The PGG System—User Manual

Peter Thiemann*
Universität Freiburg
thiemann@informatik.uni-freiburg.de

December 30, 2008

1 Introduction

The PGG system is a partial evaluation system for the full Scheme language as defined in the R5RS report [19]. It has the following features

- offline partial evaluation using the cogen approach;
- correct specialization of imperative code;
- side effects performed at specialization time;
- modular specialization;
- no restrictions on primitives and static inputs (they are not restricted to have first-order types);
- handles `eval`, `apply`, `call-with-values`, and control operators correctly;
- flexible control of memoization;
- language extensions (user-defined algebraic datatypes, `make-cell`, `cell-set!`, `cell-ref`, `cell-eq?`);
- representation analysis;
- fast specialization (the system produces generating extensions);
- multi-level specialization.

*Copyright © Peter Thiemann, 1998-2008

The system does not have a post-processor. This manual does not contain explanatory material about offline partial evaluation. Section 7.1 gives some pointers to relevant literature.

Contents

1 Introduction **1**

2 Installation **5**

3 First Steps **6**
- 3.1 Power . . . 6
- 3.2 Lambda interpreter . . . 9
- 3.3 Cyclic . . . 12
- 3.4 Guide to the other examples . . . 14
- 3.5 Specialization of modular programs . . . 17
- 3.6 Specialization with respect to indexed data . . . 18

4 Reference manual **22**
- 4.1 Notation . . . 22
- 4.2 Type system . . . 22
- 4.3 Binding-time analysis . . . 22
- 4.4 Primitive operations . . . 23
- 4.5 Representation analysis . . . 23
- 4.6 Memoization . . . 24
- 4.7 Special expressions . . . 24
  - 4.7.1 eval . . . 24
  - 4.7.2 apply . . . 25
  - 4.7.3 lambda-poly . . . 25
- 4.8 Predefined operators . . . 25
- 4.9 Directives . . . 26
  - 4.9.1 define-without-memoization . . . 26
  - 4.9.2 define-data . . . 26
  - 4.9.3 define-type . . . 27
  - 4.9.4 define-primitive . . . 27
  - 4.9.5 define-memo . . . 28
  - 4.9.6 load . . . 29
  - 4.9.7 begin . . . 29
- 4.10 Commands . . . 29
  - 4.10.1 Creating a generating extension . . . 29
  - 4.10.2 Running a generating extension . . . 30
  - 4.10.3 Continuing a specialization . . . 32
  - 4.10.4 Suspend a deferred specialization . . . 32
  - 4.10.5 Resurrect a deferred specialization . . . 32
- 4.11 Settable options . . . 32
- 4.12 Utilities . . . 34

5 Differences to Scheme **34**

6 Reading a generating extension **34**

7 Technical background **36**
7.1 Partial evaluation in general 36
7.2 Directly related publications 37
7.3 Structure of the implementation 37

8 Known problems **38**

2 Installation

To use the system, you first have to install the Scheme48 system [20], which is available from Richard Kelsey's homepage `http://www.s48.org/`. The current version 1.8 is required to run PGG.

Once you have installed Scheme48, unpack the distribution file by

```
kailua> mkdir pgg-1.4
kailua> cd pgg-1.4
kailua> zcat ⟨path-where-you-downloaded⟩/pgg-1.4.tar.gz | tar xvf -
```

(with `kailua>` being the shell's prompt) This creates the directory `pgg-1.4` in the current directory.

Next, you should build yourself an image file of the system, to speed up loading later on. To do this type:

```
kailua> cd pgg-1.4
kailua> make
(echo ",bench on"; \
 echo ",config,load genext-packages.scm pgg-packages.scm"; \
 for package in pgg-residual pgg ; do \
 echo ",load-package $package"; \
 done ; \
 echo ",open pgg signals"; \
 echo ",open auxiliary pgg-library pgg-specialize pp"; \
 echo ",collect"; \
 echo ",dump pgg.image \"(PGG-1.2 made by $LOGNAME ‘date‘)\""; \
 echo ",exit" ) \
| scheme48 -h 10000000
Welcome to Scheme 48 1.8 (made by thiemann on Fri Aug  8 16:50:56 CEST 2008).
Copyright (c) 1993-2008 by Richard Kelsey and Jonathan Rees.
Please report bugs to scheme-48-bugs@s48.org.
Type ,? (comma question-mark) for help.
> will compile some calls in line
> > > > > > Before: 2527559 words free in semispace
After:  4203309 words free in semispace
> Writing pgg.image
>
kailua>
```

Next time you want to use PGG, type

```
kailua> scheme48 -h 6000000 -i pgg.image
```

to save the time spent with loading and compiling the system. You might want to put the above into a shell script. The `-h` parameter determines the heapsize which might have to be increased when dealing with larger programs. The `pgg.image` file may be moved to an arbitrary location, it is independent of the directory containing the PGG distribution.

3 First Steps

This section goes through a few examples of using PGG. It assumes that the system has been started in the `pgg-1.1` directory. The subdirectory `example` contains the sources of all examples.

3.1 Power

One of the simplest examples is the exponentiation function `power`. It resides in file `examples/power.scm`.

```
(define (power x n)
  (if (= 0 n)
      1
      (* x (power x (- n 1)))))
```

To specialize it, PGG must know three things

- where to find the source program;
- the name of the entry point;
- the binding times of the parameters of the entry point.

The latter two are specified using a binding-time skeleton, i.e., a list that contains the entry point and the binding times of the parameters. In the example, `'(power 1 0)` is a sensible binding-time skeleton. It specifies the entry point `power` and the binding times `1` (dynamic) for the base `x` and `0` (static) for the exponent `n`.

```
> (cogen-driver (list "examples/power.scm") '(power 1 0))
bta-run
bta-solve
bta-solve done
'((define (specialize-$goal x2) ...
>
```

PGG's answer is the corresponding generating extension. Pretty printing yields:

```
(define (specialize-$goal x2)
  (specialize $goal '(power 1 0) (list 'x1 x2)))
(define (power x_1 n_1)
  (if (_op 0 = 0 n_1)
      (_lift 0 1 1)
      (_op 1 * x_1 (power x_1 (_op 0 - n_1 1)))))
(define ($goal x_1 n_1)
  (power x_1 n_1))
```

To use the generating extension, we need to compile it. There are several ways to do that:

```
> (define genext
    (cogen-driver (list "examples/power.scm") '(power 1 0)))
bta-run
bta-solve
bta-solve done
; no values returned
> (load-program genext)
; no values returned
>
```

Alternatively, we can first save the generating extension to a file and then load and compile the file.

```
> (writelpp genext "/tmp/power-10.scm")
#{Unspecific}
> (load "/tmp/power-10.scm")
/tmp/power-10.scm
; no values returned
>
```

The latter approach is recommended if the source program does not yet specialize satisfactorily. In this case, inspection of the generating extension reveals possible problems. For this reason, the syntax of the generating extension is as close as possible to binding-time annotated Scheme.

Now that we have loaded the generating extension, we are ready to specialize. This is facilitated by the `specialize-$goal` function provided as part of the generating extension.

```
> (specialize-$goal 0)
'(power-2 x1)
> (get-residual-program)
'((define (power-2 x-3) 1))
>
```

The specializer responds with the call template for the residual program, `'(power-2 x1)`, indicating that `power-2` is the entry point of the residual program and that it takes one parameter. The specializer puts the residual program in a variable whose contents can be retrieved with the `get-residual-program` procedure, for further examination, compilation, or to save it to a file.

Here is a more interesting run, specializing `power` for `n=4`.

```
> (specialize-$goal 4)
'(power-2 x1)
> (p (get-residual-program))
((define (power-2 x-3)
   (let* ((mlet-11 (* x-3 1))
          (mlet-9 (* x-3 mlet-11))
          (mlet-7 (* x-3 mlet-9)))
     (* x-3 mlet-7))))
```

(The function `p` invokes the pretty printer.)

This residual program looks more complicated than we expected. The reason is that PGG—by default—avoids to duplicate or to reorder residual code. This feature makes it easy to have impure (side-effecting) primitives. In the present case, we know that `*` is pure and that no code duplication arises from it. An appropriate declaration,

```
(define-primitive * - pure)
```

as provided in the file `"examples/pure-arith.scm"`, instructs PGG that `*` is indeed a pure function. Now we can say

```
> (define genext
    (cogen-driver (list "examples/power.scm"
                        "examples/pure-arith.scm") '(power 1 0)))
...
> (load-program genext)
; no values returned
> (specialize-$goal 4)
```

and PGG generates the expected code:

```
(define (power-2 x-3)
  (* x-3 (* x-3 (* x-3 (* x-3 1)))))
```

A post-processor would have reduced the expression `(* x-1 1)` to `x-1`. This example demonstrates that there is none. It is nevertheless possible to obtain the same effect by slightly rewriting the source program. This is left as as exercise.

3.2 Lambda interpreter

This section shows a classic example, an interpreter for an applied lambda calculus with Scheme's constants, a conditional, and primitive operations. The input to the interpreter is a lambda expression, a list of free variables, and a list of values of the free variables. The following grammar specifies the concrete syntax of expressions.

```
E ::= X | (lambda (X) E) | (apply E E)
   |  C | (if E E E) | (O E*)
```

This interpreter employs *partially static data* to represent the environment. The environment is a list of pairs of variable name and value. The intention is that the length of the list and all variable names are static, but the values are dynamic. Traditionally[1], the Scheme built-in lists cannot be used for this, so we define a new algebraic datatype for this purpose.

```
(define-data my-list (my-nil) (my-cons my-car my-cdr))
```

This line declares the algebraic datatype `my-list` with constructors `my-nil` and `my-cons` (see 4.9.2). The elements of this datatype may be *partially static*, i.e., the components may have a different (higher) binding time than the structure itself. In addition, they can be memoized separately.

It is a little tedious to enter such an environment by hand, so we also supply a function that transforms a static list of names and a dynamic list of values into an environment. Finally, it calls the interpreter function `int`.

```
(define (main exp names values)
  (let loop ((names names) (values values) (env (my-nil)))
    (if (null? names)
        (int exp env)
        (loop (cdr names) (cdr values)
              (my-cons (my-cons (car names) (car values)) env)))))
```

The interpreter has two local functions, `int*` and `apply-prim`. `Int*` evaluates a list of expressions to a list of values. `Apply-prim` takes a primitive operator and a list of value and returns the result. The interesting part of `apply-prim` is its use of `eval`. `Eval`'s argument `op` is static, whereas the result of `eval` is dynamic.

```
(define (int exp env)
  (let loop ((exp exp))
    (define (int* exp*)
      (let recur ((exp* exp*))
```

[1] In partial evaluation, that is.

```
      (if (null? exp*)
          '()
          (cons (loop (car exp*))
                (recur (cdr exp*))))))
  (define (apply-prim op args)
    (apply (eval op (interaction-environment))
           args))
  (cond
   ((constant? exp)
    exp)
   ((not (pair? exp))
    (lookup exp env))
   ((eq? (car exp) 'IF)
    (let ((test-exp (cadr exp))
          (then-exp (caddr exp))
          (else-exp (cadddr exp)))
      (if (loop test-exp)
          (loop then-exp)
          (loop else-exp))))
   ((eq? (car exp) 'LAMBDA)
    (lambda (y)
      (int (caddr exp) (my-cons (my-cons (caadr exp) y) env))))
   ((eq? (car exp) 'APPLY)
    ((loop (cadr exp))
     (loop (caddr exp))))
   (else
    (apply-prim (car exp) (int* (cdr exp))))))
```

All that's missing are two auxiliary functions, `constant?` and `lookup`, that indicate whether an expression denotes a constant and perform lookup in the environment.

```
(define (constant? e)
  (or (boolean? e)
      (number? e)
      (and (pair? e) (eq? (car e) 'QUOTE))))

(define (lookup v env)
  (let loop ((env env))
    (if (eq? v (my-car (my-car env)))
        (my-cdr (my-car env))
        (loop (my-cdr env)))))
```

As already mentioned, the idea is that the inputs `exp` and `names` are static and that `values` is dynamic. So we start the binding-time analysis with

```
> (define genext
```

```
    (cogen-driver (list "examples/int.scm") '(main 0 0 1)))
bta-run
bta-solve
bta-solve done
; no values returned
>
```

To load this generating extension, we need to load the `define-data` operation from module `pgg-residual`.

```
> ,open pgg-residual
> (load-program genext)
> (specialize-$goal 5 '())
'(main-2 x3)
> (p (get-residual-program))
((define (main-2 x-3) 5))
> (specialize-$goal '(+ x y) '(x y))
'(main-2 x3)
> (p (get-residual-program))
((define (main-2 x-3)
   (let* ((mlet-5 (cdr x-3))
          (mlet-7 (car x-3))
          (mlet-9 (cdr mlet-5))
          (mlet-11 (car mlet-5)))
     (+ mlet-7 mlet-11))))
> (specialize-$goal '(lambda (x) (+ x y)) '(y))
'(main-2 x3)
> (p (get-residual-program))
((define (main-2 x-3)
   (define (loop-4 mlet-3)
     (lambda (y_1-5)
       (+ y_1-5 mlet-3)))
   (let* ((mlet-5 (cdr x-3)) (mlet-7 (car x-3)))
     (loop-4 mlet-7))))
>
```

The examples demonstrate that the environment is specialized away. Only the dynamic values survive and become parameters (this is called "arity raising"). Furthermore, `eval` and `apply` have been specialized satisfactorily, as demonstrated by the last two specializations: `(+ mlet-7 mlet-11)` and `(+ y_1-5 mlet-3)` is the corresponding residual code.

The auxiliary definition of `loop-4` is introduced automatically by the specializer to avoid a non-terminating specialization. In the example, there is no danger of non-termination because the recursive calls only decompose the source expression. Hence, it is safe to turn off memoization for the function `int` by changing the first line of its definition to

```
(define-without-memoization (int exp env)
  ...)
```

After constructing a new generating extension, we obtain a simpler residual program.

```
(define ($goal-1 values-1)
  (let* ((mlet-2 (cdr values-1))
         (mlet-3 (car values-1)))
    (lambda (y_1-4) (+ y_1-4 mlet-3))))
```

3.3 Cyclic

This example demonstrates specialization of imperative programs.

```
(define-data my-list (my-nil) (my-cons my-car my-cdr))
(define (main d)
  (let ((cycle (my-cons 1 (make-cell (my-nil)))))
    (cell-set! (my-cdr cycle) cycle)
    (zip d cycle)))
(define (zip d s)
  (if (null? d)
      '()
      (cons (cons (car d) (my-car s))
            (zip (cdr d) (cell-ref (my-cdr s))))))
```

The list `cycle` is completely static, but the cdr of `cycle` contains a reference to cycle itself. This cyclic list of ones is passed as an argument to the function `zip` which zips it together with a dynamic list `d`. Unrolling the dynamic list involves memoization, hence the specializer must memoize the cyclic structure passed as an argument to `zip` to avoid infinite specialization. Here is what happens.

```
> (define genext
    (cogen-driver (list "examples/cyclic.scm") '(main 1)))
bta-run
effect analysis: fixpointing done
bta-solve
bta-solve done
> (p genext)
((define-data my-list (my-nil) (my-cons my-car my-cdr))
 (define (specialize-$goal)
   (specialize $goal '(main 1) (list 'x1)))
 (define (main d_2)
   (let ((cycle_1 (_ctor_memo 0
                              (0 0)
```

```
                              #f
                              my-cons
                              1
                              (_make-cell_memo 0
                                               3
                                               0
                                               (_ctor_memo 0
                                                           ()
                                                           #f
                                                           my-nil)))))
    (_message!_memo 0 (_s_t_memo 0 my-cdr cycle_1) cell-set! cycle_1)
    (zip d_2 cycle_1)))
 (define (zip d_1 s_1)
   (multi-memo 1 1 'zip-2 zip-2 #f '(1 0) (list d_1 s_1)))
 (define (zip-2 d_1 s_1)
   (_if 1
        (_op 1 null? d_1)
        (_lift 0 1 '())
        (_op 1
             cons
             (_op 1 cons (_op 1 car d_1) (_lift 0 1 (_s_t_memo 0 my-car s_1)))
             (zip (_op 1 cdr d_1)
                  (_s_t_memo 0 cell-ref (_s_t_memo 0 my-cdr s_1))))))
 (define ($goal d_2)
   (main d_2)))
>
```

The function `_ctor_memo` constructs the memoized representation of a constructor. Its first argument is the binding time of the structure itself, its second argument is the list of binding times of the components (all `0` in this case). `_make-cell_memo` constructs a memoized reference cell, the first argument is the binding time of the address and the next argument `3` is the unique label of the corresponding `make-cell` operation in the source program. `_s_t_memo` accesses or tests memoized data objects, the implementation handles them all uniformly.

The operation `_define-data` serves to transfer the datatype definition to the residual program.

To load this generating extension, we need to make the `define-data` operation available.

```
> ,open pgg-residual
Load structure pgg-residual (y/n)? y
[pgg-residual
cogen-ctors.scm ]
```

```
Newly accessible in user: (define-data)
> (load-program genext)
> (specialize-$goal)
'(main-2 x1)
> (p (get-residual-program))
((define (main-2 x-3)
   (define (zip-4 x-3)
     (let ((mlet-5 (null? x-3)))
       (if mlet-5
           '()
           (let* ((mlet-11 (car x-3))
                  (mlet-9 (cons mlet-11 1))
                  (mlet-13 (cdr x-3))
                  (mlet-15 (zip-4 mlet-13)))
             (cons mlet-9 mlet-15)))))
   (zip-4 x-3)))
```

The cyclic structure vanishes on specialization. The construction of the pair `(x . 1)` is implemented by `(cons mlet-11 1)`.

3.4 Guide to the other examples

- `examples/2lazy.scm` a two-level interpreter for a lazy first-order language, implements updatable closures using references.

  ```
  > (define genext
      (cogen-driver (list "examples/2lazy.scm") '(lazy-2int 0 0 0 1)))
  > (load-program genext)
  ```

  The parameters of `(lazy-2int prg goal xs* xd*)` are

  - `prg` the program;
  - `goal` the entry point of `prg` (a symbol);
  - `xs*` the static parameters;
  - `xd*` the dynamic parameters.

  The static parameters may include *configuration variables* of the form `(CV` i`)` which refers to the ith dynamic parameter.

  To perform specialization, we need to load some auxiliary functions

  ```
  (load "examples/2lazy-support.scm")
  ```

  It contains the example programs `lazy1` and `lazy2`. Example calls of the specializer include

```
> (specialize $goal '($goal 0 0 0 1) (list lazy1 'f '(42) 'DYN))
> (specialize $goal '($goal 0 0 0 1) (list lazy1 'f '((CV 1)) 'DYN))
> (specialize $goal '($goal 0 0 0 1) (list lazy2 'f '((CV 1) (CV 2) (CV 3)) 'DYN))
> (specialize $goal '($goal 0 0 0 1) (list lazy2 'f '((CV 1) (CV 2) 13) 'DYN))
> (specialize $goal '($goal 0 0 0 1) (list lazy2 'f '((CV 1) 7 11) 'DYN))
> (specialize $goal '($goal 0 0 0 1) (list lazy2 'f '(#t (CV 1) (CV 2)) 'DYN))
> (specialize $goal '($goal 0 0 0 1) (list lazy2 'f '(#f (CV 1) (CV 2)) 'DYN))
> (specialize $goal '($goal 0 0 0 1) (list lazy2 'f '(#f (CV 1) 17) 'DYN))
```

- `examples/app.scm` contains the append function for lists.
- `examples/dotprod.scm` compute the scalar product of three vectors. This is an example for multi-level specialization.

```
> (define genext (cogen-driver (list "examples/dotprod.scm")
                               '(dotprod 0 1 2 3)))
> (load-program genext)
> (specialize-$goal 2)
'(multi-memo 2 2 'dotprod-2 dotprod-2 #f '(0 1 2) (list x2 x3 x4))
```

This answer indicates that the residual program is again a generating extension, which can be loaded and specialized further. Let's have a look

```
> (p (get-residual-program))
((define (dotprod-2 x-7 x-5 x-3)
   (let* ((mlet-15 (car x-7))
          (mlet-17 (_op 1 car x-5))
          (mlet-13 (_op 1 * (_lift0 1 mlet-15) mlet-17))
          (mlet-19 (_op 2 car x-3))
          (mlet-11 (_op 2 * (_lift 1 1 mlet-13) mlet-19))
          (mlet-21 (cdr x-7))
          (mlet-23 (_op 1 cdr x-5))
          (mlet-25 (_op 2 cdr x-3))
          (mlet-33 (car mlet-21))
          (mlet-35 (_op 1 car mlet-23))
          (mlet-31 (_op 1 * (_lift0 1 mlet-33) mlet-35))
          (mlet-37 (_op 2 car mlet-25))
          (mlet-29 (_op 2 * (_lift 1 1 mlet-31) mlet-37))
          (mlet-39 (cdr mlet-21))
          (mlet-41 (_op 1 cdr mlet-23))
          (mlet-43 (_op 2 cdr mlet-25))
          (mlet-27 (_op 2 + mlet-29 (_lift0 2 0))))
     (_op 2 + mlet-11 mlet-27))))
```

This time, we have to *load the residual program* to continue specializing. The answer from the previous specialization step tells us the name `dotprod-2` of the entry point.

```
> (load-program (get-residual-program))
> (specialize dotprod-2 '(dotprod-2-1 0 1 2) '((111 222) v2 v3))
'(multi-memo 1 1 'dotprod-2-1 dotprod-2-1 #f '(0 1) (list v2 v3))
> (load-program (get-residual-program))
> (specialize dotprod-2-1 '(dotprod-2-1-1 0 1) '((333 444) v3))
'(dotprod-2-1-1 v3)
> (p (get-residual-program))
((define (dotprod-2-1-1 v-3)
   (let* ((mlet-5 (car v-3))
          (mlet-7 (* 36963 mlet-5))
          (mlet-9 (cdr v-3))
          (mlet-11 (car mlet-9))
          (mlet-13 (* 98568 mlet-11))
          (mlet-15 (cdr mlet-9))
          (mlet-17 (+ mlet-13 0)))
     (+ mlet-7 mlet-17))))
```

This is the final specialized program after three steps.

- `object` a class of counter objects. A mini-example with state.

```
> (define genext (cogen-driver (list "examples/object.scm") '(main)))
> (load-program genext)
> (specialize-$goal)
```

- `pm` Olivier Danvy's pattern matcher [7]

```
> (define genext (cogen-driver (list "examples/pm.scm") '(match 0 1)))
```

- `unify` imperative unification of terms where variables are implemented by references.

```
> (define genext (cogen-driver (list "examples/unify.scm") '(main 0 1)))
> (load-program genext)
> (specialize-$goal '(cst 555))
> (specialize-$goal '(var 555))
> (specialize-$goal '(bin (var 1) (var 1)))
> (specialize-$goal '(bin (var 1) (bin (cst 4711) (var 1))))
```

3.5 Specialization of modular programs

As an advanced feature, it is possible to encapsulate the generating extension in a module. We recap the example of the `power` function to illustrate it. In addition to the usual parameters for `cogen-driver` we need to specify a filename for the output.

```
> (cogen-driver (list "examples/power.scm") '(power 1 0) "/tmp/power1.scm")
bta-run
bta-solve
bta-solve done
'((define (power x_1 n_1) (if (_op 0 = 0 n_1) (_lift 0 1 1) (_op 1 * x_1 (power x_1 (_op 0 -
>
```

This command generates two files:

- `/tmp/power1.scm` contains the code of the generating extension (pretty printed) and
- `/tmp/power1.config.scm` contains the declarations for the interface and the structure of the generating extension. For the example, PGG generates the following declarations:

```
(define-interface
  power1-interface
  (export $goal))
(define-structure
  power1
  power1-interface
  (open scheme signals define-data pgg-library)
  (files power1))
```

To use the generating extension from this module, we need to make Scheme48 aware of it.

```
> ,config,load /tmp/power1.config.scm
/tmp/power1.config.scm
>
```

Now the system can load and compile the module, just by referencing it with its name.

```
> ,open power1
Load structure power1 (y/n)? y
[define-data cogen-ctors.scm]
[power1 /tmp/power1.scm]
>
```

Finally, we can specialize in the same way as before.

```
> (specialize $goal '($goal 1 0) '(x 0))
'($goal-1 x)
> (get-residual-program)
'((define ($goal-1 x-1) 1))
>
```

Section 4.10.1 in the reference part lists a number of options to gain more control over the module declaration.

3.6 Specialization with respect to indexed data

It is possible to split the static data into an indexed set of data fragments. The main catch is that only one particular indexed value is available to each single run of the specializer, the current world. The specializer can request arbitrary elements (worlds) from this set using a special construct. If the request concerns the current world then the specializer continues right away. Otherwise, it checks the memoization cache. If the requested world has already been seen in the past, it might be possible to resolve the request. Otherwise, the specializer generates a new memoization point which waits until the requested world becomes available to the specializer, possibly for the second time.

The most striking application for this feature is the separate compilation of modular programs by specializing an interpreter. In this application, the index values are the names of modules and the standard semantics of the special construct is to load the module's text into memory.

An an example, we consider the compilation of a simple register machine language. Here is an example session.

```
> (load "examples/modint-examples.scm")
examples/modint-examples.scm
> (p module1)
((add (jz 1 copy)
      (decr 1)
      (incr 0)
      (jump add))
 (finis))
> (p module2)
((copy (jz 2 test)
       (incr 1)
       (decr 2)
```

```
       (jump copy))
 (test (jz 1 finis)
       (jump add)))
```

The `main` function of the interpreter for this register-machine language accepts four parameters, a function that maps a label to a module name, `modulename-of`, the entry label, `name`, the number of registers, `nargs`, and the initial contents of the registers, `initial_args`. The `name` and `nargs` inputs are known statically, the other inputs are dynamic.

```
> (define genext
    (cogen-driver '("examples/modint-base.scm" "examples/modint.scm")
                  '(main 1 0 0 1)))
bta-run
interpret-type: #(type-all t #(type-app -> (#(type-app b ()) #(type-app -> (#(type-ap
interpret-type: #(type-all t #(type-var t))
bta-solve
bta-solve done
> ,open pgg-residual
> (writelpp genext "/tmp/modint0.scm")
> (load "/tmp/modint0.scm")
> (specialize-$goal 'add 2)
'(main-1 x1 x4)
```

Specialization stops right before loading the first module. So far, it generated code for transferring the input list into the registers:

```
> (p (get-residual-program))
((define (main-1 x-2 x-1)
   (let* ((mlet-3 (car x-1))
          (mlet-4 (cdr x-1))
          (mlet-5 (car mlet-4))
          (mlet-6 (cdr mlet-4))
          (mlet-7 (x-2 'add)))
     (jump-global-2 x-2 mlet-3 mlet-5))))
```

The call to `jump-global-2` refers to code that will be generated as soon as the next module becomes available. This fact is signalled to the system via the `continue` function.

```
> (continue 'mod1 module1)
```

At any point between invocations of `continue` it is possible to suspend the state of specialization to a file. The corresponding command is

```
> (suspend "/tmp/suspended.scm")
```

Another, later session with `pgg` can resume this specialization after loading the generating extension and reading the suspended file using `resurrect`.

```
> (load "/tmp/modint0.scm")
> (load "examples/modint-examples.scm")
> (resurrect "/tmp/suspended.scm")
#t
> (continue 'mod2 module2)
> (continue 'mod1 module1)
```

The last two calls to `continue` complete the specialization of the interpreter of modular register machine programs.

The file `modint-mutual.scm` contains a more sophisticated implementation that compiles each module only once. Here is a transcript:

```
> (define genext
    (cogen-driver '("examples/modint-base.scm" "examples/modint-mutual.scm")
                  '(main 0 1 0 1)))
bta-run
interpret-type: #(type-all t #(type-app -> (#(type-app b ()) #(type-app -> (#(type-ap
interpret-type: #(type-app -> (#(type-app b ()) #(type-app b ())))
bta-solve
bta-solve done
> (writelpp genext "/tmp/regcompiler2.scm")
> (load "/tmp/regcompiler2.scm")
/tmp/regcompiler2.scm
> (specialize-$goal exported-labels 3)
'(main-1 x2 x4)
```

Here is the startup code for the compiled program:

```
> (p (get-residual-program))
((define (main-1 x-2 x-1)
   (let* ((mlet-3 (car x-1))
          (mlet-4 (cdr x-1))
          (mlet-5 (car mlet-4))
          (mlet-6 (cdr mlet-4))
```

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

Here is the code for the first module:

```
> (continue 'mod1 module1)
> (p (get-residual-program))
((define (jump-2 mlet-3 mlet-2 mlet-1)
   (if (zero? mlet-2)
       (jump-4 mlet-3 mlet-2 mlet-1)
       (jump-2 (+ mlet-3 1) (- mlet-2 1) mlet-1)))
 (define (jump-3 mlet-3 mlet-2 mlet-1)
   mlet-3))
```

Here is the code for the second module:

```
> (continue 'mod2 module2)
> (p (get-residual-program))
((define (jump-5 mlet-3 mlet-2 mlet-1)
   (if (zero? mlet-2)
       (jump-3 mlet-3 mlet-2 mlet-1)
       (jump-2 mlet-3 mlet-2 mlet-1)))
 (define (jump-4 mlet-3 mlet-2 mlet-1)
   (if (zero? mlet-1)
       (jump-5 mlet-3 mlet-2 mlet-1)
       (jump-4 mlet-3 (+ mlet-2 1) (- mlet-1 1)))))
```

The input for this section, along with one more example, can be found in file `examples/sample_modules_session.scm`.

4 Reference manual

4.1 Notation

The syntax definition uses an extended BNF where all symbols are terminals, except

- nonterminal symbols are capitalized;
- `::=`, `|`, `*`, `+`, `[`, `]` are metasymbols with the usual meaning (definition, alternative, zero or more repetitions, one or more repetitions, begin of optional part, end of optional part).

4.2 Type system

The type system is the system of simple types with a $\top$ type and recursion. The type language comprises the types

- basic, for every expression that *never* evaluates to a function or an element of an algebraic datatype as defined by `define-data`;
- $[\tau_1, \ldots, \tau_n] \to \tau_0$, for every expression that *always* evaluates to a function;
- `TC`$[\tau_1, \ldots, \tau_n]$, for every expression that *always* evaluates to an element of the algebraic datatype named `TC` (the τ_i are the types of the arguments of the constructors in some unspecified fixed order);
- $\top$, for every expression that cannot be given one of the other types, in particular for an expression that may evaluate
  - to a function and also to some non-function value;
  - to an element of datatype `TC` and also to an element of a different datatype `TC'` or an element of a non-algebraic type.

The PGG system performs type inference for this system using Henglein's algorithm [14]. Due to the presence of recursive types, the result of the type inference is a graph where each node is annotated with a type constructor.

4.3 Binding-time analysis

The binding-time analysis assigns a binding-time to each node in the type graph and ensures that the binding-time assignment is well-formed. Well-formedness of such a binding-time assignment B means that the annotation

on a type node is always less than or equal to the annotations on the direct descendants of that node. A node of type $\top$ is well-formed if it assumes the maximum possible binding time (it is kept dynamic throughout all stages of specialization). That means that potential type clashes are postponed to the last stage of running the program.

The binding-time analysis inserts a lift-expression on top of each expression of basic type that occurs as

- the argument of a primitive operations;
- the argument of a function;
- the "then" or "else" arm of a conditional; or
- the argument of a data constructor.

A lift-expression injects a value computed at specialization time into the residual program.

4.4 Primitive operations

For primitive operations, the binding time analysis imposes a second set S of binding-time annotations on the nodes of the type graph. The well-formedness criterium for them has two aspects. First, the S annotation is always greater than or equal to the B annotation. Second, the S annotation of a node is greater than or equal to the S annotations of each direct descendant node in the type graph. For each expression that performs a primitive operation, the specializer requires that all arguments are *leveled*. That is, the binding time analysis enforces that the S and B annotations of all arguments are equal. In consequence, all argument computations take place at the *same* binding time (see [31]).

This restriction makes it safe to allow primitive operations to have functions as arguments.

4.5 Representation analysis

PGG performs a representation analysis that assigns to each node in the type graph yet another binding time M. The M annotation of a type is wired in such a way that it reflects the maximum level $+1$ at which expressions of that type will be subject to memoization, 0 if they are never memoized. For example, an expression of a type with an M value of 0 will never be memoized. All those expressions will use the standard representation of

values of that type. If the B annotation is zero and the M value is greater than zero, the memoized representation will be used, which incurs a runtime overhead. The general condition is that expressions with $B < M$ use the memoized representation and the others use the standard representation.

Unfortunately, there is a catch: if a type is memoized and at the same time required to be leveled then the type must assume the maximum binding time. This is, because primitive operations cannot deal with the memoized representation. The catch is that we now have a cyclic dependency: B implies the placement of memoization points, which implies a setting of M, which implies a deteriorated setting of B, which implies the placement of more memoization points, and so on.

4.6 Memoization

PGG automatically inserts memoization points on top of dynamic conditionals and on top of dynamic lambdas. However, this only happens if the branches of the dynamic conditional or the body of the dynamic lambda contains a control transfer at specialization time, i.e., a static function call. Furthermore, if several of these are nested in an expression, only the outermost receives a memoization point. This is a slight refinement of the standard strategy [3, 2].

It is possible to turn this feature off for a given function by defining it using `define-without-memoization` (see 4.9.1). In any case, memoization points can be defined and inserted manually (see 4.9.5).

4.7 Special expressions

4.7.1 eval

The binding-time analysis and the specializer treat `eval` specially. The binding-time analysis enforces that the argument of `eval` is leveled. The result type is also leveled and it may have any binding time that is greater than or equal to the argument's binding time.

If the binding times are equal then the `eval` function is called at the specified level. If the binding times differ by one then the specializer simply drops the static argument value into the residual program. Otherwise, the specializer preserves the static argument for the next level and decrements the binding-time difference.

4.7.2 apply

If a function is declared as the `apply` function (see 4.9.4) then the specializer uses a special postprocessor that transforms expressions of the form

```
(apply f (cons x1 (cons x2 (cons x3 ...))))
```

into

```
(f x1 x2 x3 ...)
```

To this end, it is necessary to declare `cons` as `pure` (see 4.9.4).

4.7.3 lambda-poly

In addition to the standard lambda abstraction there is a polyvariant memoizing abstraction operator.

```
E ::= (lambda-poly (V*) E*)
```

The specializer treats a dynamic `lambda-poly` just like an ordinary `lambda`. A static `lambda-poly` specializes to a vector of all required specializations of the abstraction. The specializer constructs a partially static value consisting of a memoization map and a reference to the vector. Both, the vector and the memoization map, are initially empty. Whenever the specializer applies a `lambda-poly` it looks up the static skeleton of the arguments in the memoization map. If a specialization for this skeleton is already present in the map, it constructs a reference to the corresponding position in the vector. Otherwise, it extends the memoization map, constructs a new specialization of the `lambda-poly`, and inserts it into the vector.

This construct implements a first-class memoization mechanism and it could be used to replace the usual memoization.

4.8 Predefined operators

These operators can be used in source programs.

The module `cogen-boxops` exports the following operators that manipulate references (boxed values):

```
(make-cell exp)
```

allocates a new mutable reference cell which initially contains the value of `exp`. Returns the reference to the new cell.

```
(cell-ref exp)
```

returns the value stored in the referenced cell if `exp` evaluates to a reference.

`(cell-set! exp1 exp2)`

stores the value of `exp2` in the cell referenced by `exp1` provided this value is a reference. The return value is unspecified.

4.9 Directives

The directives are only allowed at the top-level of a source program.

4.9.1 define-without-memoization

```
D ::= (define-without-memoization (P V*) D0* E*)
  |   (define-without-memoization P E)
```

Define procedure `P`. The specializer does not to automatically insert memoization points in the body of `P`.

4.9.2 define-data

```
D ::= (define-data TC (C Ci*)+)
  |   (define-data TC hidden (C Ci*)+)
```

Define the algebraic datatype `TC` with constructors `C` and selectors `Ci`. In addition, constructor test operations `C?` are defined. The name `TC` is used during type checking to check for equality of types.

For example, the declaration

```
(define-data list
  (nil)
  (cons car cdr))
```

defines the constructors `nil` (nullary) and `cons` binary, the constructors tests `nil?` and `cons?`, and the selectors `car` and `cdr`.

The binding-time analysis considers algebraic datatypes as partially static, i.e., the arguments of a constructor can have a different (higher) binding time than the constructor itself. In such cases, the specializer performs arity raising when appropriate. The constructors, selectors, and test operations are binding-time polyvariant, i.e., each use of such an operation may have a different binding-time pattern.

The second form of `define-data` declares a datatype whose elements are ignored by the memoization mechanism. This is a potentially dangerous feature because it can change the meaning of a program during specialization by cheating the memoization mechanism.

4.9.3 define-type

```
D ::= (define-type (P B*) B)
```

Declares the arity of primitive operation `P`. The actual values of the `B`s are currently ignored.

Example:

```
(define-type (cons * *) *)
```

declares the operator `cons` of arity 2.

A variable which is declared as an operator but does not occur at operator position in an expression is eta-expanded by the frontend according to its declaration.

4.9.4 define-primitive

```
D ::= (define-primitive O T [dynamic|error|opaque|pure|apply|Number])
```

Declares the operator `O` of type `T` with an optional property.

The parameter `T` declares the type of the operator. It can be either `-`, indicating that the type of `O` is not restricted, or it can specify a polymorphic type for `O`. The grammar is as follows:

```
T  ::=  - | T0
T0 ::=  (all TV T0) | (rec TV T0) | (TC T0*) | TV
```

Here, `TV` stands for a type variable (an arbitrary symbol) and `TC` stands for a type constructor (an arbitrary symbol). The syntax, `(all TV T0)`, declares that variable `TV` is all-quantified in `T0`, like $\forall alpha.\tau_0$. The syntax, `(rec TV T0)`, declares a recursive type, like $\mu\alpha.\tau_0$. The remaining cases are type constructor application and occurrence of a type variable. For convenience, the function type constructor, `->`, is treated specially. Writing `(-> t1 ... tn t0)` declares a Scheme function that takes `n` parameters ($\texttt{n} \geq 0$) and delivers a result of type `t0`.

The properties `dynamic` and `opaque` are synonyms. Each of them forces the binding time of `O`'s result to be dynamic. The property `error` advises the binding-time analysis that the result of the operation can assume any type whatsoever (because an error primitive raises an exception and never returns a value) and that its binding time is determined from the binding times of the arguments as with any primitive operation. The remaining properties advise the specializer what to do when it residualizes the operator. The

`pure` property states that the operator does not have side effects. Instead of creating a let-binding for the expression `(O V*)`, the specializer will treat it as a value, potentially discarding or duplicating the expression. The `apply` property states that the operator `O` is the `apply` function as defined in the Scheme standard.

If the property is a `Number` it declares the least binding time of the operator.

4.9.5 define-memo

```
D ::= (define-memo M Number [Active])
  |   (define-memo M Number 'deferred)
```

Defines `M` as a unary operator indicating a memoization point at level `Number`. This is also the binding time of the memoization point. For two-level specialization this number is `1`. Useful in connection with `define-without-memoization`. The optional parameter `Active` defines the minimum level of specialization at which the specialization point is active. The default is `0`, that is, the specialization point is always active. Useful in connection with multi-level specialization if the same program is specialized with different levels.

Both parameters, `Number` and `Active` may be integer expressions using the free variable `max-level`, which is bound to the maximum binding-time presently in use.

Example: Similix defines the operator `_sim-memoize` as an indicator for memoization points. To achieve the same behavior in PGG requires the following declaration.

```
(define-memo _sim-memoize 1)
```

The second form of the directive declares an operator to construct deferred memoization points. An applied occurrence of a deferred memoization point has the form

```
(M V E)
```

When specialization hits upon a deferred memoization point, it extracts the static skeleton and looks it up in a secondary cache. Just as with standard memoization points, it creates a function call to a specialized version of `E`. The difference is that the specialization of `E` depends on a future value of `V`, so it must be deferred until the value of `V` becomes available.

4.9.6 load

```
D ::= (load Filename)
```

includes the contents of the file named by the string `Filename` into the program. The included file may contain additional `load`s, without limit on the nesting level. The `Filename` argument is always interpreted relative to the current directory that Scheme48 is running in.

4.9.7 begin

```
D ::= (begin D*)
```

As in Scheme, top-level definitions may appear nested inside a `begin`. This is handy if a macro is to expand to more than one definition.

4.10 Commands

This section summarizes the available top-level commands of the PGG system.

4.10.1 Creating a generating extension

The main entry point of the system is the function `cogen-driver`. It takes the following parameters

```
(cogen-driver InputSpec BindingTimeSkeleton)
```

where

- `InputSpec` is either a single string specifying the name of a "jobfile" that contains a list of filenames, or a list of strings each of which specifies a filename: the source program is the content of all these files concatenated together.
- `BindingTimeSkeleton` is a list where the first element is a symbol denoting the main function of the source program and the remaining elements are binding times for the parameters of the main function. The main function must have exactly as many parameters as the `BindingTimeSkeleton` indicates.

A binding time is a non-negative integer, with `0` denoting static. The PGG system assumes that the binding time of the result of the main function is

the maximum of `1` and the binding times of the function's parameters, i.e., the `BindingTimeSkeleton`.

The result of calling `cogen-driver` is a generating extension, i.e., a list of Scheme definitions that can be loaded and run.

This function is defined in module `pgg`. If it is not accessible try

```
> ,open pgg
```

at the top-level Scheme48 prompt.

A number of optional parameters may be specified after the `BindingTime-Skeleton` argument. They allow the generation of the generating extension as a Scheme48 module. The following options are recognized.

- `(goal SYMBOL)` specifies that `SYMBOL` will be the name of the specialization entry point (i.e., the first parameter to `specialize`) of the generating extension.
- `(export SYMBOL ...)` adds the listed `SYMBOL`s to the export list of the generated module.
- `(open SYMBOL ...)` considers the listed `SYMBOL`s as module names that are to be opened to run the generating extension.
- `(files SYMBOL ...)` considers the listed `SYMBOL`s as names of files that will be included in the generating module.
- `(SYMBOL ...)` is included as an option line in the structure declaration for the generating module.
- `STRING` the name of the file where the module should be written. Must be the last; subsequent options are ignored. Writes the files `STRING.scm` and `STRING.config.scm`, after stripping any extension from `STRING`.

4.10.2 Running a generating extension

The function `specialize` is the human interface to running a generating extension. It has two forms.

```
(specialize GenextMain BindingTimeSkeleton ARGS)
```

and

```
(specialize BindingTimeSkeleton ARGS NEW-GOAL)
```

where

- `GenextMain` is the main function of the generating extension,
- `BindingTimeSkeleton` is a list where the first element is a symbol denoting the *name* of main function of the generating extension, and the remaining elements are binding times its parameters. The list of binding times must be identical to the one given to `cogen-driver` when creating this generating extension.
- `ARGS` is the list of arguments to the generating extension. Its length must be equal to the number of binding times supplied in the `Binding-TimeSkeleton`. The positions corresponding to `0` entries in the skeleton contain the currently static arguments. The positions corresponding to other entries in the skeleton must contain symbols, they are used as stubs for generating identifiers.
- (optional argument) `NEW-GOAL` is the name of the entry point for the specialized program. If it is not specified, PGG invents one for you, but admittedly not a very original one.

The result is a call template for the specialized function, a list consisting of the name of the function and of names of the arguments. The residual program can be retrieved via `(get-residual-program)`. It is a list of Scheme definitions. Furthermore, the variable `*support-code*` contains additional code, for example data definitions that are necessary to run the specialized program.

If the returned call template has the form

```
'(multi-memo Level 'Goal Goal Bts Args)
```

then we have done one step of a *multi-level specialization* [13]. It means that the residual program is again a generation extension where `Level` is a number, `Goal` is the name of the entry point, `Bts` are the binding times of the arguments, and `Args` is a list of symbols of the same length. It can be loaded as usual and specialized again with `specialize` by constructing the `BindingTimeSkeleton` from `Goal` and `Bts`.

This function is defined in module `pgg-residual`. If it is not accessible try

```
> ,open pgg-residual
```

at the top-level Scheme48 prompt.

4.10.3 Continuing a specialization

The function `continue` continues a specialization that has been suspended at a deferred memoization point.

```
(continue Name Arg)
```

The parameter `Name` identifies the index that the specialization waits for. It is used to match the index value in pending deferred memoization points. The example in Section 3.6 uses the symbols `'mod1` and `'mod2` for this purpose.

The parameter `Arg` is the indexed value. This is a value of base type and its representation is completely up to the programmer. The example in Section 3.6 uses the empty list to indicate an empty world.

4.10.4 Suspend a deferred specialization

The function `suspend` writes the current memoization cache and the cache of deferred specializations to a file.

```
(suspend Filename)
```

The `Filename` parameter indicates the name of the file in which the memoization cache and the deferred cache are stored.

4.10.5 Resurrect a deferred specialization

The function `resurrect` installs a memoization cache and deferred cache from a file. It sets up the system to continue a previously suspended specialization.

```
(resurrect Filename)
```

The `Filename` parameter indicates the name of the file in which the memoization cache and the deferred cache are stored.

Returns `#t` if the file was successfully read. Otherwise, it returns `#f`.

4.11 Settable options

These options are accessible in module `cogen-globals` except where otherwise noted. Some of them only make sense for programmers who want to use the frontend separately.

- `(set-bta-display-level! n)` Default: 1.

  Display output from the binding-time analysis, $0 \leq$ `n` ≤ 4. 0 means no output.

- `(set-effect-display-level! n)` Default: 1.

  Display output from the effect analysis. 0 means no output.

- `(set-scheme->abssyn-let-insertion! v)` Default: `#f`.

  Instruct the frontend to insert let expressions for the bound variables of lambdas and definitions.

  Useful if the frontend is to be used in other projects.

- `(set-memo-optimize! v)` Default: `#t`.

  Optimize the representation of functions and algebraic datatypes. Use expensive memoized representation only for data that actually passes a memoization point.

- `(set-abssyn-maybe-coerce! v)` Default: `#t`.

  Instructs the frontend to insert provisional lift expressions at certain places. The backend eliminates these later on if they are useless. Can be turned of for using the frontend separately.

- `(set-generate-flat-program! b)` Default: `#f`.

  Instructs the generating extension to produce flat programs. By default, the residual programs have exaclty one top-level definition, all others are nested inside and invisible to the outside.

- `(gensym-ignore-name-stubs!)` (module `cogen-gensym`)

  Instructs the generating extension to ignore name stubs when generating fresh symbols.

- `(gensym-use-name-stubs!)` (module `cogen-gensym`) Default.

  Instructs the generating extension to use provided name stubs wherever possible.

- `(set-memolist-stages! n)` Default: `0`.

  Set optimization level for memoization table. If set to `n` then memoization uses `n` cascaded association lists, indexed by the first `n` elements of the static projection at a memoization point.

- `(set-lambda-is-pure! v)` Default: `#t`.

  The code generator considers lambda abstractions as pure values if this flag is set.

- `(set-lambda-is-toplevel! v)` Default: `#f`.

  Generate a toplevel function for each memoized lambda abstraction if set. Required for `suspend` and `resurrect` to work properly.

4.12 Utilities

Pretty printing is available through function `p` in module `pretty-print`. The function takes one parameter, the expression that is to be pretty-printed.

The function `(writelpp LIST FILE)` writes the list `LIST` to file `FILE` applying the pretty printer to each element of the list. It is defined in module `auxiliary`.

The contents of modules are generally available by typing

```
> ,open Modulename
```

to the top-level command line interpreter of Scheme48.

5 Differences to Scheme

PGG assumes a declarative semantics: the order of a sequence of top-level definitions does not matter, even if they are spread over several files.

PGG implements R5RS macros with the restriction that macros defined by `let-syntax` and `letrec-syntax` cannot expand to macro definitions.

6 Reading a generating extension

For debugging purposes it is sometimes helpful to read the generating extension, because it is a representation of the binding-time annotated program. Besides standard Scheme constructs it contains the following kinds of expressions, most of which are implemented as macros in module `pgg-library`. The semantics of the ellipsis `...` is the same as in the `syntax-rules` patterns of Scheme [19], zero or more repetitions of the preceding item. The list is sorted alphabetically.

- `(multi-memo level fname fct bts args)` denotes a memoization point at level `level`, `fname` is a symbol specifying the name of the generating function to run, `fct` is the function itself, `bts` is a list of binding times describing the arguments of the function, `args` is the list of arguments (must have the same length as `bts`)
- `(_app lv f a ...)` application of non-memoized function `f` to arguments `a ...` at level `lv`
- `(_app lv f a ...)` application of memoized function `f` to arguments `a ...` at level `lv`
- `(_begin lv bl e1 e2)` a `begin` at level `lv`, `bl` is the binding time of `e1`
- `(_cell-set!_memo lv ref arg)` updates a memoized reference cell `ref` at level `lv` with value `arg`
- `(_cell-eq?_memo lv ref1 ref2)` tests two memoized reference cells `ref1` and `ref2` for equality at level `lv`
- `(_ctor_memo lv (bt ...) ctor arg ...)` creates a memoized object with constructor `ctor`, `lv` is the binding time of the structure, `(bt ...)` are the binding times of the arguments `arg ...`
- `(_eval lv diff body)` the `body` becomes available at level `lv` then it is delayed for `diff` levels
- `(_freevar lv arg)` a free variable `arg` at level `lv`
- `(_if lv bl e1 e2 e3)` the conditional at level `lv`, `bl` is the binding time of the branches `e2` and `e3`, `e1` is the condition
- `(_lift0 lv val)` delay the value `val` to level `lv`
- `(_lift lv diff value)` the `value` becomes available at level `lv`, then it is delayed for another `diff` levels
- `(_lambda lv v* e)` non-memoized lambda abstraction at level `lv`, formals `v*`, body `e`
- `(_lambda_memo lv arity label fvs bts f)` memoized lambda abstraction at level `lv`, `arity` is a list of symbols serving as stubs for variable names, `label` is the unique label of the lambda, `fvs` is a list

of the (values of the) free variables, `f` is a function that maps the values of the free variables and the variable names generated from `arity` to a new body

- `(_make-cell_memo lv lab bt arg)` creates a memoized reference cell at level `lv`, with unique label `lab`, `bt` is the binding time of the argument `arg`
- `(_op lv op arg ...)` the operator `op` applied to `arg ...` at level `lv`
- `(_op_pure lv op arg ...)` the pure operator `op` applied to `arg ...` at level `lv`
- `(_s_t_memo lv sel v)` a selector or test for a memoized datastructure at level `lv`, `sel` is the selector or test function, `v` is the argument
- `(_vlambda lv (fixed-var ...) var body)` same as `_lambda`, but for variable arity; the list of `fixed-var` names the obligatory arguments and `var` names the optional argument list
- `(_vlambda_memo lv fixed-vars var label vvs bts f)` memoized lambda abstraction with variable arity functions, see `_lambda_memo` and `_vlambda` for explanation

7 Technical background

7.1 Partial evaluation in general

Good starting points for the study of partial evaluation are Jones, Gomard, and Sestoft's textbook [18], Consel and Danvy's tutorial notes [6], Mogensen and Sestoft's encyclopedia chapter [23], and Gallagher's tutorial notes on partial deduction [12]. Further material can be found in the proceedings of the Gammel Avernæs meeting (PEMC) [1, 11], in the proceedings of the ACM conferences and workshops on Partial Evaluation and Semantics-Based Program Manipulation (PEPM) [15, 4, 25, 27, 24, 5, 8], and in special issues of various journals [16, 17, 21, 28]. A comprehensive volume on partial evaluation appeared in the Lecture Notes of Computer Science series [9]. Sestoft maintains an online bibliography [26].

The above paragraph is taken from the introduction to the 1998 Symposium on Partial Evaluation [10] which is a collection of concise articles characterizing the state of the art, stating challenging problems, and outlining promising directions for future work in partial evaluation.

7.2 Directly related publications

The following publications explain various parts of the PGG system.

- Cogen in Six Lines [29] explains how to derive a handwritten multi-level cogen from a multi-level specializer and applies this to the construction of a continuation-based handwritten multi-level cogen. This is done in continuation-passing style and in direct style with control operators.
- Towards Specialization of Full Scheme [31] explains the specialization of `eval`, `apply`, and `call/cc`. It relies on a binding-time analysis that allows for higher-order primitive operations.
- Implementing Memoization for Partial Evaluation [30] gives details about the implementation strategy for partially static values in PGG.
- Correctness of a Region-Based Binding-Time Analysis [32] defines and proves correct a binding-time analysis for a lambda calculus with side-effects.
- Sound Specialization in the Presence of Computational Effects [22] defines a specialization calculus based on Moggi's computational lambda calculus and shows how to implement it. This calculus is the basis of PGG's specialization algorithm.

7.3 Structure of the implementation

The frontend of PGG is similar to the frontend of a Scheme compiler.

The first pass renames all variables, expands macros, expands back-quotes, transforms named lets to letrecs, and collects mutable variables. The second pass performs assignment conversion, eliminating all `(set! v e)` operations in favor of `(cell-set! v e')`, replacing all uses of `v` by `(cell-ref v)`, and changing the definition of `v` accordingly. The third pass performs lambda lifting. The fourth pass transforms to abstract syntax and performs eta expansion.

The next phase is binding-time analysis. It consists of type inference, effect analysis (if `cell-set!` and friends have been used), construction of the binding-time constraints, solution of the constraints, and the introduction of memoization points.

Finally, the backend produces the generation extension.

8 Known problems

- Syntax errors are not dealt with gracefully.
- Do not use identifiers that end with `([-_][0-9]+)+` (interpreted as a regular expression for the `regexp` library) for procedures and global variables.

Acknowledgments

Most parts of the system have been developed while the author was at Tübingen University. Special thanks to Michael Sperber for unwaveringly testing the system, pushing it to its limits, suggesting new features, finding many problems (as well as some surprising features), and supplying some bug fixes. Thanks also to Simon Helsen and Frank Knoll who suffered through various versions of the system.

References

[1] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*, Amsterdam, 1988. North-Holland.

[2] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[3] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.

[4] Charles Consel, editor. *Proceedings of the 1992 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Francisco, CA, June 1992. Yale University. Report YALEU/DCS/RR-909.

[5] Charles Consel, editor. *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

[6] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

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

[8] Olivier Danvy, editor. *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '99*, San Antonio, Texas, USA, January 1999. BRICS Notes Series NS-99-1.

[9] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Dagstuhl Seminar on Partial Evaluation 1996*, number 1110 in Lecture Notes in Computer Science, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.

[10] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *1998 Symposium on Partial Evaluation*, volume 30 of *ACM Computing Surveys*. ACM Press, September 1998.

[11] Andrei P. Ershov, Dines Bjørner, Yoshihiko Futamura, K. Furukawa, Anders Haraldsson, and William Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987 (New Generation Computing, vol. 6, nos. 2,3)*. Ohmsha Ltd. and Springer-Verlag, 1988.

[12] John Gallagher. Tutorial on specialisation of logic programs. In Schmidt [25], pages 88–98.

[13] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In Doaitse Swierstra and Manuel Hermenegildo, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95)*, number 982 in Lecture Notes in Computer Science, pages 259–278, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[14] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In John Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, number 523 in Lecture Notes in Computer Science, pages 448–472, Cambridge, MA, 1991. Springer-Verlag.

[15] Paul Hudak and Neil D. Jones, editors. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, New Haven, CT, USA, June 1991. ACM. SIGPLAN Notices 26(9).

[16] Journal of Functional Programming 3(3), special issue on partial evaluation, July 1993. Neil D. Jones, editor.

[17] Journal of Logic Programming 16 (1,2), special issue on partial deduction, 1993. Jan Komorowski, editor.

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

[19] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[20] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[21] Lisp and Symbolic Computation 8 (3), special issue on partial evaluation, 1995. Peter Sestoft and Harald Søndergaard, editors.

[22] Julia Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proceedings of the Theoretical Aspects of Computer Software*, number 1281 in Lecture Notes in Computer Science, pages 165–190, Sendai, Japan, September 1997. Springer-Verlag.

[23] Torben Æ. Mogensen and Peter Sestoft. Partial evaluation. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.

[24] William Scherlis, editor. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, La Jolla, CA, USA, June 1995. ACM Press.

[25] David Schmidt, editor. *Proceedings of the 1993 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993. ACM Press.

[26] Peter Sestoft. Bibliography on partial evaluation. Available through URL ftp://ftp.diku.dk/pub/diku/dists/jones-book/partial-eval.bib.Z.

[27] Peter Sestoft and Harald Søndergaard, editors. *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Fla., June 1994. University of Melbourne, Australia. Technical Report 94/9, Department of Computer Science.

[28] Theoretical Computer Science, special issue on partial evaluation, 1998. Charles Consel, editor.

[29] Peter Thiemann. Cogen in six lines. In Kent Dybvig, editor, *Proceedings of the 1996 International Conference on Functional Programming*, pages 180–189, Philadelphia, PA, May 1996. ACM Press, New York.

[30] Peter Thiemann. Implementing memoization for partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, number 1140 in Lecture Notes in Computer Science, pages 198–212, Aachen, Germany, September 1996. Springer-Verlag.

[31] Peter Thiemann. Towards partial evaluation of full Scheme. In Gregor Kiczales, editor, *Reflection'96*, pages 95–106, San Francisco, CA, USA, April 1996.

[32] Peter Thiemann. Correctness of a region-based binding-time analysis. In *Proceedings of the 1997 Conference on Mathematical Foundations of Programming Semantics*, volume 6 of *Electronic Notes in Theoretical Computer Science*, page 26, Pittsburgh, PA, March 1997. Carnegie Mellon University, Elsevier Science BV. URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

Index

support-code **31**

apply **25**, **27**, 37

begin **29**
binding time
  skeleton 6, **29**, **31**
  specification of 28, 29

cell-ref **25**, 37
cell-set! **25**, 37
cogen-driver 6, 7, 12, **29**
continue 19, **32**

define-data 9, 11–13, 22, **26**
define-memo **28**
define-primitive **27**
define-type **27**
define-without-memoization . . 11, **26**, 28
dynamic **27**

error **27**
eval **24**, 37

gensym-ignore-name-stubs! . . . **33**
gensym-use-name-stubs! **33**
get-residual-program 8, **31**

heapsize 6

lambda-poly **25**
load **29**

make-cell **25**
memoization point 24, 28
  deferred 28
multi-level specialization . . 15, 28, 31

opaque **27**

p **34**
partially static 9, 26
pgg.image **6**
pure 25, **27**

resume **32**
resurrect 20, 34

set-abssyn-maybe-coerce! **33**
set-bta-display-level! **33**
set-effect-display-level! . . . **33**
set-generate-flat-program! . . **33**
set-lambda-is-pure! **34**
set-lambda-is-toplevel! **34**
set-memo-optimize! **33**
set-memolist-stages! **33**
set-scheme->abssyn-let-insertion! **33**
specialization
  modular programs 30
specialize **30**
suspend 20, **32**, 34

writelpp **34**