Online Type-Directed Partial Evaluation for Dynamically-Typed Languages

sumii

1999 年 1 月 12 日

要旨

Danvy の type-directed partial evaluation [Dan96b] を、Scheme のような動的に型付けされた言語に適用する方式を提案する。

1 背景

partial evaluation (PE) とは、二つ (以上) の入力を持つプログラム p と、その一番目の入力 s が与えられたときに、任意の二番目の入力 d に対して $\mathtt{p}_\mathtt{s}\ \mathtt{d} = \mathtt{p\ s\ d}$ を満たすような、特化されたプログラム $\mathtt{p}_\mathtt{s}$ を生成する処理である。たとえば、λ-calculus において、$\mathtt{p} = \lambda\mathtt{f}.\ \lambda\mathtt{x}.\ (\mathtt{f}\ (\mathtt{f}\ \mathtt{x}))$, $\mathtt{s} = \lambda\mathtt{z}.\ \mathtt{z}$ とすると、$\mathtt{p}_\mathtt{s}$ は $\lambda\mathtt{x}.\ \mathtt{x}$ となる (べきである)。

従来の *syntax-directed* partial evaluation (SDPE) では、p と s の式を見て、それを symbolic に操作することにより、$\mathtt{p}_\mathtt{s}$ を求める。それに対して、*type-directed* partial evaluation (TDPE) では、すでに評価されてしまった (p s) の値とその型から、評価すると (p s) になるような式のある種の強正規形 (long $\beta\eta$-normal form) を求める。

上の例だと、(p s) は (たとえば) $\forall\alpha.\ \alpha \to \alpha$ と型付けられるので、ある型の値を渡すと、その型の値を返す関数であるとわかる。そこで、fresh な symbol を適当に生成し ('v とする)、(p s) に渡してみると、'v という symbol が返ってくる。'v を渡すと 'v が返ってくるのだから、(p s) の値は、'(lambda (v) v) という式を評価した値と「同じ」であるとわかる。これは、(p s) を `(lambda (v) ,((p s) 'v)) と「η 展開」してから評価したことに相当する。このように与えられた型の値を式に展開することを (TDPE では) *reification* という。

関数の返値がまた関数などである場合には、reification を再帰的に行う。たとえば、$\lambda\mathtt{x}.\ \lambda\mathtt{y}.\ (1+2)$ を評価した値 (f とする) は $\forall\alpha.\ \forall\beta.\ \alpha \to \beta \to \mathit{int}$ と型付けられるので、f に 'v という symbol を与えると、また関数 (g とする) が返ってくることが、型よりわかる。そこで、さらに g に 'w という symbol を与えると、3 が返ってくる。したがって、f は '(lambda (v) (lambda (w) 3)) という式に reify される。

関数の引数のほうが関数である場合にはどうするべきか？ たとえば、$\mathtt{f} = \lambda\mathtt{g}.\ (\mathtt{g}\ (1+2))$ を考えてみる。f は $\forall\alpha.\ (\mathit{int} \to \alpha) \to \alpha$ と型付けられる。ここで、もし naive に f を 'v に適用したら、symbol が整数に適用されてしまうことになり、型エラーが発生してしまう。そこで、$\mathit{int} \to \alpha$ という型の値を得るために、'v を $\lambda\mathtt{x}.\ `(\mathtt{v}\ ,\mathtt{x})$ と「η 展開」する。これを f に与えれば、ちゃんと '(v 3) という式が返ってくるので、f は '(lambda (v) (v 3)) という式に reify される。このように式を与えられた型の値に展開することを (TDPE では) *reflection* という。

関数だけではなく、組についても同様の方法で reification/reflection が可能である。まとめると、Scheme の simply-typed な subset に対する TDPE は、図 1 のように書ける。

```
;  types are represented as follows:
;    '(base . name)
;    '(func . (domain . codomoin))
;    '(pair . (car . cdr))

(define (reify t v)
  (cond
   [(eq? (car t) 'base)
    v]
   [(eq? (car t) 'func)
    (let ([x (gensym!)])
      `(lambda (,x) ,(reify (cddr t) (v (reflect (cadr t) x)))))]
   [(eq? (car t) 'pair)
    `(cons ,(reify (cadr t) (car v)) ,(reify (cddr t) (cdr v)))]))

(define (reflect t e)
  (cond
   [(eq? (car t) 'base)
    e]
   [(eq? (car t) 'func)
    (lambda (v) (reflect (cddr t) `(,e ,(reify (cadr t) v))))]
   [(eq? (car t) 'pair)
    (cons (reflect (cadr t) `(car ,e)) (reflect (cddr t) `(cdr ,e)))]))

(define tdpe reify)
```

図 1: TDPE for Simply-Typed Subset of Scheme

2 問題点

上述の方法を simply-typed λ-calculus よりも強力な言語に適用しようとすると、 reification は簡単なのだが reflection が困難になる。たとえば、 symbol を整数に reflect することはできないので、整数を引数とする関数を TDPE するには、 reification/reflection とは別個の方法が必要になる。 variant type や recursive type に関する reflection も (できなくはないのだが) 複雑になる。一般に、 type system が強力になればなるほどに、 reflection も困難になる。

その最たる例は動的型付けである。 Scheme のような動的に型付けられた言語では、ある関数を reify しようとしても、その引数はいろいろな型を持ち得るので、 'v や 'w などの symbol をどう reflect すれば良いのかがわからず、はたと困ってしまう。 (ちなみに、すでに評価された値の型は`procedure?` とか `pair?` とかを使って調べられるので、 reflection さえどうにかなれば、 reification は

```
(define (reify v)
  (cond
   [(integer? v)
    v]
   [(procedure? v)
    (let ([x (gensym!)])
      `(lambda (,x) ,(reify (v (reflect x)))))]
   [(pair? v)
    `(cons ,(reify (car v)) ,(reify (cdr v)))]
   ...))
```

のように直ちにできる。)

また、たとえ可能な場合でも、 reflection は PE 時や実行時に非効率の原因となる。たとえば、 $\mathtt{f} = \lambda \mathtt{z}.\ \mathtt{z}$ として、 $\mathtt{g} = \lambda \mathtt{x}.\ \textbf{if}\ \mathtt{true}\ \textbf{then}\ \mathtt{f}(\mathtt{x})\ \textbf{else}\ \mathtt{f}(\lambda \mathtt{y}.\ \mathtt{y})$ を考える。 ML のような type system では、 g は $\forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$ と型付けられるので、 g の引数は関数であることになる。そこで、 g を reify するときは、 (たと

えば）'v という symbol を λa. `(v ,a) という関数に reflect してから、g に渡す。すると、λa. `(v ,a) という関数が返ってくる。それを reify すると（たとえば）'(lambda (w) (v w)) のような式になる。よって、全体としては'(lambda (v) (lambda (w) (v w))) という式が得られる。

しかし、この場合は、もし 'v を reflect せずに g に渡していれば、'v が返ってきただろうから、'(lambda (v) v) という、より簡単な式がより容易に得られたはずである。

3 解決策

以上のように、reflection にはいろいろと問題が多い。そもそもなぜ reflection が必要になったかといえば、f = λg. (g (1 + 2)) のような関数を reify するときに、f を単なる symbol に apply すると、(g (1 + 2)) という関数適用において、関数の部分に symbol が来てしまうためであった。

そこで、reflection を行うかわりに、関数適用などの primitive な operation を、operand に式が来たら、結果も式を返すように拡張することを提案する。たとえば、関数適用については、普通の関数適用のかわりに、

$$
\begin{array}{lcll}
f \ @' \ a & = & (f \ a) & (f \text{ がすでに評価された値の場合}) \\
f \ @' \ a & = & `(f \ , (\texttt{reify} \ a)) & (f \text{ がまだ評価されていない式の場合})
\end{array}
$$

と定めた @' を用いるようにすれば、関数への reflection は不要になる。上の例だったら、f' = λg. (g @' (1 + 2)) とすれば、f' を 'v に apply しても型エラーにはならず、ちゃんと'(v 3) という式が返ってくる。

pair についても、car' や cdr' を、p がまだ評価されていない式のときは (car' p) = `(car ,p) および(cdr' p) = `(cdr ,p) と定めればよい。

ただ、このようにすると、reflection が不要になるかわりに、従来はすでに評価された値した来なかった operand に、まだ評価されていない式が来るようになるので、それらを区別するための仕組みが必要になる。それには、まだ評価されていない式には専用の tag をつけ、各 primitive operation では必要に応じて operand にその tag がついていないかを test すればよい。Scheme だったら、たとえば、

- '~v とか '~w とかのように、~ で始まる symbol
- '(_@ ~v ~w) とか '(_cons ~x ~y) とかのように、先頭要素が _ で始まる symbol である list

で式を表すことにして、それらの symbol は「予約語」としてしまえば、

```
(define (dynamic-variable? x)
  (and (symbol? x)
       (eqv? #~ (car (string->list (symbol->string x))))))

(define (dynamic-operation? x)
  (and (pair? x)
       (symbol? (car x))
       (eqv? #_ (car (string->list (symbol->string (car x)))))))

(define (dynamic? x)
  (or (dynamic-variable? x)
      (dynamic-operation? x)))
```

のように区別をつけることができ、関数適用などは

```
(define (_@ proc args)
  (if (dynamic? proc)
      `(_@ ,proc ,(reify args))
      (proc args)))
```

のように定義することができる。これらの tag や test のオーバーヘッドは、あくまで PE 時にしか存在しない。通常の実行時には、@' やcar' は、普通の関数適用や car でおきかえてしまって構わないからである。

補足： 関数適用などの primitive な operation を、上で述べたように拡張するというアイディアは、実はすでに提案されている [Dan97, She97] のだが、その対象はごく限られており、 reflection も併用されていた。したがって、動的に型付けされた言語に適用することもできなかったし、 2 節で述べたような非効率を除去することも (完全には) できなかった。

4 定式化

以上のアイディアは以下のように定式化できる。まず、対象の言語としては、 λ-calculus の中で λ-calculus の項を扱えるようにした、いわゆる two-level λ-calculus を用いる。その syntax は次のように与えられる。

$$\begin{array}{rcl} t & ::= & x \mid \lambda x.\ t \mid t_1 \mathbin{@'} t_2 \mid \mathbf{pair}(t_1, t_2) \mid \mathbf{fst}'(t) \mid \mathbf{snd}'(t) \\ & \mid & \underline{x} \mid \underline{\lambda x.}\ t \mid t_1 \mathbin{\underline{@'}} t_2 \mid \underline{\mathbf{pair}}(t_1, t_2) \mid \underline{\mathbf{fst}}'(t) \mid \underline{\mathbf{snd}}'(t) \end{array}$$

一行目の形の項を static な項といい、二行目の形の項を dynamic な項という。下線はちょうど Scheme の quasi-quote に相当する。たとえば、$\lambda x.\ (\underline{y} \mathbin{@'} x)$ は Scheme の (`lambda (x) '(_@ y ,x)`) にあたる。(普通は static な項には upper-line を引くようだが、見づらくなるのでここでは省略する。)

(operational) semantics は、基本的には普通の λ-calculus と同じだが、 $@'$, $\mathbf{fst}'$, $\mathbf{snd}'$ に対する簡約規則が次のように拡張される。

$$\begin{array}{rcl} (\lambda x.\ t_1) \mathbin{@'} t_2 & \to & [t_2/x]t_1 \\ \mathbf{fst}'(\mathbf{pair}(t_1, t_2)) & \to & t_1 \\ \mathbf{snd}'(\mathbf{pair}(t_1, t_2)) & \to & t_2 \end{array} \qquad \begin{array}{rcll} t_1 \mathbin{@'} t_2 & \to & t_1 \mathbin{\underline{@'}} t_2 & (\text{if } t_1 \text{ is dynamic}) \\ \mathbf{fst}'(t) & \to & \underline{\mathbf{fst}}'(t) & (\text{if } t \text{ is dynamic}) \\ \mathbf{snd}'(t) & \to & \underline{\mathbf{snd}}'(t) & (\text{if } t \text{ is dynamic}) \end{array}$$

$t_1 \to t_2$ のとき $\lambda x.\ t_1 \to \lambda x.\ t_2$ のように、 static な λ 抽象の内側も簡約する reduction を strong reduction といい、そうしない reduction を weak reduction という。

reification のアルゴリズムは、下のように定める。ここで、$\downarrow$ は “reify” をあらわす unary operator である。

$$\begin{array}{lcll} \downarrow(\lambda x.\ t) & \to & \underline{\lambda y.}\ \downarrow((\lambda x.\ t) \mathbin{@'} \underline{y}) & (\text{where } \underline{y} \text{ is fresh}) \\ \downarrow\mathbf{pair}(t_1, t_2) & \to & \underline{\mathbf{pair}}(\downarrow t_1, \downarrow t_2) & \\ \downarrow\underline{x} & \to & \underline{x} & \end{array} \qquad \begin{array}{lcl} \downarrow(t_1 \mathbin{\underline{@'}} t_2) & \to & \downarrow t_1 \mathbin{\underline{@'}} \downarrow t_2 \\ \downarrow\underline{\mathbf{fst}}'(t) & \to & \underline{\mathbf{fst}}'(\downarrow t) \\ \downarrow\underline{\mathbf{snd}}'(t) & \to & \underline{\mathbf{snd}}'(\downarrow t) \end{array}$$

たとえば、 $\lambda f.\ \mathbf{fst}'(f \mathbin{@'} (\lambda x.\ \mathbf{fst}'(\mathbf{pair}(x, x))))$ という項は、 weak reduction の下で、次のように reify される。

$$\begin{array}{rl} & \downarrow(\lambda f.\ \mathbf{fst}'(f \mathbin{@'} (\lambda x.\ \mathbf{fst}'(\mathbf{pair}(x, x))))) \\ \to & \underline{\lambda v.}\ \downarrow(\mathbf{fst}'(\underline{v} \mathbin{@'} (\lambda x.\ \mathbf{fst}'(\mathbf{pair}(x, x))))) \\ \to & \underline{\lambda v.}\ \downarrow(\mathbf{fst}'(\underline{v} \mathbin{\underline{@'}} (\lambda x.\ \mathbf{fst}'(\mathbf{pair}(x, x))))) \\ \to & \underline{\lambda v.}\ \downarrow(\underline{\mathbf{fst}}'(\underline{v} \mathbin{\underline{@'}} (\lambda x.\ \mathbf{fst}'(\mathbf{pair}(x, x))))) \\ \to & \underline{\lambda v.}\ \underline{\mathbf{fst}}'(\downarrow(\underline{v} \mathbin{\underline{@'}} (\lambda x.\ \mathbf{fst}'(\mathbf{pair}(x, x))))) \\ \to & \underline{\lambda v.}\ \underline{\mathbf{fst}}'(\downarrow\underline{v} \mathbin{\underline{@'}} \downarrow(\lambda x.\ \mathbf{fst}'(\mathbf{pair}(x, x))))) \\ \to & \underline{\lambda v.}\ \underline{\mathbf{fst}}'(\underline{v} \mathbin{\underline{@'}} \underline{\lambda w.}\ \downarrow(\mathbf{fst}'(\mathbf{pair}(\underline{w}, \underline{w})))) \\ \to & \underline{\lambda v.}\ \underline{\mathbf{fst}}'(\underline{v} \mathbin{\underline{@'}} \underline{\lambda w.}\ \downarrow\underline{w}) \\ \to & \underline{\lambda v.}\ \underline{\mathbf{fst}}'(\underline{v} \mathbin{\underline{@'}} \underline{\lambda w.}\ \underline{w}) \end{array}$$

Scheme で実装して試すと、確かに同じ結果が得られる。

```
> (define t
    (lambda (f)
      (_@ f
          (lambda (x)
            (_car (cons x x))))))
```

```
> (reify t)
(_lambda (~v3) (_@ ~v3 (_lambda (~v4) ~v4)))
>
```

ここで、`_@` は $@'$ に、`_car` は $\mathbf{fst}'$ に相当する (`_lambda` は普通の `lambda` と同じ)。

$\downarrow$ の「正しさ」は次のように述べられる。

定理： t を、完全に static な (= dynamic な項をまったく含まない) 任意の項とする。もし $\downarrow t$ が weak normal form t' を持てば、t' は完全に dynamic である (= static な項をまったく含まない)。さらに、t' から下線をすべて取り除けば、t の strong normal form が得られる。

証明は $\downarrow$ の operand の構造に関する帰納法による。

5 Online SDPE との関連

我々の "type-directed" PE は、普通の online な syntax-directed PE とよく似ている。実は、普通の online SDPE に対して、この間のミーティングでお話したような

- higher-order abstract syntax を使って書き直す
- syntax constructor を "peval" と合成する

という変形をすると、先と同じ $@'$, $\mathtt{fst}'$, $\mathtt{snd}'$ などの定義が得られる。また、$\downarrow$ は、higher-order abstract syntax から普通の first-order abstract syntax への変換に一致する。

「TDPE は SDPE よりも効率が良い」といわれてきたが、それは type-directed だからというよりは、上のような変形によって interpretive overhead が削減されているためであった、といえるかもしれない。

補足： 上のような変形は、offline SDPE に対しては従来から行われていたようだが [GJ95, Thi96]、online SDPE に対して行ったという話は (調べた限りでは) 見当たらなかった。

6 拡張と限界

6.1 算術演算

整数に関する演算などは、他の primitive operation と似たような方法で容易に扱える。たとえば、整数の乗算だったら、$\times'$ という operator を

$$
\begin{array}{llll}
t \times' 0 \to 0 & & t_1 \times' t_2 \to t & \text{(if both } t_1 \text{ and } t_2 \text{ are static, and } t_1 \times t_2 = t\text{)} \\
0 \times' t \to 0 & & t_1 \times' t_2 \to t_1 \mathbin{\underline{\times}}' t_2 & \text{(if either } t_1 \text{ or } t_2 \text{ is dynamic, and } t_1, t_2 \neq 0, 1\text{)} \\
t \times' 1 \to t & & & \\
1 \times' t \to t & & \downarrow(t_1 \mathbin{\underline{\times}}' t_2) \to \downarrow t_1 \mathbin{\underline{\times}}' \downarrow t_2 &
\end{array}
$$

のように定義すればよい。もちろん

$$(t_1 \mathbin{\underline{\times}}' t_2) \times' t_3 \to t_1 \mathbin{\underline{\times}}' t \qquad \text{(if } t_1 \text{ is dynamic, } t_2 \text{ and } t_3 \text{ are static, and } t_2 \times t_3 = t\text{)}$$

などの、もっと複雑な簡約を行うように $\times'$ を定義することも可能である。

6.2 副作用

実行時に起こるべき副作用は、従来の TDPE と同様の let-insertion [Dan96a] によって扱うことができる。たとえば、

```
(lambda (name)
  (_display "Hello, ")
  (_display name)
  (_display ". 1 + 2 is ")
  (_display (+ 1 2))
  (_display ".")
  (_newline))
```

という関数を reify したら、

```
(_lambda (~v1)
  (_let ((~v2 (_display "Hello, ")))
    (_let ((~v3 (_display ~v1)))
      (_let ((~v4 (_display ". 1 + 2 is ")))
        (_let ((~v5 (_display 3)))
          (_let ((~v6 (_display ".")))
            (_let ((~v7 (_newline)))
              ~v7)))))))
```

という式になる (`_lambda` や `_let` は普通の `lambda`や `let` と同じ) 。

これは、 reify 中かどうかを表す global な flag を用意して、`_display` や `_newline` は、その flag が立っていたら let-insertion を行い、その flag が寝ていたら実際の `display` や`newline` を呼び出すように定義することにより実現されている。 let-insertion は

- shift/reset [DF89, Fil94] のような partial continuation operator を使う
- 一個の global な mutable cell で binding を出し入れする

などの方法で実現できる。

`_display` や `_newline` は (意味のある) 値を返さないが、 (意味のある) 値を返し、副作用も持つような関数について、ちゃんと簡約を行うこともできる。たとえば、使うたびに “hello” と表示するcons (`verbose-cons` とする) があるとしたら、

```
(define (_verbose-cons a d)
  (if reifying
      (let ((p (gensym!)))
        (insert-let p `(_verbose-cons ,(reify a) ,(reify d)))
        (cons a d)) ; instead of p
      (cons a d)))
```

のように `_verbose-cons` を定義すれば、

```
> (define (f x)
    (_car (_verbose-cons x x)))
> (reify f)
(_lambda (~v1)
  (_let ((~v2 (_verbose-cons ~v1 ~v1)))
    ~v1))
```

のように、ちゃんと簡約が行える。

`set!` のように、 PE 時に起きるべきか実行時に起きるべきかが自明でない副作用を持つ関数は、今のところはうまく扱えない。

6.3 再帰

再帰の扱いは大きな問題である。再帰変数 (再帰呼び出しの有無を決定する変数) が未知である再帰関数を naive に reify すると、 reification が停止しなくなってしまう。たとえば、冪乗を計算する関数を

```
(define (power b e)
  (_if (_zero? e)
       1
       (_* b (power b (_+ e -1)))))
```

のように書いて、(reify (lambda (e) (power 2 e))) を評価すると、無限ループに陥ってしまう。これは、if 文において条件部分が dynamic だと、全体を reify するためには then 部分と else 部分の両方を reify しなければならないためである。

しかたがないので、今のところは

```
(define (_fix proc)
  (lambda (arg)
    (if (dynamic? arg)
        `(_@ (_fix ,(reify proc)) ,(reify arg))
        ((proc (_fix proc)) arg))))
```

のように、再帰関数の第一引数が dynamic だったら再帰呼び出しを行わないような fixed-point operator を定義している。 programmer は、再帰変数が未知である再帰関数を reify したかったら、_fix を明示的に使って、再帰変数が第一引数になるように書かなければならない (see below)。

```
> (define power
    (lambda (b)
      (_fix
       (lambda (p)
         (lambda (e)
           (_if (_positive? e)
                (_* b (_@ p (_+ e -1)))
                1))))))
> (reify (power 2))
(_lambda (~v1)
  (_@ (_fix (_lambda (~v2)
              (_lambda (~v3)
                (_if_ (_positive? ~v3)
                      (_* 2 (_@ ~v2 (_+ ~v3 -1)))
                      1))))
      ~v1))
```

7 おまけ

TDPE を使えば、すでに評価されてしまった関数を式に戻すことができるので、 partial continuation を reify することにより、「go もどき」を簡単に実装することができる。

```
> (define (go host)
    (shift (lambda (pk)
             (let* ((code (reify pk))
                    (program `(reset (,code ())))
                    (command (string-append "rsh "
                                            host
                                            " 'cd ~sumii/work/dynamic-tdpe/"
                                            " && exec chez go.scm'"
                                            " | /usr/local/gnu/bin/tail -c +73"))
                    (ports (process command))
                    (input-port (list-ref ports 0))
                    (output-port (list-ref ports 1)))
```

```
                (write program output-port)
                (close-output-port output-port)
                (read-and-display-all input-port)
                (close-input-port input-port)))))
> (define (_go host)
    (if reifying
        (let ((x (gensym!)))
          (insert-let x `(_go ,host))
          x)
        (go host)))
> (define (go-and-hello hosts)
    (if (null? hosts)
        "DONE"
        (let ((h (car hosts))
              (r (cdr hosts)))
          (_go h)
          (_system "exec hostname")
          (_display "Hello, world!")
          (_newline)
          (go-and-hello r))))
> (reset (go-and-hello (list "harp" "theremin" "celesta")))
harp
Hello, world!
theremin
Hello, world!
celesta
Hello, world!
"DONE"
>
```

参考文献

[Dan96a] Olivier Danvy. Pragmatic aspects of type-directed partial evaluation. Technical Report RS-96-15, Basic Research in Computer Science, 1996. Available from `http://www.brics.dk/RS/96/15/`. Appears in Danvy, Glück and Thiemann, editors, Partial Evaluation, PE '96 Proceedings, LNCS 1110, 1996, pages 73–94.

[Dan96b] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 242–257, 1996.

[Dan97] Olivier Danvy. Online type-directed partial evaluation. Technical Report RS-97-53, Basic Research in Computer Science, 1997. Available from `http://www.brics.dk/RS/97/53/`. Extended version of an article to appear in the Third Fuji International Symposium on Functional and Logic Programming, FLOPS '98 Proceedings (Kyoto, Japan, April 2–4, 1998).

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

[Fil94] Andrzej Filinski. Representing monads. In *Proceedings of the 21st Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 446–457, 1994.

[GJ95] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *Programming Languages: Implementations, Logics and Programs, 7th International Symposium*, No. 982 in Lecture Notes in Computer Science, pp. 259–278. Springer-Verlag, 1995.

[She97] Tim Sheard. A type-directed, on-line partial evaluator for a polymorphic language. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 22–35, 1997.

[Thi96] Peter J. Thiemann. Cogen in six lines. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pp. 180–189, 1996.