Multi-Level Specialization
(Extended Abstract)

Robert Glück[1] and Jesper Jørgensen[2]

[1] DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark.
Email: glueck@diku.dk

[2] Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark.
Email: jesper@dina.kvl.dk

Abstract. Program specialization can divide a computation into several computation stages. The program generator which we designed and implemented for a higher-order functional language converts programs into very compact multi-level generating extensions that guarantee fast successive specialization. Experimental results show a remarkable reduction of generation time and generator size compared to previous attempts of multiple self-application.

1 Introduction

The division of programs into *two stages* has been studied intensively in partial evaluation and mixed computation to separate those program expressions that can be safely evaluated at specialization time from those that cannot. The main problem with the binding-time analysis of standard *partial evaluation*, *e.g.* as presented in [13], is the need to specify the availability of data in terms of 'early' (*static*) and 'late' (*dynamic*). This two-point domain does not allow to specify multi-level transition points (*e.g.* "dynamic until stage n"). This has limited the operation of partial evaluators to a conservative two-level approximation. Our goal is more general: *multi-level specialization*.

This paper presents the key ingredients of our approach to multi-level specialization. We introduce a general binding-time domain that expresses different 'shades' of static input. This means that a given program can be optimized with respect to some inputs at an earlier stage, and others at later stages. This modification requires several non-obvious extensions of standard partial evaluation techniques, such as *multi-level generating extensions* [10], a generalization of Ershov's (two-level) generating extension [8]. The main payoff of this novel approach becomes apparent in multiple self-application: experimental results show an impressive reduction of generation time and code size compared to previous attempts of multiple self-application.

Our approach to multi-level specialization, which we call *multi-cogen approach*, shares the advantages of the traditional cogen approach [3]: the generator and the generating extensions can use all features of the implementation language (no restrictions due to self-application); the generator manipulates only syntax trees (no need to implement a self-interpreter); values in generating extensions are represented directly (no encoding overhead); and it becomes easier to demonstrate correctness for non-trivial languages (due to the simplicity of the transformation). Multi-level generating extensions are portable, stand-alone programs that can be run independently of the multi-level generator.

Our multi-level binding-time analysis [11] has the same accuracy as and is slightly faster than the two-level analysis in Similix [5] (when compared on two levels), a state-of-the-art partial evaluator, which is notable because we did not optimize our implementation for speed. The results are also significant because they clearly demonstrate that multi-level specialization scales up to advanced languages without performance penalties. The methods developed for converting programs into fast and compact multi-level generating extensions can also be taken advantage of in conventional (two-level) compiler generators.

Recently our approach was extended to continuation-based partial evaluation [20] and used in a program generator system for Standard Scheme [21]. Closely related work has been initiated by several researchers including a language for hand-writing program generators [19] and an algebraic description of multi-level lambda-calculi [16, 17].

We assume familiarity with the basic notions of partial evaluation, for example as presented in [14] or [13, Part II]. Additional details about multi-level specialization can be found in [11, 12] on which this presentation is based.

2 Generating Extensions

We summarize the concept of multi-level generating extensions [10]. The notation is adapted from [13]: for any program text, `p`, written in language L we let $[\![\mathtt{p}]\!]_{\mathrm{L}}\ \mathtt{in}$ denote the application of the L-program `p` to its input `in`. For notational convenience we assume that all program transformers are L-to-L-transformers written in L.

Ershov's Generating Extensions. A program generator `cogen`, which we call a *compiler generator* for historical reasons, is a program that takes a program `p` and its binding-time classification (bt-classification) as input and generates a program generator `p-gen`, called a *generating extension* [8], as output. The task of `p-gen` is to generate a residual program `p-res`, given static data $\mathtt{in}_0$ for `p`'s first input. We call `p-gen` a *two-level* generating extension of `p` because it realizes a two-staged computation of `p`. A generating extension `p-gen` runs potentially much faster than a program specializer because it is a program generator devoted to the generation of residual programs for `p`.

$$\begin{array}{rl} \mathtt{p\text{-}gen} = & [\![\mathtt{cogen}]\!]_{\mathrm{L}}\ \mathtt{p}\ \text{`}\mathtt{SD}\text{'} \\ \left.\begin{array}{r} \mathtt{p\text{-}res} = \\ \mathtt{out} = \end{array}\begin{array}{l} [\![\mathtt{p\text{-}gen}]\!]_{\mathrm{L}}\ \mathtt{in}_0 \\ [\![\mathtt{p\text{-}res}]\!]_{\mathrm{L}}\ \mathtt{in}_1 \end{array}\right\} & \textit{two stages} \end{array}$$

Multi-Level Generating Extensions. Program specialization can do more than stage a computation into two stages. Suppose `p` is a source program with n inputs. Assume the input is supplied in the order $\mathtt{in}_0 \ldots \mathtt{in}_{n-1}$. Given the first input $\mathtt{in}_0$ a multi-level generating extension produces a new specialized multi-level generating extension $\mathtt{p\text{-}mgen}_0$ and so on, until the final output `out` is produced given the last input $\mathtt{in}_{n-1}$. Multi-level specialization using multi-level generating extensions is described by

$$
\left.\begin{aligned}
\mathtt{p\text{-}mgen}_0 &= [\![\mathtt{mcogen}]\!]_{\mathrm{L}}\ \mathtt{p}\ `\mathtt{0}\ldots n\mathtt{-1}\text{'} \\
\mathtt{p\text{-}mgen}_1 &= [\![\mathtt{p\text{-}mgen}_0]\!]_{\mathrm{L}}\ \mathtt{in}_0 \\
&\ \ \vdots \\
\mathtt{p\text{-}mgen}_{n-2} &= [\![\mathtt{p\text{-}mgen}_{n-3}]\!]_{\mathrm{L}}\ \mathtt{in}_{n-3} \\
\mathtt{p\text{-}res}'_{n-1} &= [\![\mathtt{p\text{-}mgen}_{n-2}]\!]_{\mathrm{L}}\ \mathtt{in}_{n-2} \\
\mathtt{out} &= [\![\mathtt{p\text{-}res}'_{n-1}]\!]_{\mathrm{L}}\ \mathtt{in}_{n-1}
\end{aligned}\right\}\ n\ \textit{stages}
$$

Our approach to multi-level specialization is *purely off-line*. A program generator `mcogen`, which we call a multi-level compiler generator, or short multi-level generator, is a program that takes a program `p` and a bt-classification $t_0 \ldots t_{n-1}$ of `p`'s input parameters and generates a *multi-level generating-extension* $\mathtt{p\text{-}mgen}_0$. The order in which input is supplied is specified by the bt-classification. The smaller the bt-value t_i, the earlier the input becomes available.

It is easy to see that a standard (two-level) generating extension is a special case of a multi-level generating extension: it returns only an 'ordinary' program and never a generating extension. Programs `p-gen` and $\mathtt{p\text{-}mgen}_{n-2}$ are examples of two-level generating extensions.

3 Construction Principles

We now turn to the basic methods for constructing multi-level generating extensions. Our aim is to develop a program generator well-suited for multi-level specialization. Efficiency of the multi-level generating extensions, as well as their compactness are our main goals. We will use Scheme, an untyped, strict functional programming language, as presentation language.

Construction Principles Our approach is based on the observation that the standard static/dynamic annotation of a program is a special case of a more general multi-level annotation and on the observation that annotated programs can be considered as generating extensions given an appropriate interpretation for their annotated operations. From these two observations, we draw the following conclusions for the design of our multi-level program generator and the corresponding generating extensions.

- A non-standard, *multi-level binding-time analysis* together with a phase converting annotations into executable multi-level generating extensions forms the core of a multi-level generator `mcogen`.
- Multi-level generating extensions $\mathtt{p\text{-}mgen}_i$ can be represented using a *multi-level language* providing support for code generation etc.

Multi-Level Language. The multi-level language, called MetaScheme, is an annotated, higher-order subset of Scheme where every construct has a bt-value $t \geq 0$ as additional argument (Fig. 1). The underlining of an operator, *e.g.* $\underline{\texttt{if}}_t$, together with the bt-value t attached to it, is its *annotation*. The language provides a lift operator $\underline{\texttt{lift}}_t^s$ to coerce a value with bt-value t to a value with a later bt-time $t+s$. It is clear that not all multi-level programs have a consistent annotation. The typing rules given in the next section define *well-annotated* multi-level programs.

$p \in \mathit{Program}$; $d \in \mathit{Definition}$; $e \in \mathit{Expression}$; $c \in \mathit{Constant}$;
$x \in \mathit{Variable}$; $f \in \mathit{FctName}$; $op \in \mathit{Operator}$; $s,t \in \mathit{BindingTimeValue}$

$$
\begin{array}{lll}
p ::= d_1 ... d_m & & \\
d ::= \texttt{(define (}f\ x_1 ... x_n\texttt{)}\ e\texttt{)} & & \\
e ::= c & \mid x & \mid \texttt{(}\underline{\texttt{if}}_t\ e_1\ e_2\ e_3\texttt{)} \\
\quad \mid \texttt{(}\underline{\texttt{lambda}}_t\ \texttt{(}x_1 ... x_n\texttt{)}\ e\texttt{)} & \mid \texttt{(}e_0\ \underline{@}_t\ e_1 ... e_n\texttt{)} & \mid \texttt{(}\underline{\texttt{let}}_t\ \texttt{((}x\ e_1\texttt{))}\ e_2\texttt{)} \\
\quad \mid \texttt{(}f\ e_1 ... e_n\texttt{)} & \mid \texttt{(}\underline{op}_t\ e_1 ... e_n\texttt{)} & \mid \texttt{(}\underline{\texttt{lift}}_t^s\ e\texttt{)}
\end{array}
$$

Fig. 1. Abstract syntax of MetaScheme ($0 \leq n, 0 < m$).

Representing Multi-Level Generating Extensions Programs annotated with multiple binding-times need to be represented as executable programs and supplied with an appropriate interpretation for their annotated operations.

We follow the approach suggested by the observation that multi-level programs can be considered as programs provided a suitable interpretation for their annotated operations. Static expressions ($t = 0$) can be evaluated by the underlying implementation, while dynamic expressions ($t > 0$) are calls to code generating functions. A generating extension then consists of two parts:

1. *Multi-level program.* Representation of an annotated program as executable program.
2. *Library.* Functions for code generation and specialization.

Example 1. Consider as example the three-input program `iprod` which computes the inner product `v·w` of two vectors `v`, `w` of dimension `n` (Figure 2). Depending on the availability of the input, the computation of the inner product can be performed in one, two, and three stages. The residual program obtained by specializing `iprod` *wrt* two inputs `n`=`3` and `v`=[`7 8 9`] is shown in Figure 3. A call of the form `(ref i v)` returns the `i`'th element of vector `v`.

Figure 4 shows a three-level version of the inner product where the arguments of `iprod` have the following binding-times: `n:0`, `v:1`, and `w:2`. The program is annotated using a *concrete multi-level syntax* of MetaScheme where all dynamic operations have a binding-time value as additional argument. Binding-time annotations can be represented conveniently by marking every dynamic operation with an underscore (_). The general format of dynamic operations is

```
(define (iprod n v w)
  (if (> n 0)
      (+
       (* (ref n v)
          (ref n w))
       (iprod (- n 1) v w))
      0))
```

Fig. 2. Source program.

```
(define (iprod-nv w)
  (+ (* 9 (ref 3 w))
     (+ (* 8 (ref 2 w))
        (+ (* 7 (ref 1 w)) 0))))
```

Fig. 3. Residual program (n=3, v=[7 8 9]).

```
(define (iprod3 n v w)
  (if (> n 0)
      (_ '+ 2
       (_ '* 2
        (_ 'lift 1 1
         (_ 'ref 1 (lift 1 n) v))
         (_ 'ref 2 (lift 2 n) w))
       (iprod3 (- n 1) v w))
      (lift 2 0)))
```

Fig. 4. A multi-level program.

```
(define (_ op t . es)
  (if (= t 1)
     `(,op . ,es)
     `(_ (QUOTE ,op) ,(- t 1) . ,es)))

(define (lift s e)
  (if (= s 1)
     `(QUOTE ,e)
     `(LIFT ,(- s 1) (QUOTE ,e))))
```

Fig. 5. Multi-level code generation.

`(_ '`*op* t e_1 ... e_n`)` where t is the binding-time value and e_i are annotated argument expressions (the underscore `_` is a legal identifier in Scheme). If $t = 0$, then we simply write `(`*op* e_1 ... e_n`)`. For example, for `(if`$_0$ e_1 e_2 e_3`)` we write `(if` e_1 e_2 e_3`)`, and for `(lift`$_0^s$ e`)` we write `(lift` s e`)`.

Multi-Level Code Generation Figure 5 shows an excerpt of the library. The functions are the same for all generating extensions.

Function `_` has three arguments: an operator `op`, a binding-time value `t`, and the arguments `es` of the operator `op` (code fragments). If `t` equals `1` then the function produces an expression for `op` that can be evaluated directly by the underlying implementation. Otherwise, it reproduces a call to itself where `t` is decreased by 1. Argument `t` is decremented until `t` reaches `1` which means that `op` expects its arguments in the next stage.

Function `lift` 'freezes' its argument `e` (a value). It counts the binding-time value `s` down to 1 before releasing `s` as literal constant. An expression of the form `(_ 'lift` t s e`)` is used when it takes t specializations before the value of e is known and $t + s$ specializations before it can be consumed by the enclosing expression ($s > 0$). Since `lift` is just an ordinary function, it can be delayed using function `_` (necessary as long as the value of e is not available).

Running a Multi-Level Program The body of the two-level generating extension in Figure 6 is obtained by evaluating the three-level generating extension

```
(define (iprod3-n v w)
  (_ '+ 1 (_ '* 1 (lift 1 (ref 3 v))
               (_ 'ref 1 (lift 1 3) w))
     (_ '+ 1 (_ '* 1 (lift 1 (ref 2 v))
                  (_ 'ref 1 (lift 1 2) w))
        (_ '+ 1 (_ '* 1 (lift 1 (ref 1 v))
                     (_ 'ref 1 (lift 1 1) w))
              (lift 1 0)))))
```

Fig. 6. A generated generating extension (n=3).

in Figure 4 together with the definitions for multi-level code generation in Figure 5 where `n = 3`. Bound checks are eliminated, binding time arguments are decremented, e.g. `(_ 'ref 1 ... w)`. Evaluating `iprod-n` with `v`=[`7 8 9`] returns the same program as shown in Figure 3.

The example illustrates the main advantages of this approach: fast and compact generating extensions. No extra interpretive overhead is introduced since library functions are linked with the multi-level program at loading/compile-time. The library adds only a constant size of code to a multi-level program. Static operations can be executed by the underlying implementation. One could provide an interpreter for multi-level programs, but this would be less efficient.

Programs can be generated very elegantly in Scheme because its abstract and concrete syntax coincide. Other programming languages may need more effort to obtain syntactically correct multi-level programs. Generating extensions for languages with side-effects, such as C, require an additional management of the static store to restore previous computation states [1]. The paper [12] extends the above methods into a full implementation with higher-order, polyvariant specialization.

4 Multi-Level Binding-Time Analysis

We specify a *multi-level binding-time analysis* (MBTA) for the multi-level generator `mcogen` in the remainder of this paper. The task of the MBTA is briefly stated: given a source program p, the binding-time values (bt-values) t_i of its input parameters together with a maximal bt-value ν, find a consistent multi-level annotation of p which is, in some sense, the 'best'. We give typing rules that define well-annotated multi-level programs and specify the analysis.

The typing rules formalize the intuition that early values may not depend on late values. They define *well-annotated* multi-level programs. Before we give the set of rules, we formalize bt-values and bt-types.

Definition 1 (binding-time value). *A* binding-time value *(bt-value) is a natural number $t \in \{0, 1, \ldots, \nu\}$ where ν is the* maximal bt-value *for the given problem.*

A *binding-time type* τ contains information about the type of a value, as well as the bt-value of the type. The bt-value of an expression e in a multi-level program is equal to the bt-value $\|\tau\|$ of its bt-type τ. In case an expression is well-typed (*wrt* a monomorphic type system with recursive types and one common base type), the type component of its bt-type τ is the same as the standard type.

Definition 2 (binding-time type). *A type τ is a (well-formed) binding-time type* wrt ν, *if $\vdash\tau{:}t$ is derivable from the rules below. If $\vdash\tau{:}t$ then the type τ* represents *a bt-value t, and we define a mapping $\|\cdot\|$ from bt-types to bt-values: $\|\tau\| = t$* **iff** $\vdash\tau{:}t$.

$$\{Base\}\ \frac{t \leq \nu}{\Delta\vdash B^t{:}t} \qquad\qquad \{Fct\}\ \frac{\Delta\vdash\tau_i{:}s_i \quad \Delta\vdash\tau{:}s \quad s_i \geq t \quad s \geq t}{\Delta\vdash\tau_1 ... \tau_n \rightarrow^t \tau{:}t}$$

$$\{Btv\}\ \frac{\alpha{:}t \text{ in } \Delta \quad t \leq \nu}{\Delta\vdash\alpha{:}t} \qquad\qquad \{Rec\}\ \frac{\Delta \oplus \{\alpha{:}t\}\vdash\tau{:}t}{\Delta\vdash\mu\alpha.\tau{:}t}$$

Base bt-types, shown in Rule $\{Base\}$, are denoted by B^t where t is the bt-value. We do not distinguish between different base types, *e.g.* integer, boolean, *etc.*, since we are only interested in the distinction between base values and functions. Rule $\{Fct\}$ for function types requires that the bt-values of the argument types $\tau_1 ... \tau_n$ and the result type τ are *not* smaller than the bt-value t of the function itself because neither the arguments are available to the function's body nor can the result be computed *before* the function is applied. Rule $\{Btv\}$ ensures that the bt-value t assigned to a type variable α is never greater than ν. Rule $\{Rec\}$ for recursive types $\mu\alpha.\tau$ states that τ has the same bt-value t as the recursive type $\mu\alpha.\tau$ under the assumption that the type variable α has the bt-value t. The notation $\Delta \oplus \{\alpha : t\}$ denotes that the bt-environment Δ is extended with $\{\alpha : t\}$ while any other assignment $\alpha : t'$ is removed from Δ. This is in accordance with the equality $\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$ which holds for recursive types.

An *equivalence relation* on bt-types allows us to type *all* expressions in our source language even though the language is dynamically typed. In particular, we can type expressions where the standard types cannot be unified because of potential type errors (function values used as base values, base values used as function values). By using this equivalence relation we can defer such errors to the latest possible binding time.

Definition 3 (equivalence of bt-types). *Let ν be a maximal bt-value and let U be the following axiom:*

$$\vdash B^\nu \dots B^\nu \rightarrow^\nu B^\nu \doteq B^\nu$$

Given two bt-types τ and τ' well-formed wrt ν, *we say that τ and τ' are equivalent, denoted by $\vdash\tau \doteq \tau'$, if $\vdash\tau \doteq \tau'$ is derivable from*

1. *Axiom U*
2. *the equivalence of recursive types (based on types having the same regular type)*
3. *symmetry, reflexivity, transitivity, and compatibility of $=$ with arbitrary contexts*

Typing Rules. The typing rules for well-annotated multi-level programs are defined in Fig. 7. Most of the typing rules are generalizations of the corresponding rules used for two-level programs in partial evaluation, *e.g.* [13]. For instance, rule $\{If\}$ for `if`-expressions annotates the construct `if` with the bt-value t of the test-expression e_1 (the `if`-expression is reducible when the result of the test-expression becomes known at time t). The rule also requires that the test expression has a first-order type.

Rule $\{Lift\}$ shows the multi-level operator $\underline{\texttt{lift}}_t^s$: the value of its argument e has bt-value t, but its results is not available until $t + s$ ($s > 0, t \geq 0$). The bt-value of an expression ($\underline{\texttt{lift}}_t^s\ e$) is the sum of the bt-values s and t. In other words, the operator delays a value to a later binding time. As is customary in partial evaluation, the rule allows lifting of first-order values only.

Rule $\{Op\}$ requires that all higher-order arguments of primitive operators have bt-value ν because this is the only way to equate them with the required base type B^t (see Definition 3). This is a necessary and safe approximation since we assume nothing about the type of a primitive operator.

$$\{Con\}\ \Gamma \vdash c{:}B^0 \qquad\qquad \{Var\}\ \frac{x{:}\tau \text{ in } \Gamma}{\Gamma \vdash x{:}\tau}$$

$$\{If\}\ \frac{\Gamma \vdash e_1{:}B^t \quad \Gamma \vdash e_2{:}\tau \quad \Gamma \vdash e_3{:}\tau \quad \|\tau\| \geq t}{\Gamma \vdash (\underline{\texttt{if}}_t\ e_1\ e_2\ e_3){:}\tau} \qquad \{Call\}\ \frac{\Gamma \vdash e_i{:}\tau_i \quad f{:}\tau_1 \ldots \tau_n \rightarrow^t \tau \text{ in } \Gamma}{\Gamma \vdash (f\ e_1 \ldots e_n){:}\tau}$$

$$\{Let\}\ \frac{\Gamma \vdash e{:}\tau \quad \Gamma\{x{:}\tau\} \vdash e'{:}\tau' \quad \|\tau'\| \geq \|\tau\|}{\Gamma \vdash (\underline{\texttt{let}}_{\|\tau\|}\ ((x\ e))\ e'){:}\tau'} \qquad \{Op\}\ \frac{\Gamma \vdash e_i{:}B^t}{\Gamma \vdash (\underline{op}_t\ e_1 \ldots e_n){:}B^t}$$

$$\{Abs\}\ \frac{\Gamma\{x_i{:}\tau_i\} \vdash e{:}\tau'}{\Gamma \vdash (\underline{\lambda}_t\ x_1 \ldots x_n.e){:}\tau_1 \ldots \tau_n \rightarrow^t \tau'} \qquad \{App\}\ \frac{\Gamma \vdash e_0{:}\tau_1 \ldots \tau_n \rightarrow^t \tau' \quad \Gamma \vdash e_i{:}\tau_i}{\Gamma \vdash (e_0\ \underline{@}_t\ e_1 \ldots e_n){:}\tau'}$$

$$\{Lift\}\ \frac{\Gamma \vdash e{:}B^t \quad s{>}0}{\Gamma \vdash (\underline{\texttt{lift}}_t^s\ e){:}B^{t+s}} \qquad \{Equ\}\ \frac{\Gamma \vdash e{:}\tau \quad \vdash \tau \doteq \tau'}{\Gamma \vdash e{:}\tau'}$$

Fig. 7. Typing rules for well-annotated multi-level programs (i ranges over $0 \leq i \leq n$).

Definition 4 (well-annotated completion, minimal completion). *Given a program p, a maximal bt-value ν, and a bt-pattern $t_1 \ldots t_k$ of p's goal function f_0, a* well-annotated completion *of p is a multi-level program p' with $|p'| = p$ iff the following judgment can be derived:*

$$\vdash p'{:}\{f_0 : B^{t_1} \ldots B^{t_k} \rightarrow B^\nu, f_1 : \tau_{11} \ldots \tau_{1n_1} \rightarrow^{t_1} \tau_1, \ldots, f_n : \tau_{n1} \ldots \tau_{nn_n} \rightarrow^{t_n} \tau_n\}$$

A well-annotated completion is minimal *if the bt-value of every subexpression e in p is less than or equal to the bt-value of e in any other well-annotated completion of p.*

Every program p has at least one well-annotated completion p' since the operations of a program can always be annotated with ν, which corresponds to all subexpressions in the completion having the bt-type B^ν. A program p can have have more than one well-annotated completion. The goal of the MBTA is to determine a well-annotated completion p' which is, preferably, 'minimal', *i.e.* all operations in a program shall be performed as early as possible.

Certain programming styles can unnecessarily dynamize operations, while others make it easier to perform operations earlier. Binding-time improvements are semantics-preserving transformations of a program that make it easier for the binding-time analysis to make more operations static [13]. Fortunately, the problem of binding-time improving programs for multi-level specialization can be reduced to the two-level case where all known techniques apply.

Example 2. Let us illustrates the use of recursive types in the MBTA. Without recursive types the expression

```
(lambda (x) (x x))
```

is only typable with type B^ν (or an equivalent type) with *maximal* bt-value ν, because the expression is not typable in the simply typed λ-calculus. The following typing with *minimal* bt-value 0 makes use of recursive type $\tau \to^0 B^0$ where τ denotes $\mu\alpha.(\alpha \to^0 B^0)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x{:}\tau\} \vdash x{:}\tau}\{Var\} \qquad \vdash \tau \doteq \tau \to^0 B^0}{\{x{:}\tau\} \vdash x{:}\tau \to^0 B^0}\{Equ\} \qquad \dfrac{}{\{x{:}\tau\} \vdash x{:}\tau}\{Var\}}{\{x{:}\tau\} \vdash x \; \underline{@}_0 \; x{:}B^0}\{App\}}{\vdash \underline{\lambda}_0 x.x \; \underline{@}_0 \; x{:}\tau \to^0 B^0}\{Abs\}$$

Here we use equivalence $\doteq$ of bt-types $\mu\alpha.\alpha \to^0 B^0$ and $\mu\alpha.(\alpha \to^0 B^0) \to^0 B^0$. The two types are equivalent because their regular types (infinite unfolded types) are equal (Definition 3). In our case unfolding $\mu\alpha.\alpha \to^0 B^0$ once gives $\mu\alpha.(\alpha \to^0 B^0) \to^0 B^0$ which proves the equality. In conclusion, recursive types enable the MBTA to give earlier binding times.

5 Results

Multiple Self-Application The payoff of the multi-cogen approach becomes apparent when compared to multiple self-application. The main problem of multiple self-application is the exponential growth of generation time and code size (in the number of self-applications). While this problem has not limited self-applicable specializers up to two self-applications, it becomes critical in applications that beyond the third Futamura projection.

An experiment with multiple self-application was first carried out in [9]: staging a program for matrix transposition into 2 – 5 levels. To compare both approaches, we repeated the experiment using the multi-level generator. We generate a two-level and a five-level generating extension, `gen2` and `gen5`, respectively.

Table 1. Performance of program generators.

out	run	time/s	mem/kcells	size/cells
`mint-gen` =	$[\![$`mcogen`$]\!]$ `mint '012'`	10.0	529	1525
`comp` =	$[\![$`mint-gen`$]\!]$ `def`	.63	34	840
`tar` =	$[\![\texttt{comp}_2]\!]$ `pgm`	.083	5.18	109

Table 2. Performance of programs.

out	run	speedup	time/ms	mem/kcells
`out` =	$[\![$`mint`$]\!]$ `def pgm dat`	1	630	44.3
`out` =	$[\![$`tar`$]\!]$ `dat`	72	8.7	1.93
`out` =	$[\![$`fac`$]\!]$ `dat`	708	.89	.037

The results [12] show an impressive reduction of generation time and code size compared to the result reported for multiple self-application [9]. The ratio between the code size of `gen2` and `gen5` is reduced from 1:100 when using multiple self-application to 1:2 when using the multi-level generator. The ratio between the time needed to generate `gen2` and `gen5` is reduced from 1:9000 when using multiple self-application to 1:1.8 when using the multi-level generator.

Meta-Interpreter As another example consider a meta-interpreter `mint`, a three-input program, that takes a language definition `def`, a program `pgm`, and its data `dat` as input. Let `def` be written in some definition language D, let `pgm` be written in programming language P (defined by `def`), and let `mint` be written in programming language L. The equational definition of `mint` is

$$[\![\texttt{mint}]\!]_{\mathrm{L}}\ \texttt{def pgm dat} = [\![\texttt{def}]\!]_{\mathrm{D}}\ \texttt{pgm dat} = [\![\texttt{pgm}]\!]_{\mathrm{P}}\ \texttt{dat} = \texttt{out}$$

While this approach has many theoretical advantages, there are substantial efficiency problems in practice: considerable time may be spent on interpreting the language definition `def` rather than on computing the operations specified by the P-program `pgm`. What we look for is a three-level generating extension `mint-gen` of the meta-interpreter `mint` to perform the computation in three stages.

$$\left.\begin{array}{r}\texttt{comp} = [\![\texttt{mint-gen}]\!]_{\mathrm{L}}\ \texttt{def}\\ \texttt{tar} = [\![\texttt{comp}]\!]_{\mathrm{L}}\ \texttt{pgm}\\ \texttt{out} = [\![\texttt{tar}]\!]_{\mathrm{L}}\ \texttt{dat}\end{array}\right\}\ \textit{three stages}$$

The three-level generating extension `mint-gen` is a compiler generator which, when applied to `def`, yields `comp`. The two-level generating extension `comp` is a compiler which, when given a P-program `pgm`, returns a target program `tar`.

In our experiment [12], the meta-interpreter `mint` interprets a denotational-style definition language. The definition `def` describes a small functional language (the applied lambda calculus extended with constants, conditionals, and a fix-operator). The program `pgm` is the factorial function and the input `dat` is the number 10.

Table 1 shows the generation times, the memory allocated during the generation and the sizes of the program generators (number of cons cells). Table 2 shows the run times of the example program using the meta-interpreter and the generated target program. For comparison, we also list the run time of `fac`, the standard implementation of the factorial in Scheme. All run times were measured on a SPARC station 1 using SCM version 4e1.

We notice that the generation of the compiler `comp` is fast (0.63s), as well as the generation of the target program `tar` (0.083s). The conversion of the meta-interpreter `mint` into a compiler generator `mint-gen` is quite reasonable (10s).

The results in Table 2 demonstrate that specialization yields substantial speedups by reducing `mint`'s interpretive overhead: they improve the performance by a factor 72. The target program `tar` produced by `comp` is 'only' around 10 times slower than the factorial `fac` written directly in Scheme. Finally, interpreting `pgm` with `mint` is 700 times slower than running the Scheme version of the factorial `fac`.

One of the main reasons why the target program `tar` is slower than the standard version `fac` is that primitive operations are still interpreted in the target programs. This accounts for a factor of around 4. Post unfolding of function calls improves the runtime of these programs further by a factor 1.3.

6 Related Work

The first hand-written compiler generator based on partial evaluation techniques was, in all probability, the system *RedCompile* for a dialect of Lisp [2]. Romanenko [18] gave transformation rules that convert annotated first-order programs into two-level generating extensions. Holst [15] was the first to observe that the annotated version of a program is already a generating extension. What Holst called "syntactic currying" is now known as the "cogen approach" [3]. The multi-cogen approach presented here is based on earlier work [10–12]. Thiemann [20] extended our approach to continuation-based specialization and implemented a multi-cogen for Standard Scheme [21].

Multi-level languages have become an issue for several reasons. They are, among others, a key ingredient in the design and implementation of generative software, e.g. [7]. Taha and Sheard [19] introduce MetaML, a statically typed multi-level language for hand-writing multi-level generating extensions. Although MetaScheme was not designed for a human programmer – we were interested in automatically generating program generators – it can be seen, together with the multi-level typing-rules, as a statically typed multi-level programming language (specialization points can be inserted manually or automatically based on the annotations).

References

1. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU Report 94/19, Dept. of Computer Science, University of Copenhagen, 1994.

2. L. Beckman, A. Haraldson, Ö. Oskarsson, E. Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.
3. L. Birkedal, M. Welinder. Hand-writing program generator generators. In M. Hermenegildo, J. Penjam (eds.), *Programming Language Implementation and Logic Programming.* LNCS 844, 198–214, Springer-Verlag 1994.
4. D. Bjørner, A.P. Ershov, N.D. Jones (eds.). *Partial Evaluation and Mixed Computation.* North-Holland 1988.
5. A. Bondorf, J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.
6. O. Danvy, R. Glück, P. Thiemann (eds.). *Partial Evaluation.* LNCS 1110, Springer-Verlag 1996.
7. U. Eisenecker. Generative programming with C++. In H. Mössenböck (ed.), *Modular Programming Languages*, *LNCS* 1204, 351–365, Springer-Verlag 1997.
8. A.P. Ershov. On the essence of compilation. In E.J. Neuhold (ed.), *Formal Description of Programming Concepts*, 391–420. North-Holland 1978.
9. R. Glück. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 309–320, ACM Press 1991.
10. R. Glück, J. Jørgensen. Efficient multi-level generating extensions for program specialization. In M. Hermenegildo, S.D. Swierstra (eds.) *Programming Languages, Implementations, Logics and Programs*, LNCS 982, 259–278, Springer-Verlag 1995.
11. R. Glück, J. Jørgensen. Fast binding-time analysis for multi-level specialization. In D. Bjørner, M. Broy, I.V. Pottosin (eds.) *Perspectives of System Informatics*, LNCS 1181, 261–272, Springer-Verlag 1996.
12. R. Glück, J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2): 113–158, 1997.
13. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall 1993.
14. N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
15. C.K. Holst. Syntactic currying: yet another approach to partial evaluation. Student report, DIKU, Dept. of Computer Science, University of Copenhagen, 1989.
16. F. Nielson, H.R. Nielson. Multi-level lambda-calculus: an algebraic description. In [6], 338–354, 1996.
17. F. Nielson, H.R. Nielson. Prescriptive frameworks for multi-level lambda-calculi. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 193–202, ACM Press 1997.
18. S.A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In [4], 445–463, 1988.
19. W. Taha, T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 203–217, ACM Press 1997.
20. P. Thiemann. Cogen in six lines. In *International Conference on Functional Programming*, 180–189, ACM Press 1996.
21. P. Thiemann. The PGG system – user manual. Dept. of Computer Science, University of Nottingham, 1998.