` become non-reducible (that specialization nevertheless reduces the application is by coincidence due to postprocessing, cf. Section 5.3).

However, when specializing

```
(define (f)
  ((caseconstr (if #f (arity0 (lambda () 3)) (arity1 (lambda (x) x)))
     ((arity1 g) g))
   8))
```

where the constructors are defined by

```
(defconstr (arity0 *) (arity1 *))
```

the two lambda-expressions and the application to argument `8` become reducible.

7.3.2 Else-branches

When using the `cond`-form for conditionals, it is advisable always to write an explicit else-branch. The reason is that the default else-branch returns `#f` which may result in overly conservative binding times. For example, the program

```
(define (f)
  ((cond
     (#t (lambda (x) x)))
   4))
```

expands into the same code as the program

```
(define (f)
  ((cond
     (#t (lambda (x) x))
     (else #f))
   4))
```

When specializing this program, the lambda-expression and the application become non-reducible (that specialization nevertheless reduces the application is by coincidence due to postprocessing, cf. Section 5.3).

However, when specializing

```
(define (f)
  ((cond
     (#t (lambda (x) x))
     (else (_sim-error 'f "Blah blah blah")))
   4))
```

the lambda-expression and the application become reducible as `_sim-error` (as any other aborting primitive, cf. Section 3.4) is "binding-time neutral".

It is advisable also to write explicit else-branches for `casematch` and `caseconstr`-forms if the error message in the else-branch can be given by a primitive defined by `defprim-abort-eoi` (cf. Section 3.4). Here the reason is that if no else-branch is supplied, the default else-branches use `_sim-error` which is defined by the more conservative `defprim-abort`. This is why explicit else-branches were used in for instance the Mixwell-interpreter (Section 4.2.1, Figure 13).

7.3.3 Separation of compound tests

Consider the following expression E where E_1 is a static expression and E_2 a dynamic expression:

```
(if (and E1 E2) E3 E4)
```

If E_1 evaluates to `#f`, the test can be determined statically (at specialization time) since E_2 need not be evaluated. The entire test `(and E1 E2)` will, however, be classified as dynamic, and thus the conditional will always be considered dynamic. The problem can be solved by rewriting E into E'

```
(if E1 (if E2 E3 E4) E4)
```

But now E_4 has been duplicated. This can be avoided by abstracting out E_4:

```
(let ((g (lambda () E4))) (if E1 (if E2 E3 (g)) (g)))
```

Expression E_4 is wrapped inside a lambda to keep strictness properties unchanged. Here `g` must be a fresh variable not occurring free in E'

Tests with `or` can be rewritten in a similar way. The expression

```
(if (or E1 E2) E3 E4)
```

is thus equivalent to

```
(if E1 E3 (if E2 E3 E4))
```

Again, duplication (this time of E_3) can be avoided by abstracting out the duplicated expression:

```
(let ((g (lambda () E3))) (if E1 (g) (if E2 (g) E4)))
```

7.3.4 Introducing primitives

If a procedure returns a static result, and if all its parameters are first-order (this can be checked by using e.g. `showpall`, cf. Section 6: check if the binding times of parameters and the return value are all static, i.e. displayed with `:s/-> s`), it is beneficial to redefine the procedure to become a user-defined primitive operator. This gives faster specialization: when executing static primitive operations, compiled versions of the primitive operators are simply applied [Con88]. In contrast to this, procedures are interpreted. This is why (user-defined) primitive operators, not (user-defined) procedures, were used for syntax dispatch in the MP-interpreter (Section 4.1.1).

Also, if all operations in a procedure definition turn out to become dynamic, no reductions will take place during specialization. Such procedures may also be redefined to become user-defined primitive operators. The benefit is again faster specialization.

Binding-time improvements of these kinds are performed automatically in the partial evaluator Schism [CD90].

7.4 Termination and generalization

Specialization is not guaranteed to terminate. In fact, termination of specialization is very likely the most difficult problem one runs into sooner or later when using Similix. This section contains some hints on how to trace non-termination and how to solve the problem.

Suppose we want to specialize the program

```
(loadt "test.adt")

(define (f x)
  (let loop ((x x) (y 0))
    (if (= x 0)
        y
```

```
          (loop (- x 1) (my+ y 1)))))
```

where primitive operator `my+` is defined in file `test.adt` by

```
(defprim my+ +)
```

Operator `my+` is identical to the standard primitive `+`, except that operator `+` is defined by `defprim-tin` rather than `defprim` (cf. Section 3.4): this difference makes specialization terminate when using `+`. However, specialization does not terminate when using `my+` as above. We shall now illustrate non-termination, so we use operator `my+`.

If we specialize the above program with `x` being dynamic, specialization loops. To get an idea of why specialization loops, we can switch on tracing:

```
> (verbose-spec 1)
#<unspecified>
> (similix 'f '(***) "test.sim")
front-end flow bt sp eod oc rl
specializing
sp:_sim-goal sp:loop-0 sp:loop-0 sp:loop-0 sp:loop-0 .........
user interrupt
> (showpall)
((define (_sim-goal x:d -> d) (f x))
 (define (f x:d -> d)
   (let loop ((x:d x) (y:s 0))
     (memo-loop-0
      (_if (_= x (lift 0))
           (lift y)
           (loop (_- x (lift 1)) (my+ y 1)))))))
>
```

We notice that specialization point `loop-0` is encountered repeatedly. By inspecting the annotated program, we notice that there are two free variables in `memo-loop-0`'s argument expression, the dynamic `x` and the static `y`. Hence the problem may be that `y` assumes infinitely many static values during specialization. We can trace the values of `y` by inserting a trace operator `soutnl` in the source program:

```
(loadt "test.adt")

(define (f x)
  (let loop ((x x) (y 0))
    (if (= x 0)
        y
        (loop (- x 1) (my+ (soutnl y) 1)))))
```

Here `soutnl` is added to file `test.adt`:

```
(defprim (soutnl v) (display v) (newline) v)
```

Now we specialize again:

```
> (verbose-spec 0)
#<unspecified>
> (loadt! "test.adt")
()
> (similix 'f '(***) "test.sim")
front-end flow bt sp eod oc rl
specializing
0
1
2
3
4
⋮
user interrupt
>
```

We called `loadt!` to activate the change in file `test.adt` (the addition of the `soutnl`-definition). The definition of `soutnl` is not very clean as the primitive performs a side-effect, yet we did not define it opaque (cf. Section 3.4). But the point is exactly to get the printing done statically (at specialization time) — and opaque operations are always dynamic.

The trace confirms that `y` assumes infinitely many values. Now we use operator `generalize` (cf. Section 3.4) to force `y` to become dynamic:

```
(loadt "test.adt")

(define (f x)
  (let loop ((x x) (y 0))
    (if (= x 0)
        y
        (loop (- x 1) (my+ (generalize y) 1)))))
```

(operator `generalize` can also be defined in file `test.adt`; recall to redo the `loadt!` operation). Specialization now terminates:

```
> (similix 'f '(***) "test.sim")
front-end flow bt sp eod oc rl
specializing
((loadt "test.adt")
 (define (f-0 x_0)
   (define (loop-0-1 x_0 y_1)
     (if (= x_0 0)
         y_1
```

```
            (loop-0-1 (- x_0 1) (my+ (generalize y_1) 1))))
      (loop-0-1 x_0 0)))
>
```

We may not want `generalize` to appear in the residual code. This can be avoided by instead specializing

```
(loadt "test.adt")

(define (f x)
  (let loop ((x x) (y 0))
    (if (= x 0)
        y
        (loop (- x 1) (my+ (generalize1 y) 1)))))

(define (generalize1 z) (if #t z (generalize z))) ; hack!
```

Specialization now gives:

```
> (similix 'f '(***) "test.sim")
front-end flow bt sp eod oc rl
specializing
((loadt "test.adt")
 (loadt "gen.adt")
 (define (f-0 x_0)
   (define (loop-0-1 x_0 y_1)
     (if (= x_0 0)
         y_1
         (loop-0-1 (- x_0 1) (my+ y_1 1))))
   (loop-0-1 x_0 0)))
>
```

Infinite specialization as described here can also occur if a partially static (**Ps**) or higher-order (**Cl**) value grows infinitely. This is more difficult to trace as operation `soutnl` cannot be used to print these values out at specialization time (partially static values can of course be printed, but applying `soutnl` to such a value changes its binding time from **Ps** to **D**).

8 System Guide

8.1 Avoiding name clashes

When Similix is loaded, a number of internal Similix system names are defined at the Scheme top-level. These names are of one of the following two forms:

- Prefixed with `_sim-`.
- Prefixed with `**Similix-` and postfixed with `**`.

Do *not* define any names of these forms.

The *only* other Scheme symbols that are defined at the top-level when loading Similix are the ones described in Section 8.3.

Also, do *not* redefine standard Scheme procedures (such as `car`, `+`, etc.).

8.2 File naming conventions

Files containing programs written in the Similix Scheme subset (see Figure 3 and Figure 4) are named *file-name*`.sim`. In all of Section 8, we use ...*sim-file* to denote file names of form *file-name*`.sim`. The system automatically completes ...*sim-file* names: for ...*sim-file* names, the user may thus omit writing the `.sim` suffix explicitly.

Files with definitions of primitive operators and user-defined constructors are usually named *file-name*`.adt`. No automatic name completion is performed for `.adt` suffixes.

8.3 Similix facilities

This section describes *all* facilities (names, symbols) that are user-available (defined at the Scheme top-level) after loading the Similix system. How to load the Similix system is described in Section 2.

Three internal global Similix variables are updated and referenced by a number of the forms described in this section:

- At any time, the variable `**Similix-preprocessed-program**` contains the latest annotated program generated during the current session.
- The variable `**Similix-residual-program**` contains the latest residual program generated during the current session.
- The variable `**Similix-current-compiler**` contains the latest compiler-generator generated program generated or loaded during the current session.

These three variables need never be referenced directly by the user, but in subtle cases it is important to know when they are updated (specified in the following sections).

In the following, we use brackets [...] to denote optional arguments.

8.3.1 Specializing

`(similix)` *procedure*

Displays information about input formats to `similix`.

Returns: unspecified.

`(similix` *goal arg-pat source-sim-file* [*n*] [*resid-goal*] [*resid-sim-file* [`'pp`]]`)` *procedure*

Partially evaluates the program in file *source-sim-file* with goal procedure *goal* with respect to the input specified by *arg-pat*. The *arg-pat* is a list of pe-values, a pe-value being either the symbol `***` denoting a dynamic value or some static value (the symbol denoting dynamic input may be redefined by `(set-dynamic-input-symbol ...)`). The length of *arg-pat* must be equal to the arity of *goal*.

If n is supplied, the specialization is run n times and timing information is output. The timing figures include specialization time only, not time for preprocessing.

If *resid-goal* is supplied, the goal procedure of the residual program gets the name *resid-goal*. Otherwise, it gets the default name *goal*`-0`.

The residual program is written onto the file *resid-sim-file* if this argument is supplied. The program is pretty-printed if `'pp` is supplied.

Updates: `**Similix-preprocessed-program**` and `**Similix-residual-program**`.

Returns: `()` if *resid-sim-file* is supplied, otherwise the residual program represented as a list.

`(similix` *arg-pat* [*n*] [*resid-goal*] [*resid-sim-file* [`'pp`]]`)` *procedure*

`(similix` *arg-pat prep-pgm* [*n*] [*resid-goal*] [*resid-sim-file* [`'pp`]]`)` *procedure*

These two forms specialize an already annotated program with respect to the input specified by *arg-pat*. The forms are useful for avoiding preprocessing if the program to be partially evaluated has already been preprocessed with respect to the same binding-time pattern (no change in which parameters are static and which are dynamic).

The first of the two forms specializes the annotated program stored in `**Similix-preprocessed-program**`. The second of the two forms specializes the annotated program *prep-pgm*; here *prep-pgm* must have been generated by `(preprocessed-program)`.

For example, instead of running

```
(similix 'append1 (list '(1 2 3) '***) "append.sim")
(similix 'append1 (list '(7 6 5) '***) "append.sim")
```

we may use the first of the two forms to perform the second specialization:

```
(similix 'append1 (list '(1 2 3) '***) "append.sim")
(similix (list '(7 6 5) '***))
```

Here the *append* program is not preprocessed when the second specialization is performed. Also, instead of running

```
(similix 'append1 (list '(1 2 3) '***) "append.sim")
⋮
(similix 'append1 (list '(7 6 5) '***) "append.sim")
```

we may use the second of the two forms to perform the second specialization:

```
(similix 'append1 (list '(1 2 3) '***) "append.sim")
(define p (preprocessed-program))
```

```
⋮
(similix (list '(7 6 5) '***) p)
```

Again, the *append* program is not preprocessed when the second specialization is performed.

[Note: if several specializations are to be performed, it is beneficial to generate a generating extension first by using `cogen`. Running `(similix (list '(7 6 5) '***))` is slower than running `(comp (list '(7 6 5) '***))`.]

If n is supplied, the specialization is run n times and timing information is output. The timing figures include specialization time only, not time for preprocessing.

If *resid-goal* is supplied, the goal procedure of the residual program gets the name *resid-goal*. Otherwise, it gets the default name *goal*`-0`.

The residual program is written onto the file *resid-sim-file* if this argument is supplied. The program is pretty-printed if `'pp` is supplied.

Uses: `**Similix-preprocessed-program**` (only the first of the two forms uses this variable).

Updates: `**Similix-residual-program**`.

Returns: `()` if *resid-sim-file* is supplied, otherwise the residual program represented as a list.

`(residual-program)` *procedure*

Returns: the value of `**Similix-residual-program**`.

`(load-residual-program)` *procedure*

Loads `**Similix-residual-program**` at the top-level.

Returns: unspecified.

`(set-dynamic-input-symbol` *symbol*`)` *procedure*

Sets the symbol that is interpreted as "dynamic input" to *symbol*. The initial value is `***`.

Returns: unspecified.

`(verbose-spec` n`)` *procedure*

Argument n must `0`, `1`, or `2`. The value of n controls trace information generated during specialization. This information is particularly useful if specialization does not terminate as it may help to locate what causes the loop.

If $n = 0$, no trace information is printed; this is the initial value.

If $n = 1$, information is printed each time the specializer encounters a specialization/memoization point. The information printed is `sp:`*name* where *name* is the name of the specialization point in the source program. To locate the specialization point in the source program, use one of the facilities described in Section 6 and Section 8.3.3.

If $n = 2$, the information printed if $n = 1$ is also printed. Additionally, each time a call to a user-defined procedure P is unfolded, the name P is printed. The two forms are distinguishable as specialization point names are preceded by `sp:`.

Returns: unspecified.

`(postunfold-on)` *procedure*

Sets the post-unfold flag. Initially, the flag is set. When this flag is set, residual procedure calls are post-unfolded by the phase described in Section 5.3. When the flag is not set, residual procedure calls are never post-unfolded.

Returns: unspecified.

`(postunfold-off)` *procedure*

Clears the post-unfold flag. Initially, the flag is set.

Returns: unspecified.

8.3.2 Preprocessing

`(front-end)` *procedure*

Displays information about input formats to `front-end`.

Returns: unspecified.

`(front-end` *goal source-sim-file*`)` *procedure*

Macro expands and converts the program in file *source-sim-file* with goal procedure *goal* into internal abstract syntax.

Procedure `front-end` is typically not called directly by the user, but it may for instance be useful for debugging.

Returns: the program text represented in abstract syntax (warning: this abstract syntax may be very large, so it may be advantageous to wrap top-level calls to `front-end` into e.g. a `define`: `(define ... (front-end ... ...))`).

`(preprocess!)` *procedure*

Displays information about input formats to `preprocess!`.

Returns: unspecified.

`(preprocess!` *goal bt-pat source-sim-file*`)` *procedure*

Front-ends and preprocesses the program in file *source-sim-file* with goal procedure *goal* w.r.t. binding-time pattern *bt-pat* as described in Section 5.2. Argument *bt-pat* is a list of binding-time values; a binding-time value must be one of either `s`, `static`, `d`, `dynamic`, or the symbol denoting dynamic input (initially `***`, redefinable by `(set-dynamic-input-symbol ...)`); the forms `s` and `static` are equivalent: they specify static first-order input; the forms `d`, `dynamic`, and the symbol denoting dynamic input are also equivalent: they specify dynamic input. The length of *bt-pat* must be equal to the number of parameters to procedure *goal*.

Procedure `preprocess!` is typically not called directly by the user, but it may be useful for binding-time debugging when inspecting annotated (preprocessed) programs: the preprocessed source program can be displayed by `showp`/`showpall`.

Updates: `**Similix-preprocessed-program**`.

Returns: the symbol `done`.

`(preprocessed-program)` *procedure*

Used to save preprocessed programs for later use (see description of `(similix` *arg-pat prep-pgm* `...)`).

Returns: a list containing (1) the name of the goal procedure used when generating `**Similix-preprocessed-program**`, and (2) `**Similix-preprocessed-program**`.

`(standard-memoization-on)` *procedure*

Sets the standard memoization flag. Initially, the flag is set. When the flag is set, standard memoization points are inserted when programs are preprocessed. Standard memoization points are generated from dynamic conditionals (conditionals that are not reduced at specialization time due to a dynamic test) and dynamic lambda-expressions (lambdas that are not beta-reduced at specialization time). See Section 7.2.3 for details.

Returns: unspecified.

`(standard-memoization-off)` *procedure*

Clears the standard memoization flag. Initially, the flag is set. When the flag is not set, memoization points are only inserted when user-specified by `_sim-memoize`. This is useful as it gives the user full control of memoization point insertion; dynamic choice of static values is enabled when the flag is cleared. See Section 7.2.3 for details.

Returns: unspecified.

`(verbose-prep-on)` *procedure*

Sets the verbose preprocessing flag. Initially, the flag is set. When the flag is set, the preprocessor gives warnings when different procedure (function) arities are mixed and when constructors from different constructor families are mixed. Such mixings make more expressions dynamic (cf. Section 3.6, Section 5.2.2, and Section 7.3.1).

Returns: unspecified.

`(verbose-prep-off)` *procedure*

Clears the verbose preprocessing flag. Initially, the flag is set.

Returns: unspecified.

8.3.3 Inspecting annotated programs

`(show)` *procedure*

Displays information about input formats to `showp`, `showpall`, `show`, and `showall`.

Returns: unspecified.

`(showp` [*definitions* [*kind*]]`)` *procedure*

Used to display the latest generated preprocessed (annotated) program as described in Section 6.

Argument *definitions* must be either a list of procedure names of top-most procedures to be displayed or `'all`; if `'all` is used, all procedures are displayed.

Argument *kind* must be either `'head` or `'all`: use `'head` for displaying only information about the formal parameters and return value of the specified *definitions*; use `'all` for displaying the full definitions.

The arguments *definitions* and *kind* may be omitted in which case default values are chosen: `'all` for *definitions*, `'head` for *kind*. In practice, you will often need just `(showp)` and `(showpall)` (the latter form is described below).

Uses: `**Similix-preprocessed-program**`.

Returns: `**Similix-preprocessed-program**` pretty-printed as described in Section 6.

`(showpall)` *procedure*

Equivalent to `(showp 'all 'all)`.

Uses: `**Similix-preprocessed-program**`.

Returns: `**Similix-preprocessed-program**` pretty-printed as described in Section 6.

`(show` *prep-pgm* [*definitions* [*kind*]]`)` *procedure*

Like `showp`, but displays an arbitrary preprocessed program *prep-pgm* where *prep-pgm* must have been generated by `(preprocessed-program)`.

Returns: *prep-pgm* pretty-printed as described in Section 6.

`(showall` *prep-pgm*`)` *procedure*

Equivalent to `(show` *prep-pgm* `'all 'all)`.

Returns: *prep-pgm* pretty-printed as described in Section 6.

`(show-variable-index-on)` *procedure*

Sets the show-variable-index flag. Initially, the flag is cleared. As mentioned in Section 3.10, Similix handles `letrec`-forms by lambda-lifting which may add additional parameters to the letrec-defined procedures. This may give name clashes which, however, are resolved internally by name indices. These indices may be useful to see for the user as we shall now illustrate. Let the file `test.sim` contain the following program:

```
(define (f x)
  (letrec ((g (lambda (x) (h x)))
           (h (lambda (y) (+ x y))))
    (g 4)))
```

Lambda lifting adds an additional parameter `x` to `h` and hence to `g`:

```
> (preprocess! 'f '(s) "test.sim")
front-end flow bt sp eod oc rl
done
> (showpall)
((define (_sim-goal x:s -> d) (lift (f x)))
 (define (f x:s -> s)
   (letrec ((g (lambda (x:s x:s -> s) (h x x)))
            (h (lambda (y:s x:s -> s) (+ x y))))
     (g 4 x))))
> (show-variable-index-on)
#<unspecified>
> (showpall)
((define (_sim-goal x0:s -> d) (lift (f x0)))
 (define (f x0:s -> s)
   (letrec ((g (lambda (x1:s x0:s -> s) (h x1 x0)))
            (h (lambda (y2:s x0:s -> s) (+ x0 y2))))
     (g 4 x0))))
>
```

Returns: unspecified.

`(show-variable-index-off)` *procedure*

Clears the show-variable-index flag. Initially, the flag is cleared.

Returns: unspecified.

8.3.4 Compiler generator

`(cogen)` *procedure*

Displays information about input formats to `cogen`.

Returns: unspecified.

`(cogen` *goal bt-pat source-sim-file* [*n*] [*cmp-goal*] [*cmp-sim-file* [`'pp`]]`)` *procedure*

Generates a generating extension of the program in file *source-sim-file* with goal procedure *goal*. Argument *bt-pat* is a binding-time pattern, i.e. a list of binding-time values; a binding-time value must be one of either `s`, `static`, `d`, `dynamic`, or the symbol denoting dynamic input (initially `***`, redefinable by `(set-dynamic-input-symbol ...)`); the forms `s` and `static` are equivalent: they specify static first-order input; the forms `d`, `dynamic`, and the symbol denoting dynamic input are also equivalent: they specify dynamic input. The length of *bt-pat* must be equal to the number of parameters to procedure *goal*.

Static parameters become the "early" parameters in the generating extension, dynamic parameters become the "late" ones. The generating extension is run by using procedure `comp`.

The generation of a generating extension is done in two steps: first, the source program is preprocessed with respect to the given *bt-pat* (see the description of procedure `preprocess!`), then the Similix-generated compiler generator is applied to the preprocessed source program.

A typical application of `cogen` is to generate a compiler from an interpreter. The interpreter's program parameter is classified as `static`, the data parameter is classified as `dynamic`.

If *n* is supplied, the compiler generator is applied *n* times to the preprocessed program and timing information is output. The timing figures do not include the time used for preprocessing the source program.

If *cmp-goal* is supplied, the goal procedure of the generated generating extension gets the name *cmp-goal*. Otherwise, it gets the default name `_sim-specialize-0`.

The generating extension is written onto the file *cmp-sim-file* if this argument is supplied. The program is pretty-printed if `'pp` is supplied.

Updates: `**Similix-preprocessed-program**` and `**Similix-current-compiler**`.

Returns: `()`.

`(cogen` [*n*] [*cmp-goal*] [*cmp-sim-file* [`'pp`]]`)` *procedure*

`(cogen` *prep-pgm* [*n*] [*cmp-goal*] [*cmp-sim-file* [`'pp`]]`)` *procedure*

These two forms curry an already annotated program. The forms are useful for avoiding preprocessing if the program to be curried has already been preprocessed.

The first of the two forms curries the annotated program stored in `**Similix-preprocessed-program**`. The second of the two forms curries the annotated program *prep-pgm*; here *prep-pgm* must have been generated by `(preprocessed-program)`.

If *n* is supplied, the compiler generator is applied *n* times to the preprocessed program and timing information is output. The timing figures do not include the time used for preprocessing the source program.

If *cmp-goal* is supplied, the goal procedure of the generated generating extension gets the name *cmp-goal*. Otherwise, it gets the default name `_sim-specialize-0`.

The generating extension is written onto the file *cmp-sim-file* if this argument is supplied. The program is pretty-printed if `'pp` is supplied.

Uses: `**Similix-preprocessed-program**` (only the first of the two forms uses this variable).

Updates: `**Similix-current-compiler**`.

Returns: `()`.

`(comp)` *procedure*

Displays information about input formats to `comp`.

Returns: unspecified.

`(comp` [*cmp-goal*] [*cmp-file*] *arg-pat* [*n*] [*resid-goal*] [*resid-sim-file* [`'pp`]]`)` *procedure*

Applies a generating extension generated by `cogen` to *arg-pat*. The length of *arg-pat* must be equal to the length of the *bt-pat* that was supplied to `cogen` when generating the generating extension. For those arguments that were specified as static in *bt-pat* when running `cogen`, supply a value in *arg-pat*. For those arguments that were specified as dynamic in *bt-pat* when running `cogen`, supply the symbol `***`.

A typical application of `comp` is to run a compiler generated by applying `cogen` to an interpreter.

If *cmp-goal* is supplied, the goal procedure of the generating extension is assumed to have this name (this name must be equal to the *cmp-goal* specified when generating the generating extension by `cogen`). Otherwise, the default name `_sim-specialize-0` is chosen.

If *cmp-file* is supplied, the generating extension is read from this file. Otherwise, the program in `**Similix-current-compiler**` is used.

If *n* is supplied, the generating extension is applied *n* times and timing information is output.

If *resid-goal* is supplied, the goal procedure of the residual program gets the name *resid-goal*. Otherwise, it gets the default name *goal*`-0` where *goal* is the goal name of the source program that was specified when generating the generating extension.

The residual program is written onto the file *resid-sim-file* if this argument is supplied. The program is pretty-printed if `'pp` is supplied.

Uses: `**Similix-current-compiler**` unless *cmp-file* is supplied.

Updates: `**Similix-current-compiler**` and `**Similix-residual-program**`.

Returns: `()` if *resid-sim-file* is supplied, otherwise the residual program represented as a list.

`(current-compiler)` *procedure*

Returns: the value of `**Similix-current-compiler**`.

8.3.5 Utilities for Similix source files

`(compile-sim-file` *sim-file*`)` *procedure*

Compiles *sim-file*.

Returns: `()`.

`(compile-and-load-sim-file` *sim-file*`)` *procedure*

Compiles and loads *sim-file*.

Returns: `()`.

`(loads` *sim-file*`)` *procedure*

Loads *sim-file*. This form should be used at the top-level instead of `load` if *sim-file* contains `casematch`- or `caseconstr`-forms (`load` does not know these forms).

Returns: unspecified.

`(loadt` *file*`)` *procedure*

Loads a file of definitions of primitive operators and constructors following the syntax of Figure 7. The form is typically only used in Similix Scheme programs (cf. Figure 3), but it may be used at the top-level. The form `loadt` side-effects a global system variable which contains compiled versions of the primitive operators and constructors. This prevents recompilation if the same file is loadt'ed more than once in a session. See also `loadt!`.

Returns: `()`.

`(loadt!` *file*`)` *procedure*

Equivalent to

```
(begin (unloadt file) (loadt file))
```

That is, recompilation and reloading of the primitive operators and constructors is enforced. If, during a session, a file *file* with primitive operator and constructor definitions is modified, *always follow the modifications by executing*

```
(loadt! file)
```

Otherwise, the modifications will not come into effect during the session. You must redo the `loadt!` for all possible full file names (with paths) that are used to refer to *file* (this may be relevant if you are specializing programs from different directories that all use *file*).

Returns: `()`.

`(unloadt` *file*`)` *procedure*

Removes the compiled versions of the primitive operators defined in *file* from the global variable updated by `loadt`. Typically only used indirectly through `loadt!`.

Returns: `unloadt-ed`.

`(sim2scheme` *sim-file*`)` *procedure*

Converts Similix Scheme programs into stand-alone Scheme programs which can be run without loading Similix first. All definitions in files loaded and loadt'ed by *sim-file* are in-lined and primitive operator and constructor definitions are converted to ordinary Scheme definitions. The output is written on the file *file-name*`.`*postfix* where *file-name* is equal to *sim-file* without possible `.sim` suffix and *postfix* is the standard postfix used for source files in the Scheme system used.

Returns: unspecified.

8.3.6 Resetting Similix

`(reset-similix)` *procedure*

Resets flags and other global variables used by Similix. Useful for resetting flags and for freeing heap space.

Returns: `()`.

8.3.7 Help-facility

`(help)` *procedure*

Prints brief overview of procedures available in Similix.

Returns: unspecified.

8.3.8 General Scheme utilities

`(file->item` *file*`)` *procedure*

Returns: the first object in *file*.

`(file->list` *file*`)` *procedure*

Returns: a list of the objects in *file*.

`(ntimes` *suspension n*`)` *procedure*

Applies *suspension* ("thunk") *n* times and prints timing information. For example, `(ntimes (lambda () (+ 1 3)) 100)` computes `(+ 1 3)` 100 times and prints timing information.

Returns: the value of `(`*suspension*`)`.

`(out` *e*`)` *procedure*

Identity procedure that displays the value of its argument. Useful for debugging.

Returns: the value of argument *e*.

`(outnl` *e*`)` *procedure*

Similar to `out`, but also displays a "newline".

Returns: the value of argument *e*.

`(outpp` *e*`)` *procedure*

Similar to `out`, but pretty-prints the value.

Returns: the value of argument *e*.

`(pp` *e*`)` *procedure*

Invokes the pretty-printer.

Returns: unspecified.

`(size` *e*`)` *procedure*

Returns: the size of the argument measured as its number of "cons" cells plus its number of vector elements (recursively).

`(writef` *e file*`)` *procedure*

Writes the value of expression *e* onto *file*.

Returns: unspecified.

`(writefpp` *e file*`)` *procedure*

Similar to `writef`, but pretty-prints the value.

Returns: unspecified.

`(writel` *l file*`)` *procedure*

Expression *l* must evaluate to a list. The form `writel` writes the elements of the value of *l* onto *file*, stripping off the outer parentheses of the value of *l*.

Returns: unspecified.

`(writelpp` *l file*`)` *procedure*

Similar to `writel`, but pretty-prints each element.

Returns: unspecified.

References

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[BEJ88] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*, IFIP TC2, North-Holland, 1988. Workshop proceedings, October 1987, Gl. Avernæs, Denmark.

[BJ93a] Anders Bondorf and Jesper Jørgensen. *Efficient analyses for realistic off-line partial evaluation: extended version.* Technical Report 93/4, DIKU, University of Copenhagen, Denmark, 1993.

[BJ93b] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on partial evaluation*, 1993.

[Bon90a] Anders Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, March 1990.

[Bon90b] Anders Bondorf. *Self-Applicable Partial Evaluation (Revised Version).* PhD thesis, DIKU, University of Copenhagen, Denmark, December 1990. DIKU report 90/17.

[Bon91a] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming,*, 17(1-3):3–34, December 1991. Revision of paper in ESOP'90, LNCS 432, May 1990.

[Bon91b] Anders Bondorf. Compiling laziness by partial evaluation. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 9–22, Springer-Verlag, August 1991.

[Bon91c] Anders Bondorf. *Similix Manual, system version 4.0.* DIKU, University of Copenhagen, Denmark, September 1991. Included in Similix distribution.

[Bon92] Anders Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California. LISP Pointers V, 1*, pages 1–10, June 1992.

[BP93] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In *Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, June 1993.

[CD89] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

[CD90] Charles Consel and Olivier Danvy. From interpreting to compiling binding times. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark. Lecture Notes in Computer Science 432*, pages 88–105, Springer-Verlag, May 1990.

[CD91] Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts. Lecture Notes in Computer Science 523*, pages 495–519, Springer-Verlag, August 1991.

[Con88] Charles Consel. New insights into partial evaluation: the SCHISM experiment. In Harald Ganzinger, editor, *ESOP'88, 2nd European Symposium on Programming, Nancy, France. Lecture Notes in Computer Science 300*, pages 236–247, Springer-Verlag, March 1988.

[CR91] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. November 1991.

[Dan91] Olivier Danvy. Semantics-directed compilation of nonlinear patterns. *Information Processing Letters*, 37(6):315–322, 1991.

[DF89] Olivier Danvy and Andrzej Filinski. *A functional abstraction of typed contexts.* Technical Report 89/12, DIKU, University of Copenhagen, Denmark, 1989.

[DF91] Olivier Danvy and Andrzej Filinski. *Representing control.* Technical Report CS-91-2, Kansas State University, 1991.

[Dyb85] Hans Dybkjær. *Parsers and partial evaluation: an experiment.* Student Report 85-7-15, DIKU, University of Copenhagen, Denmark, July 1985.

[Ers78] Andrei P. Ershov. On the essence of compilation. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420, North-Holland, 1978.

[Ers82] Andrei P. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[FN88] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151, North-Holland, 1988.

[GJ91] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.

[HG91] Carsten Kehler Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut. SIGPLAN Notices, volume 26, 9*, pages 223–233, ACM Press, June 1991.

[HH90] Carsten Kehler Holst and John Hughes. Towards binding-time improvement for free. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 83–100, Springer-Verlag, August 1990.

[IEE90] IEEE standard for the Scheme programming language. May 1990. IEEE Std 1178-1990.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[Joh85] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Conference on Functional Languages and Computer Architecture, Nancy, France. Lecture Notes in Computer Science 201*, pages 190–203, Springer-Verlag, September 1985.

[Jon88] Neil D. Jones. Automatic program specialization: a re-examination from basic principles. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282, North-Holland, 1988.

[JSS85] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202*, pages 124–140, Springer-Verlag, 1985.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[Mog89] Torben Æ. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, March 1989.

[Mos93] Christian Mossin. Partial evaluation of general parsers (extended abstract). In David Schmidt, editor, *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark*, June 1993.

[Ses85] Peter Sestoft. *The structure of a self-applicable partial evaluator*. Technical Report 85/11, DIKU, University of Copenhagen, Denmark, November 1985.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

Index