

Efficient Multi-level Generating Extensions for Program Specialization

Robert Glück* and Jesper Jørgensen

DIKU, Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
e-mail: {glueck, knud}@diku.dk

Abstract. Multiple program specialization can stage a computation into several computation phases. This paper presents an effective solution for multiple program specialization by generalizing conventional off-line partial evaluation and integrating the “cogen approach” with a multi-level binding-time analysis. This novel “multi-cogen approach” solves two fundamental problems of self-applicable partial evaluation: the generation-time problem and the generator-size problem. The multi-level program generator has been implemented for a higher-order subset of Scheme. Experimental results show a remarkable reduction of generation time and generator size compared to previous attempts of multiple self-application.

1 Introduction

Stages of computation arise naturally in many programs, depending on the availability of data or the frequency with which the input changes. Code for later stages can be optimized based on values available in earlier stages.

Partial evaluation has received much attention because of its ability to stage a program into two computation phases. A self-applicable partial evaluator, or a compiler generator, is strong enough to convert a general algorithm, say, a general parser with two inputs (grammar, string), into the corresponding *two-level generating extension*, i.e. a parser generator. Our goal is more general: the automatic construction of efficient *multi-level generating extensions* (Sect. 2).

This paper presents an effective solution to the problem of automatically generating multi-level generating extensions by generalizing conventional off-line partial evaluation and integrating the “cogen approach” [Hol89, BW94] with a multi-level binding-time analysis (Sect. 3). We demonstrate that efficient *multi-level generators* can be built that automatically produce multi-level generating extensions from arbitrary programs. For this purpose we generalized partial evaluation techniques, such as binding-time analysis, specialization points, memoization, and code generation (Sect. 4 and 5).

This novel “*multi-cogen approach*” solves two fundamental problems of self-applicable partial evaluation [Glü91]: the generation-time problem and the generator-size problem. The multi-level generator gives an impressive reduction of generation time and code size compared to multiple self-application (Sect. 6.1).

* Supported by an Erwin-Schrödinger-Fellowship of the Austrian Science Foundation (FWF) under grant J0780 & J0964.

The multi-level generator has been implemented for a higher-order subset of Scheme. It is quite remarkable that the multi-level generator can automatically convert a two-level driving interpreter (from [GJ94]) into a new compiler generator capable of a form of supercompilation, a problem which we previously failed to solve with a state-of-the-art partial evaluator (Sect. 6.2).

Our approach shares the same advantages with the ‘direct approach’ [BW94], but for multiple program specialization: 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). Last but not least, generating extensions are stand-alone programs that can be distributed without the generator.

We claim that our approach scales up to other languages as evidenced by the fact that generators for two-level generating extensions have been implemented for typed and untyped languages and different language paradigms (e.g. λ -calculus [BD94], ML [BW93], ANSI C [And94]). We believe that the “multi-cogen approach” is more practical and promising than previous attempts of multiple self-application of partial evaluation, and in particular for applications where generation time and generator size are paramount, such as run-time code generation.

2 Multi-Level Generating Extensions

We now formulate the properties of multi-level generating extensions more precisely and discuss the theoretical limitations and practical problems of conventional specialization tools.

Notation. For any program text, p , written in language L we let $\llbracket p \rrbracket_L \text{ in}$ denote the application of the L -program p to its input in . We use typewriter font for programs and their input and output. The notation is adapted from [JGS93].

2.1 Tools for Program Specialization

Turning a general program into a specialized program can be conceived of as turning a one-stage computation into a two-stage computation. Suppose p is a source program, in_0 is data known at stage one (*static*, ‘S’) and in_1 is data known at stage two (*dynamic*, ‘D’). Then computation in one stage is described by

$$\text{out} = \llbracket p \rrbracket_L \text{ in}_0 \text{ in}_1$$

Computation in two stages using a *program specializer* spec is described by

$$\begin{aligned} p_0 &= \llbracket \text{spec} \rrbracket_L p \text{ ‘SD’ in}_0 \\ \text{out} &= \llbracket p_0 \rrbracket_L \text{ in}_1 \end{aligned}$$

where the binding-time classification ‘SD’ indicates the binding-times of p ’s input (we classify each argument of a source program according to its binding time). Combining these two we obtain an equational definition of spec :

$$\llbracket p \rrbracket_L \text{ in}_0 \text{ in}_1 = \llbracket \llbracket \text{spec} \rrbracket_L p \text{ ‘SD’ in}_0 \rrbracket_L \text{ in}_1$$

The main motivation is efficiency: the specialized program p_0 runs potentially much faster than the original program p on the same input. For notational convenience we assume that spec is an $L \rightarrow L$ -specializer written in language L ; for multi-language specialization see e.g. [Glü94].

Two-Level Generating Extensions. A program generator cogen , which we call *compiler generator* for historical reasons, is a program that accepts a two-stage program p and its binding-time classification as input and generates a program generator $p\text{-gen}$, called *generating extension* [Ers78], as output. The task of $p\text{-gen}$ is to generate a specialized program p_0 , given static data in_0 for p 's first input. We call $p\text{-gen}$ a *two-level* generating extension of p because it realizes a two-staged computation of p .

$$\begin{aligned} p\text{-gen} &= \llbracket \text{cogen} \rrbracket_L p \text{ 'SD'} \\ p_0 &= \llbracket p\text{-gen} \rrbracket_L \text{in}_0 \\ \text{out} &= \llbracket p_0 \rrbracket_L \text{in}_1 \end{aligned}$$

Combing these three we obtain an equational definition of cogen :

$$\llbracket p \rrbracket_L \text{in}_0 \text{in}_1 = \llbracket \llbracket \llbracket \text{cogen} \rrbracket_L p \text{ 'SD'} \rrbracket_L \text{in}_0 \rrbracket_L \text{in}_1$$

The specialization of p is now done in two steps: the compiler generator cogen produces p 's generating extension $p\text{-gen}$ which is then used to generate the specialized program p_0 . The generating extension $p\text{-gen}$ produces the specialized program p_0 potentially much faster than the program specializer spec because $p\text{-gen}$ is a program generator specialized with respect to p .

2.2 Multiple Program Specialization

Program specialization can do more than stage a computation into two phases. Suppose p is a source program with n inputs, where in_0 is the data known at stage one, in_1 the data known at the next stage, and so on. Then the computation in one stage can be described by

$$\text{out} = \llbracket p \rrbracket_L \text{in}_0 \dots \text{in}_n$$

Computation in n stages using a program specializer spec is described by

$$\begin{aligned} p_0 &= \llbracket \text{spec} \rrbracket_L p \text{ 'SDD...D'} \text{in}_0 \\ p_1 &= \llbracket \text{spec} \rrbracket_L p_0 \text{ 'SD...D'} \text{in}_1 \\ &\vdots \\ p_{n-1} &= \llbracket \text{spec} \rrbracket_L p_{n-2} \text{ 'SD'} \text{in}_{n-1} \\ \text{out} &= \llbracket p_{n-1} \rrbracket_L \text{in}_n \end{aligned}$$

In each step the program p_{i-1} obtained from the previous stage is specialized with respect to the next input in_i . Alternatively, we can use the compiler generator cogen in each specialization step. This may improve the performance of the i th specialization in case the program p_{i-1} is specialized with respect to different inputs.

$$\begin{aligned} p\text{-gen}_i &= \llbracket \text{cogen} \rrbracket_L p_{i-1} \text{ 'SD...D'} \\ p_i &= \llbracket p\text{-gen}_i \rrbracket_L \text{in}_i \end{aligned}$$

Multiple specialization implemented by such a specialization pipeline is a form of ‘on-line’ specialization because the i th specialization stage can take all previous values $in_0 \dots in_{i-1}$ into account (regardless whether the specialization at the i th stage is on- or off-line). Recall that the defining feature of off-line specialization is that it does not take static values into account, only the binding-time classification of the input.

Discussion. An ‘on-line’ specialization pipeline has two main advantages:

- *Precision*: the specialization at the i th specialization stage can take all static *values* into account that have been provided in the previous stages.
- *Flexibility*: at each specialization stage we are free to choose *any* binding-time classification.

But the advantages of an on-line specialization pipeline are also its disadvantages. It may be too flexible in cases where the order of the inputs $in_0 \dots in_n$ is fixed. Consider specializing a meta-interpreter with three inputs: a language definition, a program and its data. Multiple specialization does not make much sense unless the definition is available before the program, and the program is available before its data.

The precision gained by applying the full specialization power of `spec` to every ‘intermediate program’ p_i has its price: specialization time. For example, program pieces that depend on input values that will be available only in later stages, may be re-analysed each time during the earlier stages. It is also likely that these program expressions have been duplicated during a previous specialization stage, e.g. by program point specialization.

It may be difficult to perform binding-time improvements on machine-generated program because the structure of the original program may be dissolved during specialization (unless we have fully automatic methods for binding-time improvements). For example, a program resulting from specializing an interpreter with respect to an interpreted program will usually reflect the structure of both programs. This makes the specialization pipeline less predictable since the success of a specialization stage i may depend on the static values used in the previous stages.

How to binding-time improve the source program p in the first place in order to avoid the need for binding-time improvements at a later specialization stage? For this, we have to predict the staticness of program pieces independently of static values (binding-time improvements make not much sense unless they improve the binding-time for large class of values).

Multi-Level Generating Extensions. Our approach to multiple 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 accepts an n -stage program p and a binding-time classification $0 \dots n$ of its input parameters and generates a *multi-level generating-extension* $p\text{-mgen}$. Given the first input in_0 the multi-level generating extension produces a new specialized multi-level generating extension $p\text{-mgen}_0$ and so on, until the final value `out` is produced from the last input in_n (illustrated in Fig. 1). Multi-level specialization using

multi-level generating extensions is described by

$$\begin{aligned}
\text{p-mgen} &= \llbracket \text{mcogen} \rrbracket_L \text{p } '0 \dots n' \\
\text{p-mgen}_0 &= \llbracket \text{p-mgen} \rrbracket_L \text{in}_0 \\
&\vdots \\
\text{p-mgen}_{n-2} &= \llbracket \text{p-mgen}_{n-3} \rrbracket_L \text{in}_{n-2} \\
\text{p}'_{n-1} &= \llbracket \text{p-mgen}_{n-2} \rrbracket_L \text{in}_{n-1} \\
\text{out} &= \llbracket \text{p}'_{n-1} \rrbracket_L \text{in}_n
\end{aligned}$$

We call this specialization pipeline off-line because the binding-time classification is consumed by `mcogen` and no classification is required at a later stage. Note that the program `p-mgeni` returned by stage i does not become the input of a specialist at the next stage, but can be executed directly. It is easy to see that a conventional, two-level generating extension (see Sect. 2.1) is a special case of a multi-level generating extension: it returns an ‘ordinary program’ and never a generating extension.² The generating extension `p-mgenn-2` is such a two-level generating extensions.

The generation of a multi-level generating extensions pays off if it is used with various static values. In any case, it will be faster to run a multi-level generating extension at stage i than to use a full specialist.

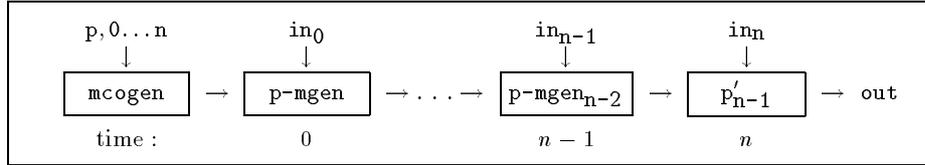


Figure 1: Program specialization with multi-level generating extensions

2.3 How to Generate Multi-Level Generating Extensions?

Assume that we have only the conventional specialization tools: a self-applicable specialist `spec` and a compiler generator `cogen`. Then we know of two methods that can, in principle, be used to generate a multi-level generating extension [Glü91]: *incremental self-application* and *multiple self-application*.

Consider a three-staged program, namely a meta-interpreter `mint`, that takes a language definition `def`, a program `pgm` and its data `dat` as input. Let `def` be written in the definition language D , let `pgm` be written in language P (defined by `def`), and let `mint` be written in some language L . The equational definition of `mint` is

$$\llbracket \text{mint} \rrbracket_L \text{def pgm dat} = \llbracket \text{def} \rrbracket_D \text{pgm dat} = \llbracket \text{pgm} \rrbracket_P \text{dat} = \text{out}$$

What we look for is the three-level generating extension `mint-cogen` of the metainterpreter `mint` to perform the computation in three stages.

$$\begin{aligned}
\text{compiler} &= \llbracket \text{mint-cogen} \rrbracket_L \text{def} \\
\text{target} &= \llbracket \text{compiler} \rrbracket_L \text{pgm} \\
\text{out} &= \llbracket \text{target} \rrbracket_L \text{dat}
\end{aligned}$$

² A compiler generator `cogen` is a three-level generating extension of a specialist `spec`.

The program `target` is an L-program which, when applied to `dat`, returns the final result `out`. The two-level generating extension `compiler` is a program which, when given a P-program `pgm`, returns an L-program `target` and is thus a P→L-compiler. The three-level generating extension `mint-cogen` is a program which, when applied to `def`, yields `compiler` and is thus a compiler generator.

Incremental Self-Application. A three-level generating extension can be constructed in two steps, using the compiler generator `cogen` (or using the Futamura projections together with a self-applicable specializer `spec`): (1) the metainterpreter `mint` is converted to an ‘auxiliary’ generating extension `gen-aux` with two inputs; (2) the generating extension `gen-aux` is converted to `mint-cogen'` with one input.

$$\begin{aligned} (1) \quad \text{gen-aux} &= \llbracket \text{cogen} \rrbracket_L \text{mint 'SSD'} \\ (2) \quad \text{mint-cogen}' &= \llbracket \text{cogen} \rrbracket_L \text{gen-aux 'SD'} \end{aligned}$$

It can easily be verified that `mint-cogen'` is a three-level generating extension by composing the above equations and using the equational definition of `cogen`.

$$\text{out} = \llbracket \llbracket \llbracket \text{mint-cogen}' \rrbracket_L \text{def} \rrbracket_L \text{pgm} \rrbracket_L \text{dat}$$

In general, we need n steps to convert an n -stage program into its n -level generating extension. In practice, incremental generation of generating extensions is more difficult than the respecialization of residual programs (see Sect. 2.2) because `cogen` is not just applied to an ‘ordinary’ residual program but to a program generator. Binding-time improvements, should they be necessary, would have to be performed on a machine-generated program generator. Most compiler generators are not geared toward the respecialization of their generating extensions. Note that incremental generation of multi-level generating extensions is the only possibility if we are given only a conventional compiler generator `cogen`.

Multiple Self-Application. Given a self-applicable specializer `spec` there is another way to generate multi-generating extensions: by multiple self-application. This requires up to three self-applications in our example (as opposed to the Futamura projections which require only two self-applications). We now state four metasystem-transition (MST) formulas that specify the generation of a target program `target''`, a compiler `compiler''`, a compiler generator `mint-cogen''` and a compiler-compiler generator `cocogen''`. The correctness of the formulas can be verified using the equational definition of `spec`. For notational convenience we distinguish between the specializers and assume that each has the corresponding arity (otherwise they are identical).

$$\begin{aligned} \text{target}'' &= \llbracket \text{spec1} \rrbracket_L \text{mint 'SSD' def pgm} \\ \text{compiler}'' &= \llbracket \text{spec2} \rrbracket_L \text{spec1 'SSSD' mint 'SSD' def} \\ \text{mint-cogen}'' &= \llbracket \text{spec3} \rrbracket_L \text{spec2 'SSSSD' spec1 'SSSD' mint 'SSD'} \\ \text{cocogen}'' &= \llbracket \text{spec4} \rrbracket_L \text{spec3 'SSSSDD' spec2 'SSSSD' spec1 'SSSD'} \end{aligned}$$

It can easily be verified that the program `mint-cogen''` is a three-level generating extension by composing the above equations and using the equational definitions of the corresponding specializers. The three-level generating extensions `mint-cogen''` can be obtained either from the 3rd or the 4th MST-formula using

`cocogen''`. Note that `cocogen''` accepts programs written in L, while `mint-cogen''` accepts definitions written in D, the language defined by `mint`.

$$\text{out} = \llbracket \llbracket \llbracket \text{mint-cogen''} \rrbracket_L \text{def} \rrbracket_L \text{pgm} \rrbracket_L \text{dat}$$

Multiple self-application has two fundamental problems [Glü91]:

- *Generation time problem.* Assume that t is the run-time of a program p , then $t * k^n$ is the time required to run p on a tower of n self-interpreters, where k is the factor of their interpretive overhead. Since most non-trivial, self-applicable specializers incorporate a self-interpreter for evaluating static program pieces, the run-time of multiple self-application grows exponentially with the number of self-applications.
- *Generator size problem.* Each level of self-application adds one more layer of code generation to the produced generating extension, i.e. code is generated that generates code-generating code that generates code-generating code, and so on. The result is another variant of the notorious encoding problem in self-application³: it may lead to an exponential growth of the size of the produced generating extensions (in the number of self-applications).

To avoid the theoretical limitations and the practical problems of conventional specialization tools we are going to develop an efficient multi-level compiler generator in the remainder of this paper.

3 Construction Principles

First we give a brief review of conventional off-line partial evaluation (a comprehensive presentation can be found in [JGS93]) and then state two observations that are the starting point for our “multi-cogen approach”. We use Scheme as our presentation language.

3.1 Conventional Off-Line Partial Evaluation

In *off-line partial evaluation* the transformation process is guided by a *binding-time analysis* performed prior to the specialization phase. The result of the binding-time analysis is a program in which all operations are annotated as either *static* or *dynamic*. Operations annotated as static are performed at specialization time, while operations annotated as dynamic are delayed until run-time (i.e. residual code is generated). Binding-time annotations can be represented conveniently using a *two-level syntax* [NN92], e.g. by marking all dynamic operations with an underscore `_op`, while leaving all static operations unchanged `op`.

Example 1. Assume that the first two parameters of a program computing the inner product of two vectors are known in advance: the dimension `n` and the first vector `v`. We use the function `(ref i v)` to access the i 'th element of a vector `v` and leave their internal representation unspecified. The result of the binding-time analysis is shown in Fig. 2.

```

(define (iprod n v w)      ; n,v:static  w:dynamic
  (if (> n 0)
      (+ (* (lift (ref n v)) (_ref (lift n) w))
          (iprod (- n 1) v w))
      (lift 0)))

```

Figure 2: A two-level program of the inner product

Lifting is necessary when static values appear in a dynamic context. The operation `lift` converts static values into corresponding pieces of program code [Rom88, Mog88]. For example, the static expressions `n` and `(ref n v)` need to be lifted.

In general, specialization will fail to terminate if all function calls in a source program are unfolded unconditionally. A standard method to avoid this problem is the insertion of specialization points. A *specialization point* is a call to a named function f that is not unfolded, but treated in the following way during specialization: if the tuple (f, \overline{arg}) with function name f and static arguments \overline{arg} has been specialized before then a call to the already specialized function is inserted; otherwise the tuple (f, \overline{arg}) is recorded in a *memoization table* and the function f is specialized with respect to the static arguments \overline{arg} . This method is known as *polyvariant program point specialization* because the same program point may be specialized with respect to different static arguments. Two well-known methods for automatically inserting specialization points are *inductive variables* [Ses88], and *dynamic conditionals* [BD91].

3.2 The Starting Point

Observation 1: Annotated Program = Generating Extension [Hol89].

An annotated program p_{ann} can be viewed as a generating extension of program p . It is only a question of the language in which one considers p_{ann} as program: static operations `op` can be executed, while dynamic operations `_op` generate program code.

Example 2. The dynamic operations in an annotated program can be defined as shown in Fig. 3. The backquote notation of Scheme is convenient for constructing list structures. For example `(list '* e1 e2) ≡ `(* ,e1 ,e2)`⁴.

```

(define (_+ e1 e2) `( + ,e1 ,e2))
(define (_* e1 e2) `( * ,e1 ,e2))
(define (_ref e1 e2) `( REF ,e1 ,e2))
(define (lift e) `( QUOTE ,e))

```

Figure 3: Two-level code generation

A specialized version of the inner product (Fig. 4) can be obtained by evaluating the annotated program (Fig. 2) together with the definitions of the dynamic

³ The other encoding problem is associated to typed languages: types in programs need to be mapped into a universal type in a partial evaluator.

⁴ Upper and lower case forms of identifiers are not distinguished in Scheme. For convenience we use upper case letters when we generate code, e.g. `(REF ,e1 ,e2)`. The expression `'datum` in Scheme is just an abbreviation for `(QUOTE datum)`.

operators (Fig. 3). The program is specialized with respect to dimension $n=3$ and vector $\mathbf{v}=(7\ 8\ 9)$, while the dynamic variable \mathbf{w} is bound to itself. The generation of `define` expressions using specialization points will be explained later.

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

Figure 4: A specialized version of the inner product

Observation 2: Multi-Level Annotation. The two-level static/dynamic annotation of a program is, obviously, a special case of a more general multi-level annotation where binding-time values $0, 1, \dots, n$ indicate at what time an operation can be reduced.

We conclude: if a two-level program can be viewed as generating extension, then a multi-level program can be viewed as multi-level generating extension and the corresponding multi-level binding-time analysis can be used to generate the multi-level generating extensions.

Note that annotated programs can be expressed very elegantly in Scheme, while other programming languages may require an extra translation step in order to obtain syntactically correct multi-level programs. However, this does not change the underlying principles of our approach.

4 Efficient Multi-Level Generators

We present an automatic program generator for multi-level generating extensions based on the two previous observations. More specifically, we generalize the following techniques of conventional partial evaluation: binding-time analysis, lifting, specialization points, memoization, and code generation. The multi-level generating extensions will be discussed in the next section.

4.1 The Source Language

The source language is a higher-order subset of Scheme. A program is a sequence of function definitions with expressions composed of variables x , constants c , primitive operators op , conditionals `if`, procedure calls, let-expressions, abstractions $\lambda x.e$ and applications $e @ e^5$. Programs written in this subset have the same semantics as in Scheme. The abstract syntax is defined in Fig. 5.

4.2 Multi-Level Binding-Time Analysis

The multi-level binding-time analysis formalizes the intuition that early values may not depend on late values. It is a generalization of the two-level binding-time analysis, e.g. [JGS93]. The result of the multi-level binding-time analysis

⁵ Without loss of generality we assume that abstractions have only one parameter.

$P \in \text{Program}; D \in \text{Definition}; e \in \text{Expression}; c \in \text{Constant};$ $x \in \text{Variable}; f \in \text{ProcName}; op \in \text{Primitive}$ $P ::= D \dots D$ $D ::= (\text{define } (f \ x \dots x) \ e)$ $e ::= x \mid c \mid (\text{if } e \ e \ e) \mid (f \ e \dots e) \mid (op \ e \dots e) \mid (\text{let } ((x \ e)) \ e) \mid \lambda x. e \mid e \ @ \ e$

Figure 5: Abstract syntax of the higher-order Scheme subset

is a *multi-level program* in which all operations are annotated with *binding-time values*. *Multi-level lifting* can be used to delay values during specialization. The program obtained by erasing all annotations in a multi-level program P is called the *underlying program* and we denote it by $|P|$. We specify the multi-level binding-time analysis using a type system.

Binding-Time Values. A binding-time value $t \in 0, 1, \dots, \mathbf{max}$ indicates at what time an operation can be reduced. The binding-time value \mathbf{max} is the maximal binding-time of the specific problem considered. Each operation op in a multi-level program is annotated with its binding-time value: op_t . From now on we refer to expressions with binding time $t = 0$ as *static*, and to expressions with binding time $t \geq 1$ as *dynamic*. The meta-variables r, s, t range over binding-time values. For every program P that we want to specialize we assume that the binding-time values of its input are given (in the form of a *binding-time pattern*), as well as the maximal binding-time value \mathbf{max} .

Lifting. The multi-level lift operator \mathbf{lift}_t^s is defined as follows: it takes t specializations before the value of its argument becomes known and $t + s$ specializations before the value is made available to the enclosing expression ($s > 0, t \geq 0$). Thus, the binding-time value of an expression $(\mathbf{lift}_t^s \ e)$ is the sum of the values s and t . We require that all \mathbf{lift}_t^s operators inserted in a multi-level program are maximized with respect to s in order to avoid redundant lift annotations. The following equivalence holds: $(\mathbf{lift}_t^{r+s} \ e) \equiv (\mathbf{lift}_{t+s}^r \ (\mathbf{lift}_t^s \ e))$.

The conventional lift operator (Sect. 3.1) is a special case of multi-level lifting where the values for s and t are constants: \mathbf{lift}_0^1 .

Binding-Time Types. A type τ is well-formed binding-time type if $\vdash \tau : t$ is derivable from the rules in Fig. 6. The first rule states that b^t is binding-time type with binding-time value t and that this binding-time value is not greater than \mathbf{max} . The type b^t is assigned to *base type* objects. The rule for function types $\tau_1 \rightarrow^t \tau_2$ states that their binding-time value is the binding-time value of the arrow and that the binding-time values of the argument and result type must not be smaller than the binding-time value of the function type. Every type τ represents a binding-time value t , and we define a function from types to binding time values: $\|\tau\| = t$ **iff** $\vdash \tau : t$. It is easy to show that for all binding-time types τ we have $\|\tau\| \leq \mathbf{max}$. The binding-time value of an expression e in a multi-level program is equal to the binding-time value $\|\tau\|$ of its binding-time type τ .

We define an equivalence relation on binding-time types that allows us to handle programs with potential type errors (function values used as base values, base values used as functions). We defer such errors to the latest possible binding-time \mathbf{max} , i.e. when the final result is produced. The equality is defined by

$$\vdash b^{\mathbf{max}} \rightarrow^{\mathbf{max}} b^{\mathbf{max}} \doteq b^{\mathbf{max}}$$

$$\boxed{\frac{t \leq \mathbf{max}}{\vdash b^t : t} \quad \frac{\vdash \tau_1 : s_1 \quad \vdash \tau_2 : s_2 \quad s_i \geq t}{\vdash \tau_1 \rightarrow^t \tau_2 : t}}$$

Figure 6: Binding-time types

and the usual rules and axioms for equality (symmetry, reflexivity, transitivity and compatibility with arbitrary contexts).

Well-Annotated Multi-Level Programs. The typing rules for well-annotated multi-level programs are given in Fig 7. Most of the typing rules are straightforward generalizations of the corresponding two-level rules, e.g. [JGS93]. For instance, the rule $\{If\}$ for `if`-expressions annotates the `if` with the binding-time 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 requires that the test expression has a first-order type.

The rule $\{Prim\}$ requires that all higher-order arguments of primitive operators have binding-time \mathbf{max} . This is a necessary and safe approximation since we assume nothing about the type of a primitive operator (the same restriction is used in the binding-time analysis of Simlix [Bon91]). The rule $\{Lift\}$ allows lifting only of first-order values. The rules $\{Def\}$ and $\{Prog\}$ type definitions and programs, respectively. Since Scheme procedures (named functions) are uncurried we write their type as $\bar{t} \rightarrow t$ where \bar{t} is a possibly empty sequence of binding-time types. The binding-time value on the arrow is left out, since the decision whether to unfold (reduce) such function is not taken during binding-time analysis.

The analysis is monovariant since all calls to the same function have the same binding-time pattern (this is ensured by appropriate lift-operators).

Definition (well-annotated completion). *A multi-level program P' is a well-annotated completion of a program P , if $|P'| = P$ and if for given a binding-time value \mathbf{max} and binding-time types (well formed with respect to \mathbf{max}) $\bar{\tau}_0, \bar{\tau}_1 \rightarrow \tau_1, \dots, \bar{\tau}_l \rightarrow \tau_l$ the following judgement can be derived*

$$\vdash P' : \{f_0 : \bar{\tau}_0 \rightarrow b^{\mathbf{max}}, \dots, f_l : \bar{\tau}_l \rightarrow \tau_l\}$$

where f_0 is the goal function of P' and $\bar{\tau}_0 = (b^{t_0}, \dots, b^{t_k})$ where (t_0, \dots, t_k) is the binding-time pattern for the input to f_0 .

The goal of the multi-level binding-time analysis is to determine a well-annotated completion P' of a program P which is, preferably, the ‘best’. A completion is the best if the binding time for every subexpression e in P is less than or equal to the binding time of e in any other well-annotated completion of P . We conjecture that, as in the two-level case, there always exists a best completion.

The correctness of two-level binding-time analysis schemes has been shown by several authors, e.g. [Wan93, Hat95]. Similar proof methods can be used in the multi-level case (beyond the scope of this paper).

$\{Const\} \Gamma \vdash c : b^0$	$\{Var\} \frac{x:\tau \text{ in } \Gamma}{\Gamma \vdash x:\tau}$	$\{Prim\} \frac{\Gamma \vdash e_i : b^t}{\Gamma \vdash (op_t e_1 \dots e_k) : b^t}$
$\{If\} \frac{\Gamma \vdash e_1 : b^t \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \quad \ \tau\ \geq t}{\Gamma \vdash (\mathbf{if}_t e_1 e_2 e_3) : \tau}$	$\{Lift\} \frac{\Gamma \vdash e : b^t \quad s > 0}{\Gamma \vdash (\mathbf{lift}_t^s e) : b^{t+s}}$	
$\{Call\} \frac{\Gamma \vdash e_i : \tau_i \quad f : \tau_1 \dots \tau_k \rightarrow \tau \text{ in } \Gamma}{\Gamma \vdash (f e_1 \dots e_k) : \tau}$	$\{\doteq\} \frac{\Gamma \vdash e : \tau \quad \vdash \tau \doteq \tau'}{\Gamma \vdash e : \tau'}$	
$\{Lett\} \frac{\Gamma \vdash e : \tau \quad \Gamma \{x : \tau\} \vdash e' : \tau' \quad \ \tau'\ \geq \ \tau\ }{\Gamma \vdash (\mathbf{let}_{\ \tau\ } ((x e)) e') : \tau'}$		
$\{Abs\} \frac{\Gamma \{x : \tau_x\} \vdash e : \tau}{\Gamma \vdash \underline{\lambda}_s x. e : \tau_x \rightarrow^s \tau}$	$\{App\} \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow^s \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 @_s e_2 : \tau_1}$	
$\{Def\} \frac{\Gamma \{x_1 : \tau_1, \dots, x_k : \tau_k\} \vdash e : \tau}{\Gamma \vdash (\mathbf{define} (f x_1 \dots x_k) e) : \tau_1 \dots \tau_k \rightarrow \tau}$		
$\{Prog\} \frac{\{f_1 : \bar{\tau}_1 \rightarrow \tau_1, \dots, f_l : \bar{\tau}_l \rightarrow \tau_l\} \vdash (\mathbf{define} (f_i \bar{x}_i) e_i) : \bar{\tau}_i \rightarrow \tau_i}{\vdash P : \{f_1 : \bar{\tau}_1 \rightarrow \tau_1, \dots, f_l : \bar{\tau}_l \rightarrow \tau_l\}}$		

Figure 7: Typing rules for the multi-level binding-time analysis

Extensions. Other multi-level binding-time analyses can be defined in a similar way. For example, using also *recursive types* $\mu\alpha.\tau$ would allow more liberal annotations (earlier binding-time values for expressions). This can be done without changing the rules in Fig. 7. It is sufficient to extend the equality on binding-time types in such a way that all types that have the same regular type are equal (according to the translation described in [CC91]).

4.3 Insertion of Multi-Level Specialization Points

We chose dynamic conditionals and dynamic abstractions for inserting specialization points because this strategy is straightforward, surprisingly effective in practice, and easily generalized to the multi-level case. We will only explain this for conditionals.

- If a test has the binding time $t = 0$ then the test can be decided at specialization time. No specialization point is inserted.
- If a test has the binding time $t \geq 1$ then the test can not be decided at specialization time. To avoid the risk of infinite unfolding, a multi-level specialization point `multi-memo` is inserted and the dynamic conditional is made into a named function whose parameters are the free variables of the conditional and whose body is the entire conditional.

As in the two-level case, this strategy avoids infinite unfolding at specialization time, but does not avoid infinite specialization if static data varies unboundedly under dynamic control. Strategies for automatically guaranteeing termination without being overly conservative are a topic of current research.

Multi-Level Specialization Points. A multi-level specialization point is more general than its traditional counterpart:

1. It is sufficient to divide the argument expressions of a specialization point into two groups ($t = \textit{static}$, $t = \textit{dynamic}$) in the two-level case, while the argument expressions may have several different binding times $t \in \{0, 1, \dots\}$ in the multi-level case.
2. The binding time of a dynamic test can only be $t = \textit{dynamic}$ in the two-level case, while a dynamic test may have any binding time $t \in \{1, 2, \dots\}$ in the multi-level case.

A multi-level specialization point `multi-memo` has the following arguments: the binding time t of the dynamic test, the name f of the function, and the *groups* g of f 's arguments with equal binding times. Each group is tagged with the binding time t of the arguments it contains. Empty groups are omitted, so the total number of groups is limited by the number of arguments of f (and does not depend on the number of specialization levels). We assume that the groups are ordered w.r.t. increasing binding time values. The use of multi-level specialization points in generating-extensions is explained in Sect. 5. See Fig. 8 for the syntax of multi-level specialization points.

4.4 Implementation

We implemented the multi-level generator for the Scheme subset. The efficiency of the implementation depends on efficiency of the multi-level binding-time analysis and the insertion of specialization points.

Based on the constraint-based binding-time analysis for higher-order languages [Hen91, BJ93] we developed an efficient multi-level binding-time analysis which has an almost-linear complexity [GJ95]. In particular, for a first-order language (without partially static structures) the multi-level binding-time analysis can be reduced to a *graph reachability problem* that runs in linear time $\mathcal{O}(n)$, where n is the number of edges in the dependency graph (bounded by the size of the source program).

The insertion of specialization points is bounded by the number of conditionals and thus by the size of the source program.

Limitations. In the present implementation of our multi-level generator we chose a few ‘poor-man’s’ solutions (since our aim was to study the principles underlying multiple program specialization): (i) Unrestrained unfolding may result in programs that terminate more often than the original program. Termination-preserving methods for two-level partial evaluation can be found in [Bon91]. (ii) All function calls, except memoization points, are unfolded. This may lead to computation duplication. Methods for preventing this problem are well-known, e.g. inserting let-expressions [Bon91], and can easily be generalized to the multi-level case.

5 Efficient Multi-Level Generating Extensions

Representation of Multi-Level Programs. We choose the same approach as in Sect. 3.2: we do not put annotations on static expressions ($t = 0$), only on dynamic expressions ($t \geq 1$). Thus, static expressions can be evaluated directly by the underlying implementation (their semantics is the same as in Scheme).

The concrete syntax is shown in Fig. 8. We could, in principle, provide an interpreter to run multi-level programs, but this would be less efficient. Dynamic expressions require three kinds of operations:

1. *Code generation*: functions that emit program code,
2. *Lifting*: a function that turns a value into a piece of code.
3. *Memoization*: a function that maintains the memoization table and specializes functions if necessary.

$ \begin{aligned} e ::= & x \mid c \mid (f \ e \dots e) \mid (\text{if } e \ e \ e) \mid (\text{op } e \dots e) \\ & \mid (\text{let } ((x \ e)) \ e) \mid (\text{lambda } (xs) \ e) \mid (e \dots e) \\ & \mid (\text{_var } t \ x) \mid (\text{lift } t \ e) \mid (\text{_ 'if } t \ e \ e \ e) \mid (\text{_ 'op } t \ e \dots e) \\ & \mid (\text{_let } t \ x \ e \ e) \mid (\text{_lambda } t \ (x \dots x) \ e) \mid (\text{_app } t \ e \dots e) \\ & \mid (\text{multi-memo } t \ f \ g \dots g) \\ g ::= & (t \ e \dots e) \end{aligned} $
--

Figure 8: Concrete syntax of multi-level programs

Structure. A multi-level generating extension consist of two parts: the multi-level program and a library that includes the code-generating functions, lifting, and memoization⁶. No extra interpretive overhead is introduced since the definitions in the library are linked with the multi-level program at loading/compile-time. The library adds only a constant size of code to the multi-level program (in contrast to [Hol89] who used macro-extensions). Thus, the size of the multi-level generating extension depends only on the size of the multi-level program (and the static data) and is independent of the number of specialization levels.

Generic Code Generation. A generic code-generating function `_` can be used that takes three arguments (Fig. 9): an operator `op`, a binding-time value `t`, and the arguments `as` (already pieces of code). Generic code generation works as follows: if the binding time `t` equals 1 then the function produces an executable expression for `op`; otherwise it reproduces a code-generating function for `op` with binding-time argument `t` decreased by 1. Repeatedly applying a code-generating function corresponds to a ‘count-down’ until the binding-time argument reaches 1 which means that the arguments for `op` will be static next time the program runs.

Multi-Level Lifting. We use the following translation for the lift operator:

$$\underline{\text{lift}}_0^s e \equiv (\text{lift } s \ e) \quad \text{and} \quad \underline{\text{lift}}_t^s e \equiv (\text{_ lift } t \ s \ e)$$

In the first case the argument `e` is static, but needs to be lifted `s` times. In the second case the argument `e` has the dynamic binding time `t` and the lift operation has to be delayed `t` times, using the code generating function `_`, before lifting the value `s` times. The `lift` operation (Fig. 9) is similar to the code generating function: it counts the binding-time value `s` down to 1 before releasing the value `val`.

⁶ Generating extensions for languages with side-effects, such as C, require an additional management of the static store in order to restore previous computation states, e.g. when specializing the branches of a dynamic conditional (see [And94]).

The code-generation operations for abstraction and application cannot in a simple way be expressed by using the generic operation `_`. Application does not have a Scheme operation `app`, say, associated with it so we cannot just write something like `(_ 'app e1 e2)`, so we have chosen to define a special code-generating function `_app` for application. This is shown in Fig. 9. The problem with abstractions is another: the formal parameter of an abstraction is free in the body of the abstraction and when the body is evaluated to some code the formal parameter must evaluate to itself. This is ensured by generating code for variables that, similarly to lifting, counts down to 1 before “releasing” the variable. If `x` is a variable bound in an expression (let-expression or abstraction) with binding time `t` then the following code is generated `(_var t x)`. The code generating function `_let` for let-expressions is similar to `_lambda`.

To ensure safe unfolding and prevent variable capturing trivial let-expressions, i.e. `(let ((x x)) e)`, are for all formal parameters in abstractions and procedures. This is similar to what Similix does.

```

(define (_ op t . as)                (define (lift s val)
  (if (= t 1)                        (if (= s 1)
    `(.op . ,as)                      `(QUOTE ,val)
    `(_ (QUOTE ,op) ,(- t 1) . ,as))) `(_LIFT ,(- s 1) (QUOTE ,val))))

(define (_lambda t xs e)            (define (_app t e . es)
  (if (= t 1)                          (if (= t 1)
    `(LAMBDA ,xs ,e)                  `(,e . ,es)
    `(_LAMBDA ,(- t 1) (QUOTE ,xs) e))) `(_APP ,(- t 1) ,e . ,es)))

(define (_let t x e1 e2)            (define (_var t x)
  (if (= t 1)                          (if (= t 1)
    `(LET ((,x ,e1)) ,e2)              x
    `(_let ,(- t 1) (QUOTE ,x) ,e1 ,e2))) `(_var ,(- t 1) (QUOTE ,x))))

```

Figure 9: Multi-level code-generation and lifting

Example 3. Assume that the program computing the inner product has three different binding times: dimension `n`:0, vector `v`:1, and vector `w`:2. The multi-level program of the inner product is shown in Fig. 10.

```

(define (iproduct3 n v w)
  (if (> n 0)
    (_ '+ 2 (_ '* 2
              (_ 'lift 1 1 (_ 'ref 1 (lift 1 n) v))
              (_ 'ref 2 (lift 2 n) w))
      (iproduct3 (- n 1) v w))
    (lift 2 0)))

```

Figure 10: A three-level program of the inner product

Multi-Level Memoization. A multi-level specialization point may give rise to several function specializations over time, as opposed to conventional partial evaluation where all specialization points are specialized only once. The function has two parts:

1. The function maintains a memoization table (in the form of an association list called a memo-list) of previously encountered tuples (f, as_0) . If the tuple (f, as_0) has not been specialized before, then the function f is specialized with respect to the static arguments and the tuple is recorded in the memo-list together with the name of the new function. Otherwise, a call to the already specialized function is inserted. This is similar to conventional specialization.
2. A multi-level specialization point reproduces itself after decreasing the binding-time values t and the binding-time tags by 1. It disappears if its binding-time value t becomes 1.

The function `multi-memo` is defined in pseudo-code in Fig. 11. The multi-level specialization function `multi-memo` takes the following arguments: a binding-time value t , a function name f , and lists of argument groups that are tagged with binding-time values $0, 1, 2, \dots$ where the list with static arguments has the tag 0 . We assumed that there is an argument group for every binding-time value $0 \dots t$, but in the actual implementation only non-empty argument groups need to be present. The multi-level specialization is polyvariant and depth-first.

A few auxiliary functions are used in the definition: the function `notSeenB4` checks whether the current specialization point exists in the memo-list; `addToMemoList` adds a new specialization point to the memo-list and generates a new residual function name which can be retrieved by function `getResName`. The function `updateMemoList` updates the memo-list with the specialized body of f . The function `call` applies a function to a list of arguments.

```

multi-memo [t f (0 as0) (1 as1)...(t ast)] =
  let pp = `(,f ,as0) in
    if notSeenB4 pp then
      addToMemoList pp;
      updateMemoList pp (call f `(,as0...,ast))
    endif;
  let fn = getResName pp in
    if t=1 then `(,fn . ,ast)
    else
      `(multi-memo ,(t-1) ,fn (0 ,as1)...,(t-1) ,ast))
    endif

```

Figure 11: Multi-level memoization

Example 4. Consider a multi-level specialization point with a binding-time value 4 and four groups of arguments tagged with the binding-time values 0, 2, 3, and 4 respectively. The multi-level specialization of f develops in several steps.

```

(multi-memo 4 f (0 ea) (2 eb ec) (3 ed) (4 ee))
  (multi-memo 3 fa (1 eb ec) (2 ed) (3 ee))
    (multi-memo 2 fa (0 eb ec) (1 ed) (2 ee))
      (multi-memo 1 fabc (0 ed) (1 ee))
        (fabcd ee)

```

First, the initial generating extension `p-mgen` specializes the function `f` with respect to the static argument e_a and the specialization point is kept. The next generating extension `p-mgen0` reproduces the specialization point without specializing `fa` because it receives no static arguments. The generating extension `p-mgen1` specializes the function `fa` with respect to the static arguments e_b and e_c . Finally, the last generating extension `p-mgen2` specializes the function with respect to the static argument e_d and produces a direct call to `fabcd`.

6 Results

The main advantages of the multi-level generator are the speed of the generation process and the compact size of the generated multi-level generating extensions. We demonstrate some of its advantages by comparing our approach with multiple self-application and solving a problem that we previously attacked unsuccessfully with a state-of-the-art partial evaluator.

6.1 Comparison with Multiple Self-Application

Some experiments with multiple self-application were reported in [Glü91], and we repeated these experiments with our multi-level generator in order to make a direct comparison: splitting a program for matrix transposition into 2 – 5 levels. The function `transpose` transposes a $5 \times n$ matrix where each of its five parameters takes one line of the original matrix. The algorithm can be transformed into a generating extension `p-mgen` that takes the first line of the matrix and generates a generating extension `p-mgen0` that takes the second line ... until the last generating extension generates a residual program that takes the last line and returns the transposed matrix as final result.

The differences in generation time (generator size) between 2 and 5 binding-time levels are drastically reduced by the multi-level compiler generator: from 1:9000 to 1:1.8 in generation-time, and from 1:100 to 1:2 in generator size. The absolute numbers for multiple self-application of a partial evaluator [Glü91] range from less than 100 ms to over 15 minutes (on a different hardware), and the size of the produced generating extensions ranges from 112 to over 12000 cons cells. The generation times using our multi-level compiler generator are given in ms and the sizes of the produced generating extensions as the number of cons cells needed to represent the program (Fig. 12).⁷ The times t_{bita} , t_{ann} and t_{total} are the times for doing the multi-level binding-time analysis, the multi-level program annotation (including the insertion of specialization points), and the total run time of the compiler generator, respectively. Fig. 13 shows the size of the generating extensions `p-mgen` to `p-mgen3` when applied to a 5×3 matrix taking one line at a time.

6.2 Generating a Compiler Generator

A multi-level generator can transform multi-staged programs directly into their generating extension. In particular, we used the multi-level generator to convert a

⁷ Times are given in cpu-seconds using Chez Scheme 3.2 and a Sparc Station 2/Sun OS 4.1 (excluding time for garbage collection, if any).

levels	2	3	4	5
t _{hta} /ms	5.5	5.4	5.1	5.1
t _{ann} /ms	8.0	11.0	15.1	19.5
t _{total} /ms	13.5	16.4	20.2	24.6
size/cells	180	236	295	357

Figure 12: Performance results for the transpose example

extension	p-mgen	p-mgen ₀	p-mgen ₁	p-mgen ₂	p-mgen ₃
size/cells	357	810	608	427	150
time/ms	28	19	12	5.6	0.1

Figure 13: Sizes and run-times of the generating extensions

two-level driving interpreter (the one defined in [GJ94]) into a compiler generator that is capable of a form of supercompilation: given a naive pattern matcher the new compiler generator returns a generating extension that, given a fixed pattern, produces specialized matchers that are equivalent to those generated by the Knuth, Morris, and Pratt algorithm (cf. [SGJ94]). We *failed* to achieve the same goal by incremental generation (Sect. 2.3) using Similix, a state-of-the-art partial evaluator for Scheme. It became impossible to respecialize the generating-extension `gen-aux` (see Sect. 2.3) with Similix’s `cogen` because the generating extension was produced in continuation-passing style where static and dynamic values flow together [GJ94]. The necessary (manual) binding-time improvement of the machine-generated generating-extension turned out to be very difficult for non-trivial programs. The multi-level generator `mcogen` solved this problem elegantly — in *one* step!

The size of the two-level driving interpreter is 1494 cons cells, the size of the generated compiler generator is 3364 cons cells and the total generation time was 1.77 s.

7 Related Work

Two-Level Generators. The first hand-written compiler generator based on partial evaluation principles was, in all probability, the system *RedCompile* for a dialect of Lisp [BHOS76]. Romanenko [Rom88] gave transformation rules that convert annotated first-order programs into their two-level generating extensions.

Holst [Hol89] was the first to observe that the annotated version of a program is already the generating extension. What Holst called “syntactic currying” is now known as the “*cogen approach*” [BW94]. Holst and Launchbury [HL92] studied this approach for a small LML-like language in order to overcome the notorious encoding problem associated with typed languages in self-application (types in programs need to be mapped into an universal type in the partial evaluator).

Birkedal and Welinder [BW93] used the “*cogen approach*” for the Core Standard ML language, Bondorf and Dussart [BD94] combined this approach with

cps-based specialization for the λ -calculus, and Andersen [And94] developed a generating extension generator for the ANSI C language.

Self-Application. Two methods have been used for improving the performance of self-application: *actions trees* [CD90], an off-line method that compiles the information obtained by the binding-time analysis into directives for the partial evaluator, and the *freezer* (metasystem jumps) [Tur89] in supercompilation, an on-line method that allows the evaluation of partially known expressions by the underlying implementation.

Both methods are less effective for multiple partial evaluation than the “multi-cogen approach” because (i) they do not avoid the transformation of dynamic expressions via the self-application tower (thus do not completely remove the interpretive overhead), and (ii) they do not address the generator size problem. However, we should note that a fair comparison with the freezer outside partial evaluation is difficult because of the different transformation paradigms.

Acknowledgments. Special thanks to Anders Bondorf for stimulating discussions about multiple self-application. Thanks to Dirk Dussart, Carsten K. Holst, Neil D. Jones, Torben Mogensen, Kristian Nielsen, Peter Sestoft, and Morten Welinder for comments.

References

- [And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
- [BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.
- [BD94] Anders Bondorf and Dirk Dussart. Improving cps-based partial evaluation: writing cogen by hand. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–9, Orlando, Florida, 1994.
- [BHOS76] Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.
- [BJ93] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on partial evaluation*, 11:315–346, 1993.
- [Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Revision of paper in ESOP’90, LNCS 432, May 1990.
- [BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. DIKU Report 93/22, DIKU, Department of Computer Science, University of Copenhagen, 1993.
- [BW94] Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings*, volume 844 of LNCS, pages 198–214, Madrid, Spain, 1994. Springer-Verlag.
- [CC91] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92:48–80, 1991.

- [CD90] Charles Consel and Olivier Danvy. From interpreting to compiling binding times. In N. D. Jones, editor, *ESOP '90*, volume 432 of *LNCS*, pages 88–105, Copenhagen, Denmark, 1990. Springer-Verlag.
- [Ers78] Andrei P. Ershov. On the essence of compilation. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.
- [GJ94] Robert Glück and Jesper Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis. Proceedings*, volume 864 of *LNCS*, pages 432–448, Namur, Belgium, 1994. Springer-Verlag.
- [GJ95] Robert Glück and Jesper Jørgensen. Constraint-based multi-level binding-time analysis of higher-order languages. Unpublished, 1995.
- [Glü91] Robert Glück. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 309–320, New Haven, Connecticut, 1991. ACM Press.
- [Glü94] Robert Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, 1994.
- [Hat95] John Hatcliff. Mechanically verifying the correctness of an off-line partial evaluator. In *PLILP'95*, LNCS. Springer-Verlag, 1995.
- [Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In John Hughes, editor, *Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts. LNCS 523*, pages 448–472. Springer-Verlag, August 1991.
- [HL92] Carsten Kehler Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. Working paper, 1992.
- [Hol89] Carsten Kehler Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.
- [JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
- [Mog88] Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.
- [NN92] Flemming Nielson and Hanne R. Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1992.
- [Rom88] Sergei A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.
- [Ses88] Peter Sestoft. Automatic call unfolding in a partial evaluator. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. North-Holland, 1988.
- [SGJ94] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Programming Languages and Systems — ESOP '94. Proceedings*, volume 788 of *LNCS*, pages 485–500, Edinburgh, Scotland, 1994. Springer-Verlag.
- [Tur89] Valentin F. Turchin. *Refal-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, Massachusetts, 1989.
- [Wan93] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, 1993.

This article was processed using the L^AT_EX macro package with LLNCS style