LISP and Symbolic Computation 10, 113–158 (1997)

An Automatic Program Generator for Multi-Level Specialization

ROBERT GLÜCK glueck@diku.dk
DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark

JESPER JØRGENSEN jesper@dina.kvl.dk
Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark

Abstract. Program specialization can divide a computation into several computation stages. This paper investigates the theoretical limitations and practical problems of standard specialization tools, presents multi-level specialization, and demonstrates that, in combination with the cogen approach, it is far more practical than previously supposed. 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 multi-level specialization by self-application. Our approach to multi-level specialization seems well-suited for applications where generation time and program size are critical.

Keywords: programming languages, program transformation, partial evaluation, generating extensions, binding-time analysis, functional languages, Scheme

1. Introduction

Stages of computation arise naturally in many programs, depending on the availability of data or the frequency with which data changes. Code for later stages can often be optimized based on data available in earlier stages. The division of programs into *two stages* has been studied intensively in the area of *partial evaluation* [8, 18, 35] to separate those program expressions that can be safely evaluated at specialization time from those that cannot. Partial evaluation can now be considered as one of the most advanced techniques for automatic program manipulation.

The main problem with standard partial evaluation, e.g., as presented in [37], is the need to specify the availability of data in terms of 'known' and 'unknown'. This two-point domain does not allow to specify multi-level transition points (e.g., "unknown 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 not only investigates multi-level specialization theoretically, but also presents the key ingredients for *efficient multi-level specialization*: a generalization of standard techniques for offline partial evaluation and their integration with the *cogen approach* [7, 31]. This requires several non-obvious extensions of standard partial evaluation techniques, such

as *multi-level generating extensions*, a generalization of Ershov's generating extensions [20]. The program generator which we designed and implemented for a higher-order subset of Scheme converts programs into very compact multi-level generating extensions that guarantee fast successive specialization.

The main payoff of this novel approach becomes apparent in the case of multiple self-application: experimental results show a remarkable reduction of generation time and generator size when compared to previous attempts of multi-level specialization by self-application [23]. The program generator permits multi-level specialization without the problems inherent in re-specialization of the output of automatic program specializers, which has been, among others, an obstacle to the generation of compilers from meta-interpreters (in the sense of interpreters taking a language definition, a program, and its data as input). Our approach to multi-level specialization seems well-suited for applications where generation time and program size are critical, such as run-time code generation or operating systems.

Our approach to multi-level specialization, which we call *multi-cogen approach*, shares the advantages of the traditional cogen approach [7, 31], but for multi-level 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, multi-level generating extensions are portable, stand-alone programs that are independent of the multi-level generator.

We claim that the multi-cogen approach scales up to other programming languages as evidenced by the fact that program generators for *two-level generating extensions* have been implemented successfully for a wide range of languages, including the λ-calculus [12], ML [5], Prolog [38], and ANSI C [1]. Recently our work has been extended to continuation-based partial evaluation [49] and closely related work has been initiated by several researchers including a language for hand-writing program generators [47], an algebraic description of multi-level lambda-calculi [43], and a framework for staged computation in modal logic [19].

1.1. This paper

This paper covers multi-level specialization starting from theoretical considerations and basic principles, to the design and implementation of the multi-level program generator and the assessment of experimental results. The paper falls into three parts:

- *Multi-level specialization.* We review standard tools for specialization, formulate multi-level specialization more precisely, and discuss the theoretical limitations and practical problems of standard specialization tools (Section 2).
- *Multi-level program generator.* We present the principles behind our design in a case study (Section 3), define the multi-level binding-time analysis, the heart of our multi-level program generator (Section 4), and extend the basic methods into a full implementation of

multi-level generating extensions with higher-order, polyvariant specialization (Section 5).
- *Experimental results.* In the last part, we discuss experimental results with applications that have been considered theoretically in earlier sections (Section 6) and conclude our presentation with comparative remarks (Section 7). Appendix A describes the meta-interpreter example in detail.

The main emphasis in this paper is on principles and methods. We explain the techniques for the construction of multi-level generators and generating extensions in sufficient detail to allow their implementation. We assume familiarity with the basic notions of partial evaluation, for example as presented in [37] or [35], Part II . The paper extends our earlier work [26] and draws upon results of [27].

2. Multi-level program specialization

In this section we review standard tools for specialization, formulate the properties of multi-level specialization more precisely, and discuss the theoretical limitations and practical problems of standard specialization tools.

Notation. For any program text, p, written in language L we let $[\![\texttt{p}]\!]_{\text{L}}$ `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 equality between applications shall always mean strong (computational) equivalence: either both sides of an equation are defined and equal, or both sides are undefined. For notational convenience we assume that all program transformers are L-to-L-transformers written in L; for multi-language specialization see [24]. The notation is adapted from [35].

2.1. Standard tools for specialization

Turning a general program into a specialized program can be conceived of as turning a one-stage computation into a two-stage computation, and performing the first stage. This is traditionally done using a program specializer or a so-called compiler generator.

2.1.1. Program specializer. Suppose `p` is a *source program* with input $\texttt{in}_0$ and $\texttt{in}_1$. Computation in one stage is described by

$$\texttt{out} = [\![\texttt{p}]\!]_{\text{L}}\ \texttt{in}_0\ \texttt{in}_1 \qquad \} \textit{ one stage} \tag{1}$$

Suppose $\texttt{in}_0$ is data known at stage one (*static*) and $\texttt{in}_1$ is data known at stage two (*dynamic*). Computation in two stages using a *program specializer* `spec` is described by

$$\left.\begin{array}{l} \texttt{p-res} = [\![\texttt{spec}]\!]_{\text{L}}\ \texttt{p}\ \texttt{`SD'}\ \texttt{in}_0 \\ \texttt{out} = [\![\texttt{p-res}]\!]_{\text{L}}\ \texttt{in}_1 \end{array}\right\} \textit{ two stages} \tag{2}$$

where the *binding-time classification* `'SD'` indicates the binding times of `p`'s input (we classify each argument of a source program as static, `'S'`, or dynamic, `'D'`, according

to its binding time). The result of specializing the source program p with respect to $\texttt{in}_0$ is a *residual program* p-res that returns the same result when applied to the remaining input $\texttt{in}_1$ as the source program p when applied to the input $\texttt{in}_0$ and $\texttt{in}_1$. We obtain the *mix-equation* [37], the correctness criterion of spec, by combining the above equations:

$$[\![\texttt{p}]\!]_{\mathrm{L}}\ \texttt{in}_0\ \texttt{in}_1 = [\![[\![\texttt{spec}]\!]_{\mathrm{L}}\ \texttt{p 'SD'}\ \texttt{in}_0]\!]_{\mathrm{L}}\ \texttt{in}_1 \tag{3}$$

All residual programs are faithful to the source program, but are often significantly faster. The main task of a specializer is to recognize which of p's computations can be precomputed at specialization time and which must be delayed until p-res's run time. For clarity we used $\texttt{in}_0$ and $\texttt{in}_1$, but it is easy to see that each of them can be replaced by a sequence of inputs.

2.1.2. Compiler generator. 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 as input and generates a program generator p-gen, called *generating extension* [20], as output. The task of p-gen is to generate a residual program p-res, given p's static data $\texttt{in}_0$. We call p-gen a *two-level* generating extension of p because it realizes p's computation in two stages.

$$\left.\begin{array}{lll} \texttt{p-gen} = [\![\texttt{cogen}]\!]_{\mathrm{L}} & \texttt{p 'SD'} & \\ \texttt{p-res} = [\![\texttt{p-gen}]\!]_{\mathrm{L}} & \texttt{in}_0 & \\ \texttt{out} = [\![\texttt{p-res}]\!]_{\mathrm{L}} & \texttt{in}_1 & \end{array}\right\} \textit{two stages} \tag{4}$$

Combing the equations in (1) and (4) we obtain an equational definition, the correctness criterion of cogen:

$$[\![\texttt{p}]\!]_{\mathrm{L}}\ \texttt{in}_0\ \ \texttt{in}_1 = [\![[\![[\![\texttt{cogen}]\!]_{\mathrm{L}}\ \texttt{p 'SD'}]\!]_{\mathrm{L}}\ \texttt{in}_0]\!]_{\mathrm{L}}\ \texttt{in}_1 \tag{5}$$

Specialization of p is now done in two steps: the compiler generator cogen produces p's generating extension p-gen which is then used to generate the residual program p-res. The generating extension p-gen produces p-res potentially much faster than the program specializer spec because p-gen is program generator specialized with respect to p.

2.2. Specialization pipelines

We discuss two forms of multi-level specialization, online and offline pipelines, and introduce the concept of a multi-level program generator and a multi-level generating extension.

2.2.1. Online specialization pipelines. Program specialization can do more than divide a computation into two stages. Suppose p is a source program with *n* arguments, where $\texttt{in}_0$ is data known at stage one, $\texttt{in}_1$ data known at stage two, and so on. Computation in one stage can be described by

$$\texttt{out} = [\![\texttt{p}]\!]_{\mathrm{L}}\ \texttt{in}_0 \cdots \texttt{in}_{n-1} \qquad \} \textit{one stage} \tag{6}$$

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

$$\left.\begin{aligned} \texttt{p-res}_1 &= [\![\texttt{spec}]\!]_{\mathrm{L}}\ \texttt{p}\ \text{`}\texttt{SDD}\cdots\texttt{D}\text{'}\ \texttt{in}_0 \\ \texttt{p-res}_2 &= [\![\texttt{spec}]\!]_{\mathrm{L}}\ \texttt{p-res}_1\ \text{`}\texttt{SD}\cdots\texttt{D}\text{'}\ \texttt{in}_1 \\ &\vdots \\ \texttt{p-res}_{n-1} &= [\![\texttt{spec}]\!]_{\mathrm{L}}\ \texttt{p-res}_{n-2}\ \text{`}\texttt{SD}\text{'}\ \texttt{in}_{n-2} \\ \texttt{out} &= [\![\texttt{p-res}_{n-1}]\!]_{\mathrm{L}}\ \texttt{in}_{n-1} \end{aligned}\right\}\ \textit{n stages} \tag{7}$$

In each stage the residual program $\texttt{p-res}_i$ obtained from the previous stage is specialized with respect to the next input $\texttt{in}_i$. The first argument of $\texttt{p-res}_i$ is always classified as static, the remaining arguments are always dynamic. In each stage the number of arguments decreases by one, thus the binding-time classification describes fewer arguments as the multi-level specialization proceeds. It is easy to verify that the specialization sequence is correct using the equational definition of `spec` in (3).

Alternatively, we can use a compiler generator `cogen` instead of `spec` to convert the residual program $\texttt{p-res}_i$ at stage $i+1$ into a generating extension $\texttt{p-gen}_i$. This may improve the performance of the specialization at stage $i+1$, e.g., if $\texttt{p-res}_i$ is specialized with respect to different input.

$$\left.\begin{aligned} \texttt{p-gen}_i &= [\![\texttt{cogen}]\!]_{\mathrm{L}}\ \texttt{p-res}_i\ \text{`}\texttt{SD}\cdots\texttt{D}\text{'} \\ \texttt{p-res}_{i+1} &= [\![\texttt{p-gen}_i]\!]_{\mathrm{L}}\ \texttt{in}_i \end{aligned}\right\}\ \textit{stage } i+1 \tag{8}$$

Multi-level specialization of `p` realized by one of the two methods above is a form of *online* specialization because specialization at stage $i+1$ can take all data $\texttt{in}_0 \cdots \texttt{in}_{i-1}$ into account that has been provided in earlier stages (regardless whether the specialization method used at stage $i+1$ is online or offline). Recall that the defining feature of *offline* specialization is that it does not take static data into account, only the binding-time classification of the input. Hence, we call this sequence of connected specializations an *online specialization pipeline*.

2.2.2. Assessment of online specialization pipelines. An online specialization pipelines has two main advantages:

- *Precision*: specialization at stage $i+1$ ($0 < i < n-1$) can take all *static values* into account that have been provided in earlier stages.
- *Flexibility*: at each stage we are free to choose *any* binding-time classification.

But the advantages of an online specialization pipeline are also its disadvantages. First, the precision gained by applying `spec`'s full transformation power to every 'intermediate' program $\texttt{p-res}_i$ has its price: specialization time. For example, program pieces that depend only on input that will be available at some later stage, may be re-analyzed in each of the preceding stages. It is also likely that these program pieces have been duplicated during an earlier stage, e.g., by polyvariant program point specialization.

Second, an online specialization pipeline may be too flexible in those cases where the order of input $\texttt{in}_0 \cdots \texttt{in}_{n-1}$ is fixed. Consider specializing a meta-interpreter with three

arguments: a language definition, a program and its data. Multi-level specialization does not make much sense unless the language definition is available before the program, and the program is available before its data.

Finally, it may be difficult to perform binding-time improvements on a machine-produced program, e.g., $\texttt{p-res}_i$ at stage $i+1$, because the structure of the original program p may have changed dramatically during the preceding stages (unless we have fully automatic methods for binding-time improvements). For example, a residual program resulting from specializing an interpreter with respect to an interpreted program will usually reflect the structure of both programs. This makes the overall behavior of an online specialization pipeline less predictable since the success of specialization at stage $i+1$ may depend on the static data provided in an earlier stage.

Moreover, how should one modify the original program p to avoid the need for binding-time improvements at a later stage? For this, we need to predict the staticness of program pieces in p *independently* of static values (binding-time improvements do not make much sense unless they improve the result of specialization for a large class of static values).

2.2.3. Offline specialization pipelines. Our approach to multi-level specialization is *purely offline*. A program generator mcogen, which we call multi-level compiler generator, or *multi-level generator* for short, is a program that takes an n-input program p and a binding-time classification $t_0 \cdots t_{n-1}$ of p's input and produces a *multi-level generating extension* $\texttt{p-mgen}_0$ as output. The order in which the input is supplied is specified by the binding-time classification. The smaller the binding-time value t_i, the earlier the input becomes available.

Without loss of generality we assume that p's input is supplied in the order $\texttt{in}_0 \cdots \texttt{in}_{n-1}$, and thus its binding-time classification is $0 \cdots n-1$. Multi-level specialization using multi-level generating extensions is described by

$$\left.\begin{aligned}
\texttt{p-mgen}_0 &= [\![\texttt{mcogen}]\!]_{\mathrm{L}}\ \texttt{p}\ \text{`}0\cdots n{-}1\text{'}\\
\texttt{p-mgen}_1 &= [\![\texttt{p-mgen}_0]\!]_{\mathrm{L}}\ \texttt{in}_0\\
&\vdots\\
\texttt{p-mgen}_{n-2} &= [\![\texttt{p-mgen}_{n-3}]\!]_{\mathrm{L}}\ \texttt{in}_{n-3}\\
\texttt{p-res}'_{n-1} &= [\![\texttt{p-mgen}_{n-2}]\!]_{\mathrm{L}}\ \texttt{in}_{n-2}\\
\texttt{out} &= [\![\texttt{p-res}'_{n-1}]\!]_{\mathrm{L}}\ \texttt{in}_{n-1}
\end{aligned}\right\}\ n \textit{ stages} \tag{9}$$

Combining these and (6) we obtain an equational definition, the correctness criterion of mcogen:

$$[\![\texttt{p}]\!]_{\mathrm{L}}\ \texttt{in}_0 \cdots \texttt{in}_{n-1} = [\![\cdots [\![[\![\texttt{mcogen}]\!]_{\mathrm{L}}\ \texttt{p}\ \text{`}0\cdots n{-}1\text{'}]\!]_{\mathrm{L}}\ \texttt{in}_0 \cdots]\!]_{\mathrm{L}}\ \texttt{in}_{n-1} \tag{10}$$

Given its first input $\texttt{in}_0$, the multi-level generating extension $\texttt{p-mgen}_0$ produces a new specialized multi-level generating extension $\texttt{p-mgen}_1$ and so on, until the final result out is produced given the last input $\texttt{in}_{n-1}$. This is also illustrated in figure 1 where a box

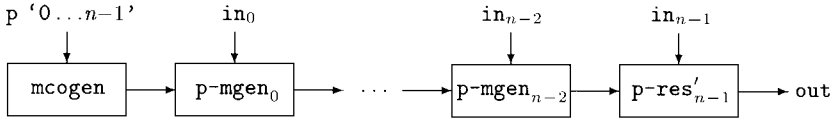

Figure 1. Program specialization with multi-level generating extensions.

denotes a program run with the input supplied from the top and the output shown to the right.

We call this sequence of specializations an *offline specialization pipeline* because once the binding-time classification is consumed by `mcogen` no classification is required later. Note that program $\texttt{p-mgen}_i$ returned by stage i does not become the input of a specializer at the next stage, but can be executed directly. Running a multi-level generating extension $\texttt{p-mgen}_i$ will be faster than using a general specializer `spec`. Multi-level specialization of `p` using multi-level generating extensions is clearly advantageous if `p` is specialized with respect to different input $\texttt{in}_0 \cdots \texttt{in}_{n-1}$.

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. Program `p-gen` in (4) and program $\texttt{p-mgen}_{n-2}$ in (9) are examples of two-level generating extensions.

Similarly, the standard (two-level) `cogen` is a special case of the multi-level `mcogen` and one may define `cogen` in the following way:

$$[\![\texttt{cogen}]\!]_{\mathrm{L}}\ \texttt{p 'SD'} \stackrel{\text{def}}{=} [\![\texttt{mcogen}]\!]_{\mathrm{L}}\ \texttt{p '01'} \tag{11}$$

In other words, `mcogen` can be used to set up any (combination of) online and offline specialization pipelines, while `cogen` can be used only for online specialization pipelines (or for incremental generation of $\texttt{p-mgen}_0$ as explained below). This is a fundamental limitation of the traditional "cogen approach".

2.3. How to generate multi-level generating extensions

Assume that we have only the standard specialization tools: a specializer `spec` and a compiler generator `cogen`. Then we know of two methods [23] that can, in principle, be used to generate multi-level generating extensions: *incremental generation* and *multiple self-application*.

As example consider a three-input 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 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

$$[\![\texttt{mint}]\!]_{\mathrm{L}}\ \texttt{def pgm dat} = [\![\texttt{def}]\!]_{\mathrm{D}}\ \texttt{pgm dat} = [\![\texttt{pgm}]\!]_{\mathrm{P}}\ \texttt{dat} = \texttt{out} \tag{12}$$

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

$$\left.\begin{array}{l} \texttt{comp} = [\![\texttt{mint-cogen}]\!]_{\mathrm{L}}\ \texttt{def} \\ \texttt{tar} = [\![\texttt{comp}]\!]_{\mathrm{L}}\ \texttt{pgm} \\ \texttt{out} = [\![\texttt{tar}]\!]_{\mathrm{L}}\ \texttt{dat} \end{array}\right\} \textit{three stages} \tag{13}$$

The program `tar` is an L-program that returns the same result when applied to `dat` as the P-program `pgm` when applied to the same input. Thus, program `tar` is a target program. The two-level generating extension `comp` is a program which, when given a P-program `pgm`, returns an L-program `tar` and is thus a P-to-L-compiler. The three-level generating extension `mint-cogen` is a program which, when applied to `def`, yields `comp` and is thus a compiler generator.

2.3.1. Incremental generation. A three-level generating extension can be constructed in two steps using the compiler generator `cogen` (or the Futamura projections [21] together with a self-applicable specializer `spec`): first the meta-interpreter `mint` is converted into an 'auxiliary' generating extension `gen-aux` which takes `def` and `pgm` as input, then `gen-aux` is converted into a generating extension `mint-cogen`$'$ which takes only `def` as input.

$$\left.\begin{array}{rl} \texttt{gen-aux} &= [\![\texttt{cogen}]\!]_{\mathrm{L}}\ \texttt{mint 'SSD'} \\ \texttt{mint-cogen}' &= [\![\texttt{cogen}]\!]_{\mathrm{L}}\ \texttt{gen-aux 'SD'} \end{array}\right\} \textit{incremental generation} \tag{14}$$

It is straightforward to verify that `mint-cogen`$'$ is a three-level generating extension using (14) and the equational definition of `cogen`.

$$\begin{aligned} [\![\texttt{mint}]\!]_{\mathrm{L}}\ \texttt{def pgm dat} &= [\![[\![\texttt{gen-aux}]\!]_{\mathrm{L}}\ \texttt{def pgm}]\!]_{\mathrm{L}}\ \texttt{dat} \\ &= [\![[\![[\![\texttt{mint-cogen}']\!]_{\mathrm{L}}\ \texttt{def}]\!]_{\mathrm{L}}\ \texttt{pgm}]\!]_{\mathrm{L}}\ \texttt{dat} \end{aligned} \tag{15}$$

Incremental generation has two main problems:

- *Step-wise generation problem.* In general, it takes $n-1$ generation steps to convert a program into an n-level generating extension using standard specialization tools. It is not possible to obtain the generating extension directly. This has its price: generation time.
- *Respecialization problem.* In practice, the main difficulty is the need to generate and transform several intermediate generating extensions. This is more difficult than the respecialization of residual programs because `cogen` is not just applied to an 'ordinary' program but to a program generator. Most compiler generators are not geared towards the transformation of the generating extensions they produce. Binding-time improvements, should they be necessary, have to be performed on a machine-produced program generator (cf. Section 2.2.2).

2.3.2. Multiple self-application. Given a self-applicable specializer `spec` there is another way to obtain a multi-generating extension: by multiple self-application. This requires up to three self-applications in our example (in contrast to the Futamura projections which

never require more than two self-applications). Note that multiple self-application requires a self-applicable specializer; a compiler generator cannot be used.

We now give four equations which specify the generation of a target program `tar″`, a compiler `comp″`, a compiler generator `mint-cogen″` and a compiler-generator generator `cogengen″`. For readability we distinguish between the specializers with indices and assume that each specializer has the appropriate arity (otherwise they are identical).

$$\begin{aligned}
\texttt{tar}'' &= [\![\texttt{spec}_1]\!]_{\mathrm{L}}\ \texttt{mint 'SSD' def pgm}\\
\texttt{comp}'' &= [\![\texttt{spec}_2]\!]_{\mathrm{L}}\ \texttt{spec}_1\ \texttt{'SSSD' mint 'SSD' def}\\
\texttt{mint-cogen}'' &= [\![\texttt{spec}_3]\!]_{\mathrm{L}}\ \texttt{spec}_2\ \texttt{'SSSSD' spec}_1\ \texttt{'SSSD' mint 'SSD'}\\
\texttt{cogengen}'' &= [\![\texttt{spec}_4]\!]_{\mathrm{L}}\ \texttt{spec}_3\ \texttt{'SSSSDD' spec}_2\ \texttt{SSSSD' spec}_1\ \texttt{'SSSD'}
\end{aligned} \tag{16}$$

Using the equations above and the equational definition of the corresponding specializer $\texttt{spec}_i$ we have

$$\begin{aligned}
\texttt{out} &= [\![\texttt{tar}'']\!]_{\mathrm{L}}\ \texttt{dat}\\
\texttt{tar}'' &= [\![\texttt{comp}'']\!]_{\mathrm{L}}\ \texttt{pgm}\\
\texttt{comp}'' &= [\![\texttt{mint-cogen}'']\!]_{\mathrm{L}}\ \texttt{def}\\
\texttt{mint-cogen}'' &= [\![\texttt{cogengen}'']\!]_{\mathrm{L}}\ \texttt{mint 'SSD'}
\end{aligned} \tag{17}$$

It can easily be verified that `mint-cogen″` is the three-level generating extension we are looking for:

$$\begin{aligned}
[\![\texttt{mint}]\!]_{\mathrm{L}}\ \texttt{def pgm dat} &= [\![\texttt{tar}'']\!]_{\mathrm{L}}\ \texttt{dat}\\
&= [\![[\![\texttt{comp}'']\!]_{\mathrm{L}}\ \texttt{pgm}]\!]_{\mathrm{L}}\ \texttt{dat}\\
&= [\![[\![[\![\texttt{mint-cogen}'']\!]_{\mathrm{L}}\ \texttt{def}]\!]_{\mathrm{L}}\ \texttt{pgm}]\!]_{\mathrm{L}}\ \texttt{dat}
\end{aligned} \tag{18}$$

We can obtain `mint-cogen″` either from the third equation in (16) or the fourth equation in (17). Note that the compiler-generator generator `cogengen″` accepts programs written in L, while the compiler generator `mint-cogen″` accepts definitions written in D, the definition language of the meta-interpreter `mint` (in other words, compiler generators with arbitrary definition languages can be implemented given an appropriate meta-interpreter).

Multiple self-application has two fundamental problems [23]:

- *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 a variant of the notorious encoding problem in self-application (types in programs need to be mapped into a universal type in a partial evaluator): it may lead to an exponential growth of the size of the produced generating extensions (in the number of self-applications).

- *Overgeneralization problem.* In multiple self-application (16), $\texttt{spec}_1$'s environment containing the static valuer (`def , pgm`) is classified as a partially static structure when $\texttt{spec}_1$ is specialized by $\texttt{spec}_2$. In practice, this may require either strong enough specialization methods in $\texttt{spec}_2$ or binding-time improvements of $\texttt{spec}_1$.

2.4. Our goal: A multi-level program generator

To conclude with our example, a multi-level generator `mcogen` solves the meta-interpreter problem elegantly in one step and, as we shall see, very efficiently:

$$\texttt{mint-cogen}''' = [\![\texttt{mcogen}]\!]_{\mathrm{L}}\ \texttt{mint '012'} \tag{19}$$

Program `mint-cogen'''` is, by definition of `mcogen` (10), the required three-level generating extension:

$$[\![\texttt{mint}]\!]_{\mathrm{L}}\ \texttt{def pgm dat} = [\![[\![[\![\texttt{mint-cogen}''']\!]_{\mathrm{L}}\ \texttt{def}]\!]_{\mathrm{L}}\ \texttt{pgm}]\!]_{\mathrm{L}}\ \texttt{dat} \tag{20}$$

To avoid the theoretical limitations and the practical problems of standard (two-level) specialization tools we are going to develop a non-standard, multi-level program generator `mcogen` in the remainder of this paper.

3. A case study: Approaches to a solution

We now turn to the basic methods for constructing a multi-level program generator. 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 present the principles behind our design in a case study. After introducing the example program, we develop our approach starting from two basic observations, and show how the example program can be converted into a two- and three-level generating extension. We will use Scheme [33], an untyped, strict functional programming language, as our presentation language. The same principles apply other programming languages.

3.1. Source and residual program

Suppose the source program is a three-input program for computing the inner product `v`·`w` of two vectors `v` and `w` of dimension `n`. Program `iprod` is shown in figure 3 where `(ref i v)` is used to access the `i`th element of a vector `v` while leaving the internal representation of the vector unspecified (e.g., it may be a list).

Figure 2 shows how the computation of the inner product can be performed in one, two, and three stages (the number below each box refers to the figure in which the respective program is shown). For example, given values for `n` and `v`, the task of `iprod`'s two-level generating extension `iprod2` is to produce a residual program `iprod-nv`.

Figure 4 shows a residual program `iprod-nv` where `n` = 3 and `v` = `[7 8 9]`. The recursion has been unfolded and the vector elements have been in-lined, eliminating the need for bound checks and data references. Running the residual program `iprod-nv` on the remaining input `w` will be faster than running the source program `iprod` on the input `n`, `v`, and `w`.

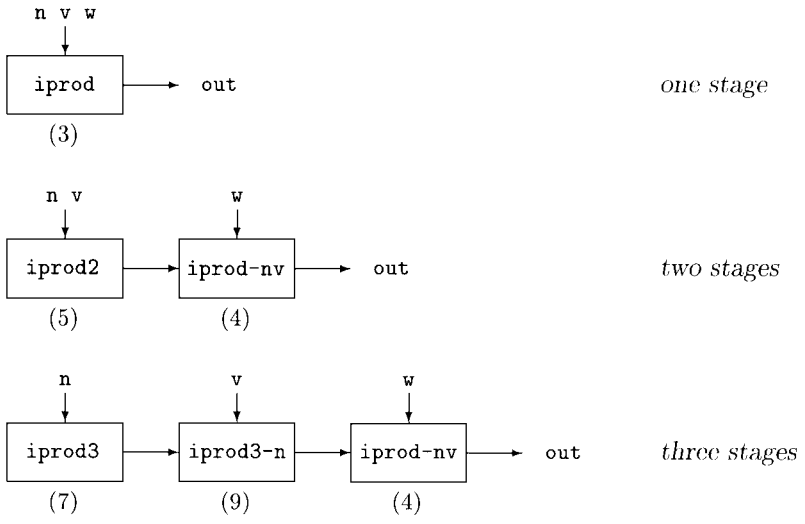

Figure 2. Case study overview: Performing a computation in one, two, and three stages.

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

Figure 3. Source program.

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

Figure 4. Residual program (n = 3, v = [7 8 9]).

But how can one obtain a two-level generating extensions `iprod2` or even a three-level generating extension `iprod3` from `iprod`?

3.2. Constructing a two-level generating extension

In *offline partial evaluation* the transformation process is guided by a *binding-time analysis* performed prior to the specialization phase [35]. The result of the binding-time analysis is a program in which all operations are classified (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* [42], e.g., by marking

```
(define (iprod2 n v w)
  (if (> n 0)
      (_+ (_* (lift (ref n v))
              (_ref (lift n) w))
          (iprod2 (- n 1) v w))
      (lift 0)))
```

Figure 5. A two-level program.

```
(define (_+ x y)   `(+ ,x ,y))
(define (_* x y)   `(* ,x ,y))
(define (_ref x y) `(REF ,x ,y))
(define (lift x)   `(QUOTE ,x))
```

Figure 6. Two-level code generation.

every dynamic operation with an underscore (`_op`), while leaving every static operation unchanged (`op`).

Observation 1 (Annotated programs are generating extensions). An annotated program p_{ann} can be viewed as a generating extension of program p [31]. It is only a question of the language in which one considers p_{ann} as a program: a static operation `op` can be executed, while a dynamic operation `_op` generates program code. In other words, a binding-time analysis, together with a phase annotating the source program, can be viewed as a generator of generating extensions.

Let us continue with our example.

1. *Two-level program.* Assume that the first two arguments of `iprod` are static and that the last argument is dynamic, i.e., `n:static`, `v:static`, and `w:dynamic`. Figure 5 shows the result of annotating `iprod` with the corresponding binding-times. The operation `lift` that has been added converts static values into corresponding pieces of program code. *Lifting* is necessary when static values appear in a dynamic context [41, 44]. For example, the result of the static expressions `(ref n v)` and `n` has to be lifted.
2. *Two-level code generation.* Figure 6 shows how dynamic operations, e.g., `_ref`, can be defined. Their definition is the same for all two-level programs. The backquote notation of Scheme is convenient for building list structures. For example, `(list 'REF 'x 'y)` can be written as `` `(REF ,x ,y) ``. Recall that the expression `'datum` is just an abbreviation for `(QUOTE datum)`. Thus, expression `` `(QUOTE ,x) `` generates a literal constant. For clarity we use upper case letters when we generate code.
3. *Running a two-level program.* The body of the residual program in figure 4 can be obtained by evaluating the annotated program in figure 5 together with the definitions of the dynamic operators in figure 6, where `n` = 3, `v` = `[7 8 9]`, and the dynamic variable `w` is bound to the symbol `w`. Numerical constants need not be quoted and we omitted the quotes generated by `lift`.

Remark. 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. This method will be explained in Section 5.2.

3.3. Constructing a multi-level generating extension

We have seen that a two-level program can be viewed as a two-level generating extension. Consequently, we can view a multi-level program as a multi-level generating extension (and use a non-standard, multi-level binding-time analysis to obtain the multi-level program annotations).

Observation 2 (Multi-level annotation). A static/dynamic annotation of a program is a special case of a multi-level annotation where binding-time values `0`, `1`, ..., `n` indicate at what stage an operation can be reduced.

A multi-level generating extension, in our case a three-level generating extension, can be obtained as follows.

1. *Multi-level program.* Figure 7 shows a three-level version of the inner product assuming that the arguments of `iprod` have the following binding-times: `n:0`, `v:1`, and `w:2`. The program is annotated using a *multi-level syntax* where all dynamic operations have a binding-time value as additional argument. The general format is `(_ 'op t` $e_1 \cdots e_n$`)` where t is the binding-time value and e_i are annotated argument expressions (note that `_` is a legal identifier in Scheme).
2. *Multi-level code generation.* Figure 8 shows the functions for multi-level code generation. They are the same for all multi-level programs.

   Generic code generation. Function `_` has three arguments: an operator `op`, a binding-time value `t`, and the arguments `es` of the operator `op` (code fragments). Code generation

```
(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)))
```

Figure 7. 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))))
```

Figure 8. Multi-level code generation.

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

Figure 9. A generated generating extension (n = 3).

works as follows: 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. Running the function several times corresponds to a 'count-down' until `t` reaches `1` which means that `op` expects its arguments in the next stage.

Multi-level lifting. 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 `_` (which is necessary as long as the value of *e* is not available).

3. *Running a multi-level program.* The body of the two-level generating extension in figure 9 can be obtained by evaluating the three-level generating extension in figure 7 together with the definitions for multi-level code generation in figure 8 where `n = 3`. The result is a specialized version of the two-level generating extension in figure 5 where bound checks are eliminated (and a multi-level annotation is used instead). Note that the binding time arguments have been decremented, e.g., `(_ 'ref 1 ... w)`, while other operations have become executable, e.g., `(ref 3 v)`. Evaluating `iprod-n` with `v = [7 8 9]` returns the same program as shown in figure 4.

3.4. Construction principles

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

- A non-standard, multi-level binding-time analysis together with a phase converting annotations into executable multi-level generating extensions form the core of a multi-level generator.
- Multi-level generating extensions consist of two parts: a multi-level program in executable form and a library providing the appropriate interpretation of annotated operations (code generation, etc.)

The case study also illustrates the main advantages of our 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. Static operations can be executed by the underlying implementation. In principle, one could provide an interpreter for multi-level programs, but this would be less efficient. (Alternatively, one could stage such an interpreter, a way to proof the correctness of the library operations [49]).

The library adds only a constant size of code to a multi-level program (in contrast to Holst [31] who used macro-expansion). The size of a multi-level generating extension is independent of binding-time values (if we ignore the logarithmic costs of representing numbers). Thus, the size of a multi-level generating extensions depends only on the size of the multi-level program (and the static data consumed in earlier stages). Indeed, multi-level generating extensions are very compact. Take the multi-level generating extension `iprod3` (figure 7) as example: it is not dramatically larger than the two-level generating extension `iprod2` (figure 5), although `iprod3` produces a generating extension as output (not just an 'ordinary' residual program as `iprod2`).

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]. However, this does not change the principles underlying our approach.

4. Multi-level binding-time analysis

We are going to develop a *multi-level binding-time analysis* (MBTA) for the multi-level program generator in this section. The task of the MBTA is briefly stated: given a source program p and the binding-time values (bt-values) of its input together with a maximal bt-value, 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 MBTA developed here is monovariant (all calls to the same function have the same bt-pattern). The main motivation for this is efficiency: type-inference based, monovariant binding-time analysis can be done by constraint normalization in almost-linear time in the size of the analyzed programs [30] (in contrast to abstract interpretation-based methods).

After defining the multi-level language (Section 4.1), we introduce binding-time types, give typing rules for well-annotated multi-level programs, and specify minimal (best) completions (Section 4.2). We conclude with a strategy for multi-level binding-time improvements (Section 4.3).

4.1. Multi-level language

The multi-level language is an annotated, higher-order subset of Scheme [33] where every construct has a bt-value $t \geq 0$ as additional argument (figure 10). In addition the multi-level language has a lift operator $\underline{\texttt{lift}}_t^s$. The underlining of an operator, e.g., $\underline{\texttt{if}}_t$, together with

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

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

Figure 10. Abstract syntax of multi-level higher-order programs ($0 \leq n, 0 < m$).

the bt-value t attached to it, is its *annotation*. Consistency and minimality of annotations will be defined below. Function calls are always unfolded so no annotation is used (the insertion of specialization points will be explained in Section 5.2).

The source language, the multi-level language without annotations and `lift` operator, is identical to the core language of Similix [10], a state-of-the-art partial evaluator, except that we omitted user-defined n-ary constructors (which are not part of Scheme).

Definition (Underlying program). Erasing all annotations and lift operators from a multi-level program p_{ann} gives the *underlying program* $|p_{\text{ann}}|$ of p_{ann}.

4.2. Consistency and minimality of annotations

Not all multi-level programs have a consistent bt-annotation. We give typing rules that formalize the intuition that early values may not depend on late values. They define *well-annotated* multi-level programs.

4.2.1. Binding-time values. For every source program p the bt-values t_i of its input are given, as well as a maximal bt-value ν. Expressions with bt-value $t = 0$ are called *static*, otherwise *dynamic*. The meta-variables s, t range over bt-values.

Definition (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.

4.2.2. Binding-time types. 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 $t = \|\tau\|$ of its bt-type τ. When an expression is well-typed (in 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 (Binding-time type). A type τ is a (well-formed) *binding-time type* with respect to ν, if $\vdash \tau : t$ is derivable from the rules below. If $\vdash \tau : t$ then the type τ has a bt-value t, and we define a mapping $\|\cdot\|$ from types to bt-values: $\|\tau\| = t$ **iff** $\vdash \tau : t$.

$$\{Base\}\ \frac{t \le \nu}{\Delta \vdash B^t : t} \qquad \{Fct\}\ \frac{\Delta \vdash \tau_i : s_i \quad \Delta \vdash \tau : s \quad s_i \ge t \quad s \ge t}{\Delta \vdash \tau_1 \cdots \tau_n \to^t \tau : t}$$

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

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, such as integer, boolean, etc., since we are only interested in the distinction between base values and functions. The rule ensures that for all well-formed bt-types $\|\tau\| \le \nu$.

Rule $\{Fct\}$ shows the bt-type for functions where $\tau_1 \cdots \tau_n \to^t \tau$ contains the bt-value t of the function as well as the bt-type of its arguments $(\tau_1 \cdots \tau_n)$ and result (τ). Since a Scheme function may take multiple arguments, we will often use the shorthand notation $\bar{\tau} \to \tau$ where $\overline{\tau}$ abbreviates $\tau_1 \cdots \tau_n$. Rule $\{Fct\}$ requires that the bt-values of the argument types $\bar{\tau}$ and the result type τ are *not* smaller than the bt-value t of the function itself because neither the actual arguments are available in 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^t 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. This is in accordance with the equality $\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$ which holds for recursive types. 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.

Henceforth, all binding-time type are assumed to be well-formed.

4.2.3. Equivalence of binding-time 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). We defer such errors to the latest possible binding time ν. An example will be shown in Section 4.2.5 after introducing the typing rules. Alternatively, this could be encoded in the typing rules, but we choose a separate equivalence relation for clarity.

Definition (Equivalence of bt-types). Let ν be a maximal bt-value and let U be the following equation:

$$\overline{B^\nu} \to^\nu B^\nu = B^\nu$$

Given two bt-types τ and τ' well-formed with respect to ν, we say that τ and τ' are equivalent, written $\vdash \tau \doteq \tau$', if $\tau = \tau'$ is derivable from equation U, the equivalence of recursive bt-types (based on types having the same regular type [15]), and the usual relation for equality (symmetry, reflexivity, transitivity, and compatibility with arbitrary contexts).

4.2.4. Typing rules. The typing rules for well-annotated multi-level programs are defined in figure 11. Most of the typing rules are generalizations of the corresponding rules used

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

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

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

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

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

$$\{Def\}\ \frac{\Gamma\{x_1:\tau_1, \ldots, x_n:\tau_n\} \vdash e : \tau}{\Gamma \vdash (\texttt{define}\ (f\ x_1 \ldots x_n)\ e) : \tau_1 \ldots \tau_n \rightarrow^t \tau}$$

$$\{Prog\}\ \frac{f_i : \bar{\tau}_i \rightarrow^{t_i} \tau_i \text{ in } \Gamma \vdash (\texttt{define}\ (f_i\ \bar{x}_i)\ e_i) : \bar{\tau}_i \rightarrow^{t_i} \tau_i}{\vdash (\texttt{define}\ (f_0\ \bar{x}_0)\ e_0) \ldots (\texttt{define}\ (f_n\ \bar{x}_n)\ e_n) : \Gamma}$$

Figure 11. Typing rules for well-annotated multi-level programs (i ranges over $0 \leq i \leq n$).

for two-level programs in partial evaluation [35]. For instance, rule $\{If\}$ for `if`-expressions annotates the construct `if` with the bt-value t of the test-expression e_1 because 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 $\underline{\texttt{lift}}^s_t$: the value of its argument e has bt-value t, but its result is not available until $t + s$ (where $s > 0$). The bt-value of an expression $(\underline{\texttt{lift}}^s_t\ e)$ is the sum of the bt-values t and s. In other words, the multi-level lift operator delays a value a value to a later binding time. As is customary in partial evaluation (see [35] for a discussion of this point), the rule allows lifting of base values only [42]. The lift operator known from standard partial evaluation (Section 3.2) is a special case of multi-level lifting, where t and s are constants: $\underline{\texttt{lift}}^1_0$.

For practical reasons, and for unique canonical completions, we require that all $\underline{\texttt{lift}}^s_t$ operators in a multi-level program have maximal s. In other words, we merge all nested lift operators into a single operator using the following equivalence: $(\underline{\texttt{lift}}^{r+s}_t e) \equiv (\underline{\texttt{lift}}^r_{t+s}\ (\underline{\texttt{lift}}^s_t\ e))$.

Rule $\{Op\}$ requires that all function-type arguments of primitive operators have bt-value ν. 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 Similix [10]; it can be avoided [50]).

Rules $\{Def\}$ and $\{Prog\}$ type definitions and programs, respectively.

4.2.5. Minimal completions

Definition (Well-annotated completion, minimal completion). Given a program p, a maximal bt-value ν, and a bt-pattern $t_1 \cdots t_n$ of p's goal function f_0, a *well-annotated completion*

of p is a multi-level program p_{ann} with $|p_{\text{ann}}| = p$ iff the following judgment can be derived:

$$\vdash p_{\text{ann}} : \{ f_0 : B^{t_1} \cdots B^{t_n} \to B^{\nu},\ f_1 : \bar{\tau}_1 \to^{t_1} \tau_1, \ldots,\ f_m : \bar{\tau}_m \to^{t_m} \tau_m \}$$

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_{ann} 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 more than one well-annotated completion. The goal of the MBTA is to determine a well-annotated completion p_{ann} which is, preferably, 'minimal', i.e., all operations in a program shall be performed as early as possible. It can be shown, as for the two-level case, that *minimal completions* always exists [27]. The correctness of typing schemes for two-level BTA has been shown by several authors, e.g., [29, 52]. Similar proof methods can be used for the multi-level case (not shown).

Example (bt-type equivalence in a derivation). Consider the expression

```
(if #t 5 (lambda (x) x))
```

where the standard type B of base value `5` and the standard type $\tau \to \tau$ of abstraction `(lambda (x) x)` cannot be unified (which is necessary to type the conditional expression). Using the equivalence relation we can assign both expressions the type B^{ν} and still derive a type for `(lambda (x) x)`. The following is a derivation for the completion `(`$\underline{\texttt{if}}_0$ `#t (`$\underline{\texttt{lift}}_0^{\nu}$ `5) (`$\underline{\texttt{lambda}}_{\nu}$`(x) x))`:

$$\dfrac{\Gamma \vdash \texttt{\#t} : B^0 \quad \dfrac{\Gamma \vdash \texttt{5} : B^0}{\Gamma \vdash (\underline{\texttt{lift}}_0^{\nu}\ \texttt{5}) : B^{\nu}} \quad \dfrac{\dfrac{\Gamma\{x\ :\ B^{\nu}\} \vdash x : B^{\nu}}{\Gamma \vdash (\underline{\texttt{lambda}}_{\nu}\ \texttt{(x)}\ \texttt{x}) : B^{\nu} \to^{\nu} B^{\nu}} \quad \vdash B^{\nu} \doteq B^{\nu} \to^{\nu} B^{\nu}}{\Gamma \vdash (\underline{\texttt{lambda}}_{\nu}\ \texttt{(x)}\ \texttt{x}) : B^{\nu}}}{\Gamma \vdash (\underline{\texttt{if}}_0\ \texttt{\#t}\ (\underline{\texttt{lift}}_0^{\nu}\ \texttt{5})\ (\underline{\texttt{lambda}}_{\nu}\ \texttt{(x)}\ \texttt{x})) : B^{\nu}}$$

4.2.6. Implementation. Based on the normalization algorithms for constraint systems described in [13] we developed an efficient MBTA [27] that runs in almost-linear time in the size of the analyzed programs. Experiments show that it runs (on selected test programs) slightly faster than the corresponding two-level analysis in Similix, which is notable because we did not optimize our implementation for speed [27] (our MBTA has the same accuracy as the BTA in Similix). The results are also significant because they clearly demonstrate that multi-level specialization scales up to advanced languages without performance penalties. This implementation of the MBTA has been used in the multi-level generator and the experiments in Section 6.

4.3. Improving programs with multiple binding times

Certain programming styles can complicate the MBTA, while others make it easier to analyze programs. The need for changing the programming style in certain situations is well-known from standard partial evaluation. *Binding-time improvements* (BTI) are semantics-preserving transformations that make it easier for the analysis to make more operations static (a collection of BTI can be found in [35], Chapter 12).

The question has been raised whether multiple binding times require new methods for BTI. Fortunately, the problem can be reduced to the two-level case where all known BTI techniques apply. A straightforward strategy for multi-level specialization is to make $\nu - 1$ two-level BTI passes over a program to improve the separation between all bt-levels. Start by improving the binding-time separation between bt-levels $\{0, \ldots, \nu - 1\}$ and $\{\nu\}$, then between bt-levels $\{0, \ldots, \nu - 2\}$ and $\{\nu - 1, \nu\}$, until the final separation between bt-levels $\{0\}$ and $\{1, \ldots, \nu\}$. Starting the separation 'at the end', immediately identifies those operations in an expression that have to be delayed longest and appropriate modifications can be made (e.g., reorder associative operations).

In conclusion, multiple binding times do not cause new complications for binding-time improvements, except that they require $\nu - 1$ two-level BTI passes.

5. Multi-level generating extensions

Programs annotated with multiple binding-times need to be converted into executable programs and supplied with an appropriate interpretation for their annotated operations. This section extends the basic methods developed in Section 3 into a full implementation of multi-level generating extensions with higher-order, polyvariant specialization.

We give the structure of generating extensions (Section 5.1), define the conversion from annotated programs into a concrete representation (Section 5.2), and explain the mechanisms for code generation (Section 5.3) and multi-level specialization (Section 5.4). We conclude with a small, but complete example (Section 5.5).

5.1. Representation of multi-level programs

A multi-level generating extension consists of two parts (cf. Section 3.4):

1. *Multi-level program.* The result of converting an annotated program into an executable program.
2. *Library.* Functions for code generation and specialization.

The representation of a multi-level program has to accommodate bt-values, specialization points, and auxiliary functions for code generation. We follow the approach suggested by Observation 1 (Section 3.2): static expressions ($t = 0$) are evaluated by the underlying implementation, while dynamic expressions ($t > 0$) are calls to code generating functions. To avoid infinite unfolding at specialization time, we need to insert specialization points. The concrete syntax of multi-level programs is shown in figure 12 (the operators will be explained below).

$p \in \mathit{Program}$; $d \in \mathit{Definition}$; $e \in \mathit{Expression}$; $c \in \mathit{Constant}$; $x \in \mathit{Variable}$;
$f \in \mathit{Fctname}$; $op \in \mathit{Operator}$; $\ell \in \mathit{Label}$; $sp \in \mathit{SpecPoint}$; $g \in \mathit{Group}$;
$s, t \in \mathit{BindingTimeValue}$

$$
\begin{array}{lll}
p & ::= & d_1 \ldots d_m \\
d & ::= & \texttt{(define (}f\ x_1 \ldots x_n\texttt{)}\ e\texttt{)} \\
e & ::= & c \qquad\qquad\qquad\qquad\qquad\quad \mid\ x \\
 & \mid & \texttt{(}op\ e_1 \ldots e_n\texttt{)} \qquad\qquad\quad \mid\ \texttt{(\_ '}op\ t\ e_1 \ldots e_n\texttt{)} \\
 & \mid & \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} \qquad\qquad\quad \mid\ \texttt{(\_ 'if}\ t\ e_1\ e_2\ e_3\texttt{)} \\
 & \mid & \texttt{(lift}\ s\ e\texttt{)} \qquad\qquad\qquad\quad \mid\ \texttt{(\_ 'lift}\ t\ s\ e\texttt{)} \\
 & \mid & \texttt{(let ((}x\ e_1\texttt{))}\ e_2\texttt{)} \qquad\quad \mid\ \texttt{(\_let}\ t\ e_1\ \texttt{(lambda (}x\texttt{)}\ e_2\texttt{))} \\
 & \mid & \texttt{(}f\ e_1 \ldots e_n\texttt{)} \qquad\qquad\quad \mid\ \texttt{(\_app}\ t\ e_0 \ldots e_n\texttt{)} \\
 & \mid & \texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e\texttt{)} \qquad \mid\ \texttt{(\_lambda}\ t\ n\ \texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e\texttt{))} \\
 & \mid & \texttt{(list 'closure}\ f\ \ell\ g_0 \ldots g_n\texttt{)} \mid\ \texttt{(\_closure}\ t\ sp\texttt{)} \\
 & \mid & \texttt{(\_call}\ t\ sp\texttt{)} \\
sp & ::= & \texttt{(memo}\ t\ \ell\ g_0 \ldots g_n\texttt{)} \\
g & ::= & \texttt{(}t\ \texttt{(}e_1 \ldots e_n\texttt{)}\ \texttt{(}e_1 \ldots e_{n'}\texttt{))}
\end{array}
$$

Figure 12. Concrete syntax of multi-level programs ($0 \leq n$, $0 < m$).

5.2. Converting annotated programs

The conversion of annotated programs into multi-level generating extensions is straightforward, except for two points: *specialization points* need to be inserted and functions created by abstractions need to be represented as *closures*. A summary of the translation is shown in pseudo-code in figure 13. Program code is generated by a recursive decent over the annotated program. The right hand sides show the construction of the code. For simplicity, function definitions are emitted as side-effect when specialization points are inserted.

5.2.1. Choosing specialization points. Compared to standard evaluation, partial evaluation uses a different reduction strategy when processing dynamic conditionals and dynamic abstractions [9, 11].

Standard evaluation evaluates only one of the branches of a conditional, but partial evaluation specializes both. Partial evaluation is therefore strict in both branches of a *dynamic conditional*. A *dynamic abstraction* cannot be beta reduced at specialization time: an abstraction has to be generated where the new body expression is the result of partially evaluating the original body. Hence, when processing dynamic abstractions, partial evaluation is strict in the body of the abstraction.

In both cases, partial evaluation is less terminating: there is a danger of infinite unfolding even if standard evaluation terminates. Hence we must create specialization points for dynamic conditionals and dynamic abstractions: every dynamic conditional/abstraction is replaced by a call to a named function where the actual arguments are the free variables of the expression and the entire conditional/abstraction is the body of the new function. These calls become the specialization points in a generating extensions, while all user-defined function calls will be unfolded at specialization time.

This strategy, introduced in Similix [9, 11] for the two-level case, is straightforward and surprisingly effective in practice, also for multi-level specialization. It guarantees that

Constant, variable:

$$\begin{array}{ll} [\![c]\!] & = \texttt{'}c \\ [\![x]\!] & = x \end{array}$$

Operator:

$$\begin{array}{ll} [\![(\underline{op}_0\ e_1 \cdots e_n)]\!] & = (op\ [\![e_1]\!] \cdots [\![e_n]\!]) \\ [\![(\underline{op}_t\ e_1 \cdots e_n)]\!] & = (\texttt{\_}\ \texttt{'}op\ t\ [\![e_1]\!] \cdots [\![e_n]\!]) \end{array}$$

Lifting:

$$\begin{array}{ll} [\![(\underline{\texttt{lift}}^s_0\ e)]\!] & = (\texttt{LIFT}\ s\ [\![e]\!]) \\ [\![(\underline{\texttt{lift}}^s_t\ e)]\!] & = (\texttt{\_}\ \texttt{'LIFT}\ t\ s\ [\![e]\!]) \end{array}$$

Local binding:

$$\begin{array}{ll} [\![(\underline{\texttt{let}}_0\ ((x\ e_1))\ e_2)]\!] & = (\texttt{LET}\ ((x\ [\![e_1]\!]))\ [\![e_2]\!]) \\ [\![(\underline{\texttt{let}}_t\ ((x\ e_1))\ e_2)]\!] & = (\texttt{\_LET}\ t\ [\![e_1]\!]\ (\texttt{LAMBDA}\ (x)\ [\![e_2]\!])) \end{array}$$

Function call, application:

$$\begin{array}{ll} [\![(f\ e_1 \cdots e_n)]\!] & = (f\ [\![e_1]\!] \cdots [\![e_n]\!]) \\ [\![(e_0\ \underline{@}_t\ e_1 \cdots e_n)]\!] & = (\texttt{\_APP}\ t\ [\![e_0]\!]\ [\![e_1]\!] \cdots [\![e_n]\!]) \end{array}$$

Conditional (insertion of specialization point):

$$\begin{array}{ll} [\![(\underline{\texttt{if}}_0\ e_1\ e_2\ e_3)]\!] & = (\texttt{IF}\ [\![e_1]\!]\ [\![e_2]\!]\ [\![e_3]\!]) \\ [\![(\underline{\texttt{if}}_t\ e_1\ e_2\ e_3)]\!] & = (\texttt{\_CALL}\ t\ (\texttt{MEMO}\ t\ \texttt{'}f_{new}\ gs)) \\ & \quad \textbf{emit}((\texttt{DEFINE}\ (f_{new}\ ys)\ (\texttt{\_}\ \texttt{IF}\ t\ [\![e_1]\!]\ [\![e_2]\!]\ [\![e_3]\!]))) \\ & \quad \textbf{where}\ \langle ys, gs\rangle = FormalsGroups[\![(\underline{\texttt{if}}_t\ e_1\ e_2\ e_3)]\!] \end{array}$$

Abstraction (insertion of specialization point):

$$\begin{array}{ll} [\![(\underline{\texttt{lambda}}_0\ xs\ e)]\!] & = (\texttt{LIST}\ \texttt{'CLOSURE}\ f_{new}\ \texttt{'}f_{new}\ gs) \\ & \quad \textbf{emit}((\texttt{DEFINE}\ (f_{new}\ ys)\ (\texttt{LAMBDA}\ (xs)\ [\![e]\!]))) \\ & \quad \textbf{where}\ \langle ys, gs\rangle = FormalsGroups[\![(\underline{\texttt{lambda}}_0\ xs\ e)]\!] \\ [\![(\underline{\texttt{lambda}}_t\ xs\ e)]\!] & = (\texttt{\_CLOSURE}\ t\ (\texttt{MEMO}\ t\ \texttt{'}f_{new}\ gs)) \\ & \quad \textbf{emit}((\texttt{DEFINE}\ (f_{new}\ ys) \\ & \qquad (\texttt{\_LAMBDA}\ t\ |xs|\ (\texttt{LAMBDA}\ (xs)\ [\![e]\!])))) \\ & \quad \textbf{where}\ \langle ys, gs\rangle = FormalsGroups[\![(\underline{\texttt{lambda}}_t\ xs\ e)]\!] \end{array}$$

Auxiliary function $FormalsGroups[\![e]\!]$ returns a tuple with formal arguments ys and groups gs. The formal arguments ys are the free variables of expression e; the groups contain the free variables of expression e ordered according to their bt-value and type.

Figure 13. Conversion of annotated programs into multi-level generating extensions.

infinite unfolding occurs only if specialization enters a completely statically controlled loop. Standard evaluation would *not* terminate in this case and we accept that the generating extension does not terminate either [35]. Strategies for guaranteeing termination in such situation, but without being overly conservative, are a topic of current research [2, 22].

5.2.2. Representing specialization points. Functions specialized in one stage, may be specialized again in the next stage (as opposed to two-level partial evaluation where a specialized function becomes part of the residual program and is never specialized again).

Hence, multi-level specialization points must have a more general structure than their two-level counterparts.

- In two-level partial evaluation it is sufficient to divide the arguments of a function f into two groups (static, dynamic). In the multi-level case the arguments may have more than two different bt-values: $t \in \{0, \ldots, \nu\}$.
- In two-level partial evaluation, a dynamic conditional/abstraction always has the same bt-value: dynamic. In the multi-level case a dynamic conditional or abstraction may have any dynamic bt-value: $t \in \{1, \ldots, \nu\}$.

To accommodate multi-level specialization in the generating extensions, we choose the format

$$(\texttt{memo}\ t\ f\ g_0 \ldots g_t)$$

for a specialization point `memo` where f is the name of the function to be specialized and t the bt-value of the specialization point. The arguments of the function are ordered into groups $g_0 \cdots g_t$ according to their bt-tags. A group has the form

$$\begin{aligned} g &::= (s\ b\ c) \\ b &::= (e_{\text{base}} \ldots e_{\text{base}}) \\ c &::= (e_{\text{closure}} \ldots e_{\text{closure}}) \end{aligned}$$

where s is the bt-value of the expressions contained in the group, and e_{base} and e_{closure} are expressions of base type and function type, respectively. The separation of expressions according to their type facilitates the comparison of closures during specialization. We assume that there is a group for each bt-value $0 \ldots t$. In the actual implementation empty groups are omitted (hence, the number of groups is limited by the number of arguments and independent of ν).

Auxiliary functions for manipulating groups are shown in pseudo-code in figure 14 and will be used later. Function `decr-groups` decrements the bt-value in a list of groups;

```
(decr-groups (1 b1 c1) ... (t bt ct)) => `((0 ,b1 ,c1) ... (,t-1 ,bt ,ct))

(drop-groups (0 b0 c0) ... (t bt ct)) => (append b0 c0 ... bt ct)

(merge-groups ((1 b1 c1) ... (t bt ct)) ... ((1 b1' c1') ... (t bt' ct'))) =>
   `((1 ,(append b1 ... b1') ,(append c1 ... c1')) ...
     (t ,(append bt ... bt') ,(append ct ... ct')))

(code-groups (0 b0 (k1 ... kn)) (1 b1 c1) ... (t bt ct)) =>
   (list g0 g1 ... gt)
   where
     g0 = `(LIST 0 (LIST . ,b0) (LIST ,k1' ... ,kn'))
     gs = `(LIST s (LIST . ,bs) (LIST . ,cs)) for s = 1, ..., t
     ki' = `(LIST 'CLOSURE ,(->f ki) ',(->l ki)
                 . ,(code-groups (->gs ki))) for i = 1, ..., n
```

Functions to select components of a closure: `->f` *returns function* f, `->l` *returns label* ℓ, *and* `->gs` *returns a list of groups* $(g_0 \cdots g_t)$.

Figure 14. Auxiliary functions for manipulating argument groups.

function `drop-groups` creates a list of all expressions contained in a list of groups; function `merge-groups` joins two or more lists of groups; and function `code-groups` generate an expression that creates a list of groups when evaluated.

Example (Multi-level specialization). Consider the specialization of a function `f` where the argument expressions $e_a \ldots e_e$ are grouped according to their bt-values 0, 2, 3, and 4, respectively. Assume that all arguments are of base type, except e_d which is of function type. Since `f` has no arguments with bt-value 1, the corresponding group has been omitted. Multi-level specialization proceeds by specializing `f` in each stage with respect to the available values until the result of the function can be computed by a direct (static) call to `fabcd`.

```
(memo 4 f (0 (e_a) ()) (2 (e_b e_c) ()) (3 () (e_d)) (4 (e_e) ()))
          (memo 3 fa (1 (e_b e_c) ()) (2 () (e_d)) (3 (e_e) ()))
          (memo 2 fa (0 (e_b e_c) ()) (1 () (e_d)) (2 (e_e) ()))
                     (memo 1 fabc (0 () (e_d)) (1 (e_e) ()))
                                  (fabcd e_e)
```

5.2.3. Representing functions. With polyvariant program point specialization it is necessary to compare the values of static arguments at specialization time. In a higher-order language, a static value may be a function, so we need to compare functions for equality. Since extensional equality of functions is undecidable, we use closure equality as approximation. A closure may, depending on the implementation, be an expression/environment pair containing a function and a list of values for its free variables. To accommodate higher-order specialization in multi-level generating extensions, we choose closures of the form

$$k ::= \texttt{(CLOSURE } f\ \ell\ g_0 \ldots g_t\texttt{)}$$

where f is a closed function, ℓ a label that identifies the abstraction that created the closure, and $g_0 \ldots g_t$ are groups containing arguments for the free variables. The groups constitute the environment component of the closure and are sorted according to their bt-tag. Function f is a closed expression that has not yet been applied to the arguments of the free variables contained in the groups. When applied to these arguments, function f returns a new function that can be applied to the actual arguments of an application (see the description of closure application in Section 5.3.4). Note that it is possible to build closures that contain closures.

Our approach is similar to the one used in the partial evaluators Similix [10] and Schism [16], but generalized to multi-level specialization. A possible optimization is to choose the closure representation only for functions that will become arguments of specialization points [50] (an easy optimization is to use ordinary lambda-expressions for abstractions with $t = v$). Other partial evaluators for higher-order languages, such as Lambdamix [28] and the hand-written compiler generator for SML [12] do not specialize functions with respect to functions and thus need no closure representation.

Remark. Partially static data structures can be represented in a way very similar to closures. The corresponding extension of the MBTA adds no further complications and the

specialization mechanism can be adapted easily (the present implementation of our multi-level generator includes partially static data structures).

5.2.4. Avoiding code duplication. To prevent the duplication of residual expressions when unfolding function calls, we insert let-expressions [11] for all formal arguments in a function definition (including those definitions introduced for dynamic conditionals and dynamic abstractions). Inserted let-expressions are unfolded only when the corresponding variable becomes static.

5.3. Code generation library

The code generating functions required to run multi-level generating extensions are described in this section. The mechanism for multi-level specialization will be explained in Section 5.4. Together they constitute the library added to the programs obtained by the conversion described in the previous section. The code generating functions are listed in figure 15.

5.3.1. Generic code-generation. A generic function, denoted by `_`, can be used for primitive operations, conditionals, and the `lift` operator. Let-expressions, abstractions and

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

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

(define (_let t e f)                     ; local binding
  (let* ((x (new-vname)) (body (f x)))
    (if (= t 1)
        `(LET ((,x ,e)) ,body)
        `(_LET ,(- t 1) ,e (LAMBDA (,x) ,body)))))

(define (_lambda t arity f)              ; abstraction
  (let* ((xs (new-vnames arity)) (body (apply f xs)))
    (if (= t 1)
        `(LAMBDA ,xs ,body)
        `(_LAMBDA ,(- t 1) ,arity (LAMBDA ,xs ,body)))))

(define (_app t e . es)                  ; application
  (if (= t 0)
      (apply (apply (k->f e) (drop-groups (k->gs e))) es)
      `(_APP ,(- t 1) ,e . ,es)))
```

Figure 15. Library for multi-level code-generation.

applications cannot be handled because local bindings and closures require a special treatment. The function takes three arguments: an operator `op`, a bt-value `t`, and the arguments `es` of the operator `op` (code fragments). If the bt-value `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 the bt-value `t` is decreased by 1.

Remark. A possible extension is to perform certain peephole optimizations by inspecting code pieces on the fly. For example, instead of generating an expression `(* 1 x)`, one may simplify it to `x`. However, this can also be accomplished by a post-processing phase or an optimizing compiler.

5.3.2. Multi-level lifting. We use the following representation of the multi-level lift operator:

$$\underline{\texttt{lift}}^{s}_{0}\, e \;\equiv\; (\texttt{LIFT}\ s\ e) \quad \text{and} \quad \underline{\texttt{lift}}^{s}_{t}\, e \;\equiv\; (\texttt{\_'LIFT}\ t\ s\ e)$$

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 bt-value t and `lift` has to be delayed t times before lifting the argument s times. Function `lift` then counts the bt-value s down to 1 before releasing value e as a literal constant. (The function is called `lift` for historical reasons, although the name `_quote` might be more appropriate since what it does can be called multi-level quoting).

5.3.3. Lambda abstraction and local binding. The treatment of dynamic abstractions and let-expressions is straightforward. To avoid variable capture, fresh variables are introduced at each stage. They replace the old variables in the let-expression or the abstraction, respectively. Here substitution is handled by lambda-expressions that are applied to the fresh variables. This correspond to the renaming performed in a standard two-level reducer [35], except that we use lambda-expressions as in [49]. Let-expressions are unfolded only if they are static (normal evaluation). This can be refined by using an occurrence count analysis [11] to determine which dynamic let-expressions can be unfolded safely without changing the termination behavior of the program.

5.3.4. Application. Since functions are represented as closures (Section 5.2.3), function application has to be implemented as *closure application*. Closure application `_app` is done in two steps: first the function contained in the closure is applied to the values of the free variables (after removing the group structure with `drop-groups`), then the resulting function is applied to the actual arguments `es` of the application.

Example (Closure application). Consider the following expression which is the result of converting the annotated expression $((\underline{\texttt{lambda}}_0\texttt{(x)}\ (\underline{+}_0\ \texttt{x y}))\ \ \underline{@}_0\ \texttt{z})$ into executable code. The expression building the closure contains function `f47`, a label `'f47`, and a single group with bt-tag `0`. When the expression is evaluated, the value of the free variable `y` is captured and the closure is created. Closure application `_app` then applies function `f47`

contained in the closure to the value of `y` and the resulting function to the actual argument `z`. The auxiliary function `f47` could also be inlined as lambda abstraction.

```
(_app 0 (list 'CLOSURE f47 'f47 (list 0 (list y) (list))) z)
```

aux. **function**: `(define (f47 y) (lambda (x) (+ x y)))`

5.4. Specialization points

The multi-level specialization mechanism used in a multi-level generating extension is described in this section. The insertion and representation of multi-level specialization points has been discussed in Sections 5.2.1 and 5.2.2.

A *specialization point* is a call to a named function f that is not unfolded at specialization time, but treated in the following way: 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* [14] because the same program point may be specialized with respect to different static arguments.

5.4.1. The multi-level specialization mechanism. Figure 16 defines the function `memo` for polyvariant and depth-first specialization. The function may generate a new function definition as side-effect. The function has two tasks:

1. *Memoization.* The function maintains a memoization table (*memo-list*) of previously encountered program points of the form `(f stat-spine)`, where `f` is a function name and `stat-spine` are the static components ($\overline{arg}$) of the arguments. If the program point `pp` has not been specialized before, the corresponding body is generated by applying `f` to `stat-spine`, and then added to the memo-list (recall that `f` is an annotated function).
2. *Self-generation.* The function decreases the bt-value `t` and produces a call to itself unless `t` becomes `1`. In the latter case only the name of the residual function and the new groups are returned.

```
(define (memo t f . gs)
  (let* ((new-args (decr-groups (get-dyn gs)))
         (stat-spine (cut-dyn gs))
         (pp `(,f ,stat-spine)))
    (if (notSeenB4 pp)
        (begin
         (addToMemoList pp)
         (updateMemoList pp (apply (eval f) (drop-groups stat-spine)))))
    (let ((fnew (getResName pp)))
      (if (= t 1)
          `(,fnew . ,new-args)
          `(MEMO ,t-1 (QUOTE ,fnew) . ,(code-groups new-args))))))
```

Figure 16. The multi-level specialization mechanism.

A few auxiliary functions are used in the definition of function `memo`: function `notSeenB4` checks whether the current program point exists in the memo-list; function `addToMemoList` adds a program point to the memo-list and generates a new residual function name which can be retrieved by function `getResName`; function `updateMemoList` updates the memo-list with the specialized body of `f`; and function `call` applies a function to a list of arguments. Note that `memo` adds a new specialization point to the memo-list *before* specializing `f` because in depth-first specialization one may encounter the same specialization point again while specializing `f`'s body; otherwise specialization may loop.

The functions `get-dyn` and `cut-dyn` are complementary: function `get-dyn` collects all dynamic expressions contained in a list of groups, while function `cut-dyn` replaces all dynamic expressions contained in a list of groups by fresh variables. We explain functions `cut-dyn` and `get-dyn` first for the first-order case and then for the higher-order case (figure 17).

- *First-order polyvariant specialization.* Since functions are never specialized with respect to static functions in the first-order case, the closure component of the static group is always empty. As a result, the definitions of `cut-dyn` and `get-dyn` are straightforward: function `get-dyn` removes the static group and returns only the dynamic groups, and function `cut-dyn` replaces all expressions in the dynamic groups by fresh variables. These definitions are sufficient, if polyvariant specialization is restricted to first-order

First-order specialization:

```
(get-dyn (0 b0 ()) (1 b1 c1) ... (t bt ct)) =>
   `((1 ,b1 ,c1) ... (,t ,bt ,ct))

(cut-dyn (0 b0 ()) (1 b1 c1) ... (t bt ct)) =>
   `((0 ,b0 ()) (1 ,x1 ,y1) ... (,t ,xt ,yt))
   where
     xs = (map newVar bs) for s = 1,...,t
     ys = (map newVar cs) for s = 1,...,t
```

Higher-order specialization:

```
(get-dyn (0 b0 (k1 ... kn)) (1 b1 c1) ... (t bt ct)) =>
   (merge-groups gs1 ... gsn `((1 ,b1 ,c1) ... (,t ,bt ,ct)))
   where
     gsi = (get-dyn (k->gs ki)) for i = 1,...,n

(cut-dyn (0 b0 (k1 ... kn)) (1 b1 c1) ... (t bt ct)) =>
   `((0 ,b0 (,k'1 ... ,k'n)) (1 ,x1 ,y1) ... (,t ,xt ,yt))
   where
     xs = (map newVar bs) for s = 1,...,t
     ys = (map newVar cs) for s = 1,...,t
     k'i = `(CLOSURE ,(k->f ki) ,(k->l ki)
                     . ,(cut-dyn (k->gs ki))) for i = 1,...,n
```

Figure 17. Finding and removing dynamic components.

values, as in [26], SML-Mix [5], and the hand-written cogens [12, 49] or if monovariant specialization is used, as in Lambdamix [28].

- *Higher-order polyvariant specialization.* A closure is a partially static structure which contains static and dynamic components. Two closures are equal when their labels are equal and their static groups are equal (modulo variable renaming in the static closures). The recursive traversal and replacement of dynamic components is done by function `cut-dyn`. The extraction of the dynamic components is done by function `get-dyn`. Since each closure contains a list of groups, the extracted components have to be merged with the already existing groups. In the actual implementation, the traversal of the groups is done only once.

5.4.2. Generating code for function calls and closures. When a specialization point disappears because $t = 1$, an ordinary function call or a closure has to be generated. For this purpose, each specialization point is enclosed in a call to `_CALL` or `_CLOSURE` (figure 13). The functions are defined by

```
(define (_CALL t e)
(if (= t 1)
`(,(m->f e) .  ,(drop-groups (m->gs e)))
`(_CALL ,(- t 1) ,e)))

(define (_CLOSURE t e)
(if (= t 1)
`(LIST 'CLOSURE ,(m->f e) ,(newlabel) . ,(code-groups (m->gs e)))
`(_CLOSURE ,(- t 1) ,e)))
```

where `_CALL` constructs a proper call to function `f` by dropping the group structure around the arguments, and `_CLOSURE` builds an expression which, when evaluated, returns a closure that can be applied with `_app`. One could instead introduce two versions of `memo`, one for dynamic conditionals and one for dynamic abstractions. However, encapsulating the memoization mechanism in a single function has the advantage that both constructs can be treated in a uniform way. The functions `m->f` and `m->gs` extract the function name and the groups from the list returned by `memo`.

5.5. A higher-order multi-level generating extension

Let us illustrate some points of higher-order multi-level generating extensions by a small, but complete example: the insertion of specialization points, the representation of functions, and the handling of local bindings. For readability, we left out let-expressions inserted before the MBTA phase (Section 5.2.4). Otherwise the code presented here is identical to the one generated by the multi-level generator. The source program, the annotated program, and the generated generating extensions are shown in figure 18.

Source program:

```
(define (f x y z)
  (let ((c (lambda (x) (+ x y))))
    (if z (c x) x)))
```

⇓ mcogen: MBTA (figure 11)

Annotated source program:

```
(define (f x y z)
  (let0 ((c (lambda0 (x) (+1 (lift0^1 x) y))))
    (if2 z (lift1^1 (c x)) (lift0^2 x))))
```

⇓ mcogen: conversion (figure 13)

Three-level generating extension:

```
(define (_sim-goal x y z)
  (f x y z))
(define (f-lambda-1 y)
  (lambda (x) (_ '+ 1 (lift 1 x) y)))
(define (f-if-2 x c z)
  (_ 'if 2 z (_ 'lift 1 1 (_app 0 c x)) (lift 2 x)))
(define (f x y z)
  (let ((c (list 'closure f-lambda-1 -1 (list 1 (list y) (list)))))
    (_call 2 (memo 2 'f-if-2 (list 0 (list x) (list c))
                             (list 2 (list z) (list))))))
```

⇓ static input x=11

Two-level generating extension:

```
(define (_sim-goal$2-1 y$1 z$1)
  (_call 1 (memo 1 'f-if-2$2-2 (list 0 (list y$1) (list))
                               (list 1 (list z$1) (list)))))
(define (f-if-2$2-2 y$7 z$8)
   (_ 'if 1 z$8 (lift 1 (+ '11 y$7)) (lift 1 '11)))
```

⇓ static input y=22

Residual program:

```
(define (_sim-goal$1-1 z$2) (f-if-2$1-2 z$2))
(define (f-if-2$1-2 z$4) (if z$4 '33 '11))
```

⇓ static input z=#t

Output: 33

Figure 18. An offline specialization pipeline.

1. *Source and annotated program.* The source programs takes two numbers `x`, `y` and a boolean value `z` as input, and creates a function which is conditionally applied depending on the boolean value. What the program does is not essential for our purpose. Assume the input variables `x`, `y`, and `z` have the bt-values 0, 1, and 2, respectively. The program is annotated by the MBTA according to the typing rules shown in figure 11.
2. *Multi-level generating extension.* The annotated program is converted into an executable three-level generating extension by the conversion rules shown in figure 13. The generating extension consists of four functions.

   The conditional `if` is dynamic (the test has bt-value 2) and therefore a specialization point `memo` has been inserted together with a new function `f-if-2` whose body is the annotated conditional (Section 5.2.1). In the specialization point the free variables of the conditional (`x`, `c`, `z`) and the arguments of the new function `f-if-2` are grouped according to their bt-value and type (base value, function) (Section 5.2.2). There is no argument with bt-value 1 and thus only groups for the bt-values 0 and 2 are needed.

   The abstraction `lambda` is not dynamic, so code has been added that evaluates to a closure (Section 5.2.3). The closure includes the function `f-lambda-2`, the negative number `-1` as label, and a group for the bt-value 1 which captures the value of the free variable `y` when evaluated. Function `f-lambda-2` evaluates to a function when applied to the value of the free variable `y`.
3. *Running the generating extensions.* Running the three-level generating extension produces a two-level generating extension which in turn returns a residual program to compute the final result. Let us run the specialization pipeline by supplying the input `11`, `22`, and `#t`.

   The two-level generating extension `_sim-goal$2-1` is obtained by running the three-level generating extension with `(_sim-goal 11 'y$1 'z$1)`, where `11` is the static input and `'y$1 'z$1` are placeholders for the dynamic input. Function `f-if-2$2-2` in the two-level generating extension is a version of `f-if-2` specialized with respect to the static value `11` and the function `f-lambda-1`. The specialized function has two arguments where `y$1` corresponds to `y` and appears as argument because `y` was a free, but dynamic variable of `f-lambda-1`. The new specialization point `memo` has two groups tagged with bt-values 0 and 1.

   The residual program `_sim-goal$1-1` is obtained by running the two-level generating extension with `(_sim-goal$2-1 22 'z$2)`, where `22` is the static input and `'z$2` is a placeholder for the dynamic input. Applying the residual program `_sim-goal$1-1` to the value `#t` returns the final result `33`.

6. Assessment of the multi-level generator

The multi-level program generator has been used on several problems, all of an experimental nature. We discuss experiments with applications that have been considered theoretically in Section 2. We compare the multi-level generator with *multiple self-application*, and use it for *online* and *offline specialization pipelines*. One of the larger applications has been the generation of a compiler generator capable of a form of supercompilation, a problem that we previously [25] attacked unsuccessfully with a state-of-the-art (two-level) partial evaluator using *incremental generation*.

6.1. Comparison with multiple self-application

The payoff of the multi-cogen approach becomes apparent when compared to specialization by 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 the use of self-applicable specializers up to two self-applications, it becomes critical in applications that beyond the third Futamura projection (Section 2.3.2).

Experiment. The first experiment with multiple self-application was carried out in [23]: staging a program for matrix transposition into 2 to 5 levels. To compare both approaches, we repeated the experiment using the multi-level generator.

Let `transp` be a source program that transposes a matrix where each argument takes one row of the matrix. Assume we want to transpose a $5 \times n$ matrix and let `a`, ..., `e` denote its five rows. Then the application of `transp` to the matrix is described by

```
out = [[transp]] a b c d e
```

where `out` is the transposed matrix (a list of rows). Using a multi-level generator `mcogen`, we converted `transp` into four different multi-level generating extensions `gen2`, ..., `gen5`.

Results. The generation time for the multi-level generating extensions `gen2`,..., `gen5` and their code size are given in Table 1. It also shows how the total generation time t_{total} is composed of t_{bta}, the time for doing the multi-level binding-time analysis, and t_{ann}, the time for converting the annotation into an executable multi-level generating extension (including the insertion of specialization points, etc.).

Table 2 shows the size and run time of each generating extensions `gen5`, $\texttt{gen5}_\texttt{a}$,..., $\texttt{gen5}_\texttt{d}$ when applied to the rows `a`,..., `e` of a 5×3 matrix.

Table 1. Performance of the multi-level generator.

Out Run	t_{bta}/ms	t_{ann}/ms	t_{total}/ms	Size/cells
`gen2 = [[mcogen]] transp '00001'`	5.5	8.0	13.5	180
`gen3 = [[mcogen]] transp '00012'`	5.4	11.0	16.4	236
`gen4 = [[mcogen]] transp '00123'`	5.1	15.1	20.2	295
`gen5 = [[mcogen]] transp '01234'`	5.1	19.5	24.6	357

Table 2. Sizes and run times of the generating extensions.

Out Run	Time/ms	Size/cells
$\texttt{gen5}_\texttt{a}$ = [[gen5]] a	28	357
$\texttt{gen5}_\texttt{b}$ = [[$\texttt{gen5}_\texttt{a}$]] b	19	810
$\texttt{gen5}_\texttt{c}$ = [[$\texttt{gen5}_\texttt{b}$]] c	12	608
$\texttt{gen5}_\texttt{d}$ = [[$\texttt{gen5}_\texttt{c}$]] d	5.6	427
out = [[$\texttt{gen5}_\texttt{d}$]] e	0.1	150

The run times are cpu-seconds obtained with Chez Scheme 3.2 on a Sparc Station 2/Sun OS 4.1 (excluding time for garbage collection, if any). The size of the programs is given as the number of cons cells needed to represent the program.

Discussion. The results show (Table 1) that the run time of the multi-level binding-time analysis does not depend on the number of staging levels. The time required for generating the program grows because more specialization points need to be inserted (their number is bounded by the number of conditionals in a program). The code size of the generating extensions grows for the same reason: extra code is added for specialization points. Otherwise, the size of the generating extensions is independent of the number of the binding-time levels.

The results show an impressive reduction of generation time and code size compared to the result reported for multiple self-application. The absolute size of the generating extensions produced by multiple self-application ranges from 112 to more than 12000 cons cells [23]. In other words, 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 (Table 1).

The absolute run time of multiple self-application to produce the generating extensions ranges from less than 100 ms to more than 15 minutes (obtained on a hardware different from the one used here) [23]. Thus, 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 (Table 1).

6.2. Comparison with incremental generation

A multi-level generating extension can be constructed by repeated application of a compiler generator. However, existing (two-level) compiler generators are not geared towards the transformation of the generating extensions they produce. The main difficulty is the need to retransform several intermediate generating extensions (Section 2.3.1). This can make it difficult, or even impossible in practice, to obtain efficient multi-level generating extensions automatically.

Experiment. The specializer projections [24] tell us how to obtain specializers from interpreters; the Futamura projections [21] tell us how to obtain compiler generators from specializers. It is an intriguing thought to combine both projections in order to generate (more powerful) compiler generators from instrumented interpreters, an application investigated in [25] where an interpreter extended with a driving mechanism was converted into a specializer capable of a form of supercompilation.

Let an interpreter `int2` for two-input programs be defined by

$$[\![\texttt{int2}]\!]_{\mathrm{L}}\ \texttt{pgm}\ \texttt{in}_0\ \texttt{in}_1 = [\![\texttt{pgm}]\!]_{\mathrm{P}}\ \texttt{in}_0\ \texttt{in}_1 \tag{21}$$

Using the specializer projections and the Futamura projections, a compiler generator `cogen2` can be obtained in two steps from the interpreter `int2`: first by converting the interpreter

into a specializer spec2, and then by converting the specializer into a compiler generator cogen2. The incremental generation of cogen2 is described by

$$\left.\begin{aligned} \texttt{spec2} &= [\![\texttt{cogen}]\!]_{\mathrm{L}}\ \texttt{int2 `SSD'} \\ \texttt{cogen2} &= [\![\texttt{cogen}]\!]_{\mathrm{L}}\ \texttt{spec2 `SD'} \end{aligned}\right\} \textit{incremental generation} \quad \begin{aligned} &(22)\\ &(23)\end{aligned}$$

Suppose spec2 is a supercompiler, then cogen2 is a compiler generator that is strong enough to achieve supercompilation effects. For example, one can convert a naive pattern matcher match into a generating extension match-gen that, given a fixed pattern, produces specialized matchers that are equivalent to those generated by the Knuth, Morris, and Pratt algorithm [46]. The program match takes a pattern and a string as input and checks whether the pattern occurs as a substring in the string. Computation of match in two stages is described by

$$\texttt{match-gen} = [\![\texttt{cogen2}]\!]_{\mathrm{L}}\ \texttt{match `SD'} \quad (24)$$

$$\texttt{kmp-match} = [\![\texttt{match-gen}]\!]_{\mathrm{L}}\ \texttt{pattern} \quad (25)$$

$$\texttt{out} = [\![\texttt{kmp-match}]\!]_{\mathrm{L}}\ \texttt{string} \quad (26)$$

The question whether (22), the generation of a specializer from an interpreter, is computationally feasible has been answered positively [25]. But the generation of cogen2 in (23) *failed*. It became impossible to respecialize the generating extension spec2. The necessary (manual) binding-time improvements of the machine-produced program spec2 turned out to be too difficult (program spec2 is written in continuation-passing style where static and dynamic values flow together [25]).

We repeated the experiment with mcogen:

$$\texttt{cogen2}' = [\![\texttt{mcogen}]\!]_{\mathrm{L}}\ \texttt{int2 `012'} \quad (27)$$

Results. Using the multi-level generator mcogen, we converted the driving interpreter int2 defined in [25] into a compiler generator that is capable of supercompilation with α-identical folding (calls are folded if they are identical up to renaming). Indeed, given a naive pattern matcher, the new compiler generator cogen2 produces a generating extension match-gen (24) that generates specialized matchers kmp-match (25) that are equivalent to those generated by the Knuth, Morris, and Pratt algorithm.

The size of the driving interpreter is 1494 cons cells, the total generation time of the generated compiler generator was 1.77 s and the size of the program is 3364 cons cells. The run times are cpu-seconds obtained with Chez Scheme 3.2 on a Sparc Station 2/Sun OS 4.1 (excluding time for garbage collection, if any).

Discussion. The multi-level generator solved the transformation problem elegantly in one step. The need to generated 'intermediate' programs is avoided. Note the speed of the transformation.

The result is also remarkable because the compiler generator obtained from the instrumented interpreter by three-level staging (27) is stronger than the program generator used

for its generation. The self-application scheme used in the generation of the compiler generator is different from the 3rd Futamura projection. This can be seen by writing out (23) using a self-applicable specializer `spec` instead of `cogen` (self-application would require `spec` = `spec2`). Indeed, `spec` is not self-applied, which would 'only' generate a `cogen` of the power of `spec`, but applied to a more powerful `spec2` which results in a more powerful `cogen2`.

$$\texttt{cogen2} = [\![\texttt{spec}]\!]_{\mathrm{L}}\ \texttt{spec 'SSD' spec2 'SD'} \tag{28}$$

6.3. Specialization pipelines for meta-programming

A multi-level generator can divide a computation into two or more stages, with the pleasing consequence that a multi-level generator can be used to set up online and offline specialization pipelines (Section 2.2). This allows applications that would otherwise not have been possible. The purpose of this section is to show, using a concrete example, how a multi-level generator can eliminate the interpretive overhead of meta-programming using online and offline specialization pipelines.

For many well-defined classes of language definitions, one may, in principle, write a meta-interpreter `mint` such that, given a language definition `def` that defines some language P, `mint` can directly execute P-programs (Section 2.3). While this approach has many theoretical advantages, there are substantial efficiency problems in practice: considerable time may be spent interpreting the language definition `def` rather than computing the operations specified by the P-program.

Experiment. Let meta-interpreter `mint` be defined as in (12). To reduce `mint`'s interpretive overhead, we perform its computation in three stages. There are two ways for doing this.

- *Online pipeline.* Generate a two-level generating extension `int-gen`, an *interpreter generator*, with the binding times `'011'` and use it to convert a P-language definition `def` into a P-interpreter `int`. Then convert the interpreter into a P-compiler $\texttt{comp}_1$ and use it to compile program `pgm`. Finally, run the target program `tar`.
- *Offline pipeline.* Alternatively, generate a three-level generating extension, a compiler-generator `mint-cogen`, from `mint` with the binding times `'012'` and run the generating extensions.

Online pipeline		**Offline pipeline**	
(*a*)	$\texttt{int-gen} = [\![\texttt{mcogen}]\!]\ \texttt{mint '011'}$	$\texttt{mint-cogen} = [\![\texttt{mcogen}]\!]\ \texttt{mint '012'}$	(*w*)
(*b*)	$\texttt{int} = [\![\texttt{int-gen}]\!]\ \texttt{def}$	$\texttt{comp}_2 = [\![\texttt{mint-cogen}]\!]\ \texttt{def}$	(*x*)
(*c*)	$\texttt{comp}_1 = [\![\texttt{mcogen}]\!]\ \texttt{int '01'}$	$\texttt{tar}_2 = [\![\texttt{comp}_2]\!]\ \texttt{pgm}$	(*y*)
(*d*)	$\texttt{tar}_1 = [\![\texttt{comp}_1]\!]\ \texttt{pgm}$	$\texttt{out} = [\![\texttt{tar}_2]\!]\ \texttt{dat}$	(*z*)
(*e*)	$\texttt{out} = [\![\texttt{tar}_1]\!]\ \texttt{dat}$		

In our experiments, 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. The size of the meta-interpreter is 533 cells. The structure of the meta-interpreter is straightforward, and no binding-time improvements were necessary when writing it, but then of course it was an "insider job". The reader is referred to Appendix A for more details about the meta-interpreter.

Results. Table 3 shows the generation times, the memory (mem) allocated during the generation and the sizes of the two- and three-level generating extension obtained from `mint` (the output). For each generating extension the total generation time is given (the sum of the run times of the multi-level binding-time analysis and the conversion phase). The size of the programs is given as the number of cons cells needed to represent the program.

Table 4 shows the generation times and sizes of the compilers $comp_1$ and $\texttt{comp}_2$ generated from the definition `def`. Table 5 shows the performance of the compilers $\texttt{comp}_1$ and $\texttt{comp}_2$ given program `pgm`. Table 6 shows the run times of the example program

Table 3. Performance of `mcogen`.

Out	Run	Time/s	Mem/kcells	Size/cells
(*a*)	`int-gen` = ⟦`mcogen`⟧ `mint '011'`	10.9	529	1486
(*w*)	`mint-cogen` = ⟦`mcogen`⟧ `mint '012'`	10.0	529	1525

Table 4. Compiler generation.

Out	Run	Time/s	Mem/kcells	Size/cells
(*b*)	`int` = ⟦`int-gen`⟧ `def`	.66	32	640
(*c*)	$\texttt{comp}_1$ = ⟦`mcogen`⟧ `int '01'`	10.8	483	1262
(*x*)	$\texttt{comp}_2$ = ⟦`mint-cogen`⟧ `def`	.63	34	840

Table 5. Compiler performance.

Out	Run	Time/ms	Mem/kcells	Size/cells
(*d*)	$\texttt{tar}_1$ = ⟦$\texttt{comp}_1$⟧ `pgm`	233	11.5	285
(*y*)	$\texttt{tar}_2$ = ⟦$\texttt{comp}_2$⟧ `pgm`	83	5.18	109

Table 6. Performance of programs.

Out	Run	Speedup	Time/ms	Mem/kcells
(*m*)	`out` = ⟦`mint`⟧ `def pgm dat`	1	630	44.3
(*i*)	`out` = ⟦`int`⟧ `pgm dat`	5.3	120	8.20
(*e*)	`out` = ⟦$\texttt{tar}_1$⟧ `dat`	30	21	2.17
(*z*)	`out` = ⟦$\texttt{tar}_2$⟧ `dat`	72	8.7	1.93
(*n*)	`out` = ⟦`fac`⟧ `dat`	708	.89	.037

using the meta-interpreter, the generated interpreter, and both target programs. 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.

Discussion. Contrary to what one might expect the offline pipeline gives the fastest target program $\texttt{tar}_2$ for the factorial which is 'only' around 10 times slower than the factorial `fac` written directly in Scheme (Table 6). Program $\texttt{tar}_2$ is more than twice as fast as $\texttt{tar}_1$ which in turn is more than five times faster than the interpretation of the factorial `pgm` by `int`. 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 programs $\texttt{tar}_1$ and $\texttt{tar}_2$ are slower than the standard version `fac` is that primitive operations are still interpreted in the target programs, e.g., an expression `(+ 1 2)` gets translated into `(((ext '+) 1) 2)` where `ext` is a function that, given `'+`, returns a curried version of the addition. 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.

The reason why $\texttt{tar}_1$ runs slower than $\texttt{tar}_2$ is that the online approach does not improve the specialization of our meta-interpreter, despite the higher precision it allows, and that our implementation uses only a simple post-processor to 'clean up' residual code (recall that $\texttt{tar}_1$ is the residual program of a residual program!). Given a better (optimizing) post-processor, we expect that $\texttt{tar}_1$ and $\texttt{tar}_2$ come closer in performance.

The compiler $\texttt{comp}_2$ generated by the offline pipeline is almost 3 times faster than the compiler $\texttt{comp}_1$ generated by the online pipeline (Table 5). The generated target program $\texttt{tar}_2$ is three times smaller than $\texttt{tar}_1$. Again this difference is mainly due to the limited post processing.

The generation of the compiler $\texttt{comp}_2$ by the offline method is—not surprisingly—faster than using the online method (Table 4). Compiler $\texttt{comp}_2$ is also smaller than compiler $\texttt{comp}_1$. Once `mint-cogen` has been generated, the generation of compilers is quite efficient and so are the compilers and their target programs.

The figures confirm our earlier results. Generation time and size of the two- and three-level generating extensions are practically the same (Table 3). There is no penalty for multi-level staging. The results demonstrate that both pipelines can yield substantial speedups by reducing `mint`'s interpretive overhead: they improve the performance by a factor 30 and 72, respectively.

7. Related work

7.1. Cogen approach

The first hand-written compiler generator based on partial evaluation techniques was, in all probability, the system *RedCompile* for a dialect of Lisp [4]. Romanenko [44] gave transformation rules that convert annotated first-order programs into two-level generating extensions.

Holst [31] 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" [7]. Holst and Launchbury [32] 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 [5, 6] used the cogen approach for the Core Standard ML language, Bondorf and Dussart [12] combined the approach with cps-based specialization for the λ-calculus, and Andersen [1] developed a compiler generator for the ANSI C language. Lawall and Danvy describe how control operators can be used in hand-written compiler generators for functional languages [40]. Recently, Jørgensen and Leuschel [38] adapted the cogen approach to logic languages. None of them considers multi-level specialization.

The multi-cogen approach presented here is based on earlier work [26, 27]. Thiemann [49] extended our approach using a continuation-based specialization style.

7.2. Multi-level languages

In the area of programming languages, Tennent [48] appears to be the first who made an informal distinction between computations at compile-time and at run time in denotational definitions. Nielson and Nielson [42] investigated two-level functional languages and showed that binding-time analysis can be expressed as a form of type checking; the possibility of multi-level languages is mentioned. Jones [36] were the first to use binding-time analysis for self-application of a partial evaluator; the reader is referred to the book [35] for a comprehensive presentation. These investigations grew out of the field of semantics-directed compilation [34].

Recently, Nielson and Nielson [43] presented an algebraic description of a *multi-level lambda-calculus* as a common framework for existing multi-level systems which also describes a restricted version of our multi-level binding-time analysis. They assume that the underlying language is typed which simplifies their treatment of our system, but their framework is more general and can be used to formulate several other existing multi-level (two-level) systems in such diverse areas as code generation and abstract interpretation.

Taha and Sheard [47] introduce MetaML, a statically typed multi-level language for hand-writing multi-level generating extensions. Although the multi-level language in Section 4.1 has not been 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).

The only other multi-level programming language we are aware of is *Alloy* [3], a logic language which provides facilities for deductions at different meta-levels and a proof system for interleaving computations at different metalevels. It is less obvious how to write program generators directly in such a language. However, there is a close connection between certain

logical systems and computation staging as pointed out by Davies and Pfenning [19] who give a framework for expressing staged computation in modal logic.

7.3. Self-application

Several approaches have been employed in order to improve the performance of self-application: *action trees* [17], an offline method that compiles the information obtained by the binding-time analysis into directives for a partial evaluator, and *freezer* (metasystem jumps) [51] in supercompilation, an online method that allows the evaluation of partially known expressions by the underlying implementation. The multi-cogen approach is a very effectice alternative to multiple self-application in partial evaluation; the multi-level generator could possibly take advantage of action trees, while the freezer is targeted towards online transformers.

8. Conclusion

We investigated the theoretical limitations and practical problems of standard specialization tools, introduced multi-level program specialization, and designed and implemented a program generator for efficient multi-level specialization. The results demonstrate that, in connection with the multi-cogen approach, multi-level specialization is far more practically than previously supposed. This becomes apparent when compared to previous attempts of multi-level specialization by self-application. The multi-level generator presented here converts programs into very compact multi-level generating extensions that guarantee fast successive specialization. Our approach to multi-level specialization seems well-suited for applications where generation time and program size are critical.

The multi-level generator permits to set up (any combination of) online and offline specialization pipelines which allows us to choose the approach that is suited best for a given problem. It permits applications of partial evaluation that would otherwise not have been possible, such as the generation of a program-transforming compiler generator from an instrumented interpreter.

The multi-cogen approach takes standard (two-level) offline partial evaluation to its extreme: offline partial evaluation for multi-level specialization. It inherits the advantages and disadvantages of standard offline partial evaluation, such as a rigid control of a specializer's behavior by a separate binding-time analysis and fixing the number of stages and their binding-time classification before the actual specialization. On the other hand, the multi-level binding-time analysis allows to predict the overall specialization behavior and may provide valuable feedback to the user before running an offline specialization pipeline which may be essential in practice. Indeed, one could argue that, in many realistic applications, predictability is even more important for multi-level specialization than for two-level specialization. Limitations of partial evaluation-based program transformation compared to other methods for program transformation have been discussed elsewhere [46].

As a side-effect we designed a multi-level programming language which, together with the typing rules, can be seen as a statically-typed programming language for hand-writing

program generators. Instead of using a multi-level program generator to obtain multi-level generating extensions automatically from 'ordinary' programs, one could hand-write program generators, thus possibly providing a more fine-grained control over a program generator to a programmer (an approach advocated in [47]).

The work presented here clarifies theoretical concepts of program specialization and makes use of (or generalizes) several state-of-the art techniques of partial evaluation including fast program analysis, efficient code generation and program annotation. It also reflects how the field of partial evaluation has constantly evolved over the last decade since the first self-applicable offline partial evaluator was put to work for a toy language [37].

Appendix

Meta-interpreter

This appendix describes the meta-interpreter used in the experiment in Section 6.3. We show the code of the meta-interpreter, the semantics definition, and the example program.

Example definition. The example semantics defines a small functional language: an applied lambda calculus with constants, variables, conditionals, abstractions, applications and fix-expressions. As meta-language we will use a form of denotational semantics closely resembling that of Schmidt [45]. For simplicity we leave out syntax definitions and semantics algebras and write $E[\![e]\!]\rho = r$ instead of $E[\![e]\!] = \lambda\rho.r$. The specific form of the meta-language will be described below when we introduce the meta-interpreter. The denotational semantics of the example language has the following valuation function:

$$
\begin{array}{ll}
E[\![c]\!]\rho & = c \\
E[\![o]\!]\rho & = ext[\![o]\!] \\
E[\![x]\!]\rho & = lup(\rho, x) \\
E[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\rho & = E[\![e_1]\!]\rho \rightarrow E[\![e_2]\!]\rho \square E[\![e_3]\!]\rho \\
E[\![\lambda x.e]\!]\rho & = \lambda u.E[\![e]\!](upd(x, u, \rho)) \\
E[\![e_1\ e_2]\!]\rho & = E[\![e_1]\!]\rho\ (E[\![e_2]\!]\rho) \\
E[\![\texttt{fix } e]\!]\rho & = fix(E[\![e]\!]\rho)
\end{array}
$$

The semantics in the form that constitutes the input to the meta-interpreter:

```
; Valuation function:
(Val E
   [(LCst c)       r = (ASV c)]                    ; constants
   [(LExt o)       r = (ext (ASV o))]              ; externals
   [(LVar v)       r = (lup r v)]                  ; variables
   [(LIf e1 e2 e3) r = (if (vfa E (ASV e1) r)      ; conditional
                           (vfa E (ASV e2) r)
                           (vfa E (ASV e3) r))]
```

```
[(LAbs v e2)    r = (abs u (vfa E (ASV e2)       ; lambda abstraction
                                  (upd v u r)))]
[(LApp e1 e2)   r = (app (vfa E (ASV e1) r)      ; application
                         (vfa E (ASV e2) r))]
[(LFix e)       r = (fix (vfa E (ASV e) r))])    ; fix
```

Example program. The example program written in the language defined by valuation function E is the factorial

$$\texttt{(fix } \lambda fac.\lambda x.\texttt{if } x = 0 \texttt{ then } 1 \texttt{ else } x * \texttt{(}fac\ \texttt{(}x - 1\texttt{)))}\ input$$

The program represented as abstract syntax tree:

```
; factorial program:
(LApp
  (LFix
    (LAbs fac
      (LAbs x (LIf (LApp (LApp (LExt =) (LVar x)) (LCst 0))
                   (LCst 1)
                   (LApp (LApp (LExt *) (LVar x))
                         (LApp (LVar fac)
                               (LApp (LApp (LExt -) (LVar x))
                                     (LCst 1))))))))
  (LVar input))
```

Meta-interpreter. The meta-interpreter takes as input a semantics definition `def*`, the name of the "goal" valuation function `vf`, a program `ast` written in the defined language (represented as abstract syntax tree), and input `i` of the program.

A semantics definition is a list of valuation functions. Each valuation function consists of a list of *alternatives* where each alternative describes the translation of one syntactic form into its meaning. Alternatives have the form

$$F[\![pat]\!]\rho = rhs$$

where F is the name of the valuation function, *pat* an abstract syntax pattern, ρ an environment, and the right-hand side *rhs* is an expression of the meta-language. A pattern describes syntactic forms using abstract syntax variables (asv) ranging over fragments of abstract syntax; they are assumed to be linear (no repeated occurrences of variables). The environment ρ maps program variables to values.

The expression *rhs* has one of the following forms: a constant c, an operation assigning meaning to an externally defined operation $ext[\![o]\!]$ (e.g., addition of integers), an operation *lup* looking up entries in the environment, a conditional, an abstraction $\lambda u.e$, an application $e\ e$, a fix point operator $fix(e)$, and a valuation function application. The application of a valuation function F can have two forms: $F[\![v]\!]\rho$ or $F[\![asv]\!](upd(x, u, \rho))$ where $upd(x, u, \rho)$ is the environment ρ updated such that it maps x to u. The first argument to the call can only be an abstract syntax variable which ensures that definitions are

compositional. For simplicity, we assumed that the meta-language uses only one environment (one can easily extend the meta-interpreter so that it can handle any number of environments).

The code of the meta-interpreter is as follows:

```
(loadt "mini.adt")

(define (meta def* vf ast i)
  ((DE def*) vf ast (lift-to ast (lambda (x) i))))

; Interpret definitions:
(define (DE def*)
  (meta-fix
    (lambda (f)
      (lambda (vf ast r)
        (MD def* f vf ast r)))))

; Match definitions:
(define (MD def* f vf ast r)
  (if (null? def*)
      (meta-error 'MD "Unknown valuation function: ~s" vf)
      (let* ([def1 (def*->def1 def*)] [vf1 (def->vf def1)])
        (if (equal? vf vf1)
            (MA (def->a* def1) ast f r)
            (MD (def*->def* def*) f vf ast r)))))

; Match alternatives:
(define (MA a* ast f r)
  (if (null? a*)
      (meta-error 'MA "Failed to match alternative: ~s" ast)
      (let* ([a (alt*->alt1 a*)] [pat (alt->pat a)])
        (P pat ast (init-ast-venv)
           (lambda (s) (ME (alt->rhs a) f s r)) ; success
           (lambda (s) (MA (alt*->next-alt* a*) ast f r))))))) ; failure

; Match pattern:
(define (P pat ast s sc fc)
  (cond
   ((pvar? pat)
    (sc (lambda (w) (if (equal? pat w) ast (s w)))))
   (else ;(pcon? pat)
    (if (equal? (ast->con ast) (pat->con pat))
        (PM (cpat->pat* pat) (cast->ast* ast) s sc fc)
        (fc s)))))

; Match multiple patterns:
(define (PM pat* ast* s sc fc)
  (if (null? pat*)
      (sc s)
      (P (car pat*) (car ast*) s
         (lambda (s1) (PM (cdr pat*) (cdr ast*) s1 sc fc))
         sc)))
```

```
; Interpret meta-expressions:
(define (ME e f s r)
  (cond
   ((cst? e) (cst->cst e))
   ((ext? e) (ext (s (asv->name (ext->o e)))))
   ((asv? e) (s (asv->name e)))
   ((if? e)  (if (ME (if->e1 e) f s r)
                 (ME (if->e2 e) f s r)
                 (ME (if->e3 e) f s r)))
   ((abs? e) (lambda (v)
               (ME (abs->body e) f s
                   (lambda (w)
                     (if (equal? (abs->var e) w) v (r w))))))
   ((app? e) ((ME (app->e1 e) f s r) (ME (app->e2 e) f s r)))
   ((vfa? e) (f (vfa->vf e) (s (asv->name (vfa->ast e)))
                (let ((ee (vfa->env-exp e)))
                  (if (upd? ee)
                      (lambda (w) (if (equal? (s (upd->var ee)) w)
                                      (r (upd->val ee))
                                      (r w)))
                      r))))
   ((fix? e) (fix (ME (fix->body e) f s r)))
   ((lup? e) (r (s (lup->avar e))))
   (else     (meta-error 'R "unknown meta syntax: ~s" e))))

; Fix-point operator of the semantic language, i.e.,
; the source language of the denotational definitions.
(define (fix f) (lambda (x) ((f (fix f)) x)))
; Fix-point operator used in the definition of the meta-language.
(define (meta-fix f) (lambda (x y z) ((f (meta-fix f)) x y z)))
; Initial asv-environment
(define (init-ast-venv)
  (lambda (w)
    (meta-error 'asv-venv "unknown abstract syntax variable ~s" w)))
; Function that makes sure that y has same bt as x
; i.e., it ``lifts'' y to x's level
(define (lift-to x y) y)
```

The target program. The target program $\texttt{tar}_2$ obtained by the offline pipeline approach (see Subsection 6.3) looks as follows:

```
(loadt "mini.adt")
(define (f-lambda-15$1-5 x$15)
  (lambda (x$16) ((x$15 (f-lambda-15$1-5 x$15)) x$16)))
(define (f-if-9$1-4 x$13 x$12 x$11)
  (if (((ext '=) x$13) '0)
      '1
      (((ext '*) x$13) (x$12 (((ext '-) x$13) '1)))))
(define (f-lambda-10$1-3 x$8 x$7) (lambda (x$9) (f-if-9$1-4 x$9 x$8 x$7)))
(define (f-lambda-10$1-2 x$4) (lambda (x$5) (f-lambda-10$1-3 x$5 x$4)))
(define (_sim-goal$1-1 x$1)
  ((let ((x$14 (f-lambda-10$1-2 x$1)))
     (f-lambda-15$1-5 x$14))
   x$1))
```

By simple post unfolding (would be implemented in a more realistic system) of this we would get:

```
(define (f-lambda-15$1-5 x$15)
  (lambda (x$16) ((x$15 (f-lambda-15$1-5 x$15)) x$16)))
(define (f-if-9$1-4 x$13 x$12 x$11)
  (if (((ext '=) x$13) '0)
      '1
      (((ext '*) x$13) (x$12 (((ext '-) x$13) '1)))))
(define (_sim-goal$1-1 x$1)
  ((f-lambda-15$1-5 (lambda (x$9) (f-if-9$1-4 x$9 x$1 x$4))) x$1))
```

The first of these functions is a copy the function `fix` from the meta-interpreter and the second which is the actual code of the target program is almost the same as the original expression. This shows that the two layers of interpretation have been removed.

Acknowledgments

Special thanks to Anders Bondorf for stimulating discussions about self-application and partial evaluation. Thanks to Torben Amtoft, Olivier Danvy, Neil D. Jones, and Hanne Nielson for valuable discussions. Thanks to Eddy Bevers, Peter Sestoft, and Peter Thiemann for constructive comments on an earlier version of this paper. The anonymous referees provided valuable feedback that improved the final version. Last but not least, we would like to thank all members of the Topps group at DIKU for providing an excellent working environment.

The first author was partly supported by an Erwin-Schrödinger-Fellowship of the Austrian Science Foundation (FWF) under grant J0780 & J0964, the second author partly by the European HCM Network "Logic Program Synthesis and Transformation" and the Belgian GOA "Non-Standard Applications of Abstract Interpretation". The work was also supported by the project "Design, Analysis and Reasoning about Tools" funded by the Danish Natural Sciences Research Council.

References

1. Andersen, L.O. Program analysis and specialization for the C programming language. Ph.D. Thesis, DIKU Report 94/19. Dept. of Computer Science, University of Copenhagen, 1994.
2. Andersen, P.H. and Holst, C.K. Termination analysis for offline partial evaluation of a higher order functional language. In *Static Analysis*, R. Cousot and D. Schmidt (Eds.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 1145, pp. 67–82, 1996.
3. Barklund, J., Boberg, K., Dell'Acqua, P., and Veanes, M. Meta-programming with theory systems, In *Meta-Logics and Logic Programming*, K. Apt and F. Turini (Eds.), Logic Programming, MIT Press, pp. 195–224, 1995.
4. Beckman, L., Haraldson, A., Oskarsson, Ö., and Sandewall, E. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, **7**:319–357, 1976.
5. Birkedal, L. and Welinder, M. Partial evaluation of Standard ML. DIKU Report 93/22. Dept. of Computer Science, University of Copenhagen, 1993.

6. Birkedal, L. and Welinder, M. Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, **8**(3):191–208, 1995.
7. Birkedal, L. and Welinder, M. Hand-writing program generator generators. In *Programming Language Implementation and Logic Programming*, M. Hermenegildo and J. Penjam (Eds.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 844, pp. 198–214, 1994.
8. Bjørner, D., Ershov, A.P., and Jones, N.D. (Eds.). *Partial Evaluation and Mixed Computation*. North-Holland, 1988.
9. Bondorf, A. *Self-Applicable Partial Evaluation*. Ph.D. Thesis, Report 90/17. DIKU Dept. of Computer Science, University of Copenhagen, 1990.
10. Bondorf, A. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, **17**(1–3):3–34, 1991.
11. Bondorf, A. and Danvy, O. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, **16**:151–195, 1991.
12. Bondorf, A. and Jørgensen, J. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, **3**(3):315–346, 1993.
13. Bondorf, A. and Dussart, D. Improving cps-based partial evaluation: Writing cogen by hand. In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Technical Report 94/9. University of Melbourne, Australia, pp. 1–9, 1994.
14. Bulyonkov, M.A. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, **21**:473–484, 1984.
15. Cardone, F. and Coppo, M. Type inference with recursive types: Syntax and semantics. *Information and Computation*, **92**:48–80, 1991.
16. Consel, C. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, pp. 145–154, 1993.
17. Consel, C. and Danvy, O. From interpreting to compiling binding times. In *ESOP '90*, N.D. Jones (Ed.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 432, pp. 88–105, 1990.
18. Danvy, O., Glück, R., and Thiemann, P. (Eds.). *Partial Evaluation*. Springer-Verlag, *Lecture Notes in Computer Science*, vol. 1110, 1996.
19. Davies, R. and Pfenning, F. A modal analysis of staged computation. In *Symposium on Principles of Programming Languages*. ACM Press, pp. 258–270, 1996.
20. Ershov, A.P. On the essence of compilation. In *Formal Description of Programming Concepts*, E.J. Neuhold (Ed.). North-Holland, pp. 391–420, 1978.
21. Futamura, Y. Partial evaluation of computation process—An approach to a compiler-compiler. *Systems, Computers, Controls*, **2**(5):45–50, 1971.
22. Glenstrup, A. and Jones, N.D. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics*, D. Bjørner, M. Broy, and I.V. Pottosin (Eds.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 1181, pp. 273–284, 1996.
23. Glück, R. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, pp. 309–320, 1991.
24. Glück, R. On the generation of specializers. *Journal of Functional Programming*, **4**(4):499–514, 1994.
25. Glück, R. and Jørgensen, J. Generating transformers for deforestation and supercompilation. In *Static Analysis*, B. Le Charlier (Ed.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 864, pp. 432–448, 1994.
26. Glück, R. and Jørgensen, J. Efficient multi-level generating extensions for program specialization. In *Programming Languages, Implementations, Logics and Programs*, M. Hermenegildo and S.D. Swierstra (Eds.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 982, pp. 259–278, 1995.
27. Glück, R. and Jørgensen, J. Fast binding-time analysis for multi-level specialization. In *Perspectives of System Informatics*, D. Bjørner, M. Broy, and I.V. Pottosin (Eds.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 1181, pp. 261–272, 1996.
28. Gomard, C.K. and Jones, N.D. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming*, **1**(1):21–69, 1991.
29. Hatcliff, J. Mechanically verifying the correctness of an off-line partial evaluator. In *Programming Languages, Implementations, Logics and Programs*, M. Hermenegildo and S.D. Swierstra (Eds.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 982, pp. 279–298, 1995.

30. Henglein, F. Efficient type inference for higher-order binding-time analysis. In *Functional Programming and Computer Architecture*, J. Hughes (Ed.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 523, pp. 448–472, 1991.
31. Holst, C.K. Syntactic currying: Yet another approach to partial evaluation. Technical report. DIKU, Department of Computer Science, University of Copenhagen, 1989.
32. Holst, C.K. and Launchbury, J. Handwriting cogen to avoid problems with static typing. Working paper, 1992.
33. IEEE. Standard for the Scheme programming language. IEEE Std 1178-1990, Institute of Electrical and Electronics Engineers, 1991.
34. Jones, N.D. (Ed.). *Semantics-Directed Compiler Generation Proceedings*, Springer-Verlag, *Lecture Notes in Computer Science*, vol. 94, 1980.
35. Jones, N.D., Sestoft, P., and Søndergaard, H. An experiment in partial evaluation: the generation of a compiler generator. In *Rewriting Techniques and Applications*, J.-P. Jouannaud (Ed.). Springer-Verlag, *Lecture Notes in Computer Science*, vol. 202, pp. 124–140, 1985.
36. Jones, N.D., Sestoft, P., and Søndergaard, H. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, **2**(1):9–50, 1989.
37. Jones, N.D., Gomard, C.K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science, Prentice-Hall, 1993.
38. Jørgensen, J. and Leuschel, M. Efficiently generating efficient generating extensions in Prolog. In [18], pp. 238–262, 1996.
39. Jørring, U. and Scherlis, W. Compilers and staging transformations. In *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*. ACM, pp. 86–95, 1986.
40. Lawall, J. and Danvy, O. Continuation-based partial evaluation. In *ACM Conference on Lisp and Functional Programming*, ACM Press. pp. 227–238, 1994.
41. Mogensen, T.Æ. Partially static structures in a self-applicable partial evaluator. In [8], pp. 325–347, 1988.
42. Nielson, F. and Nielson, H.R. *Two-Level Functional Languages*, Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Vol. 34, 1992.
43. Nielson, F. and Nielson, H.R. Multi-level lambda-calculus: An algebraic description. In [18], pp. 338–354, 1996.
44. Romanenko, S.A. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In [8], pp. 445–463, 1988.
45. Schmidt, David A. *Denotational Semantics, a Methodology for Language Development*. Allyn and Bacon, 1986.
46. Sørensen, M.H., Glück, R., and Jones, N.D. A positive supercompiler. *Journal of Functional Programming*, 1996 (to appear).
47. Taha, W. and Sheard, T. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipiulation*. ACP Press, 1997 (to appear).
48. Tennent, R.D. *Principles of Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall, 1981.
49. Thiemann, P. Cogen in six lines. In *International Conference on Functional Programming*. ACM Press, 1996, pp. 180–189.
50. Thiemann, P. Towards partial evaluation of full Scheme. In *Reflection'96*, G. Kiczales (Ed.), pp. 105–116, 1996.
51. Turchin, V.F. *Refal-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, Massachusetts, 1989.
52. Wand, M. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, **3**(3):365–387, 1993.