Hand-Writing Program Generator Generators

Lars Birkedal & Morten Welinder

DIKU, Department of Computer Science
University of Copenhagen
DK–2100 Copenhagen Ø, Denmark
e-mail: birkedal@diku.dk & terra@diku.dk

Abstract. In this paper we argue that hand-writing a program generator generator has a number of advantages compared to generating a program generator generator by self-application of a partial evaluator. We show the basic principles of how to construct a program generator generator by presenting a program generator generator for a skeletal language, and we argue that it is not more difficult to use the direct approach than the indirect approach. Moreover, we report on some promising experiments made with a prototype implementation of a program generator generator for most of the Standard ML Core Language. To the best of our knowledge, our prototype is the first succesfully implemented hand-written program generator generator for a statically typed language.

1 Introduction

A large class of similar computational problems can be solved in two essentially different ways: either by a specific program for each problem or by a general, parameterized program solving all the problems. A specific program is almost always more efficient than the general program, but the general program tends to be easier to write, maintain, and, moreover, given a new problem one does not have to write a new specific program but instead one can employ the general program.

In this paper we study the problem of automatically turning a general, parameterized program into an efficient specialized program applicable to one given problem. This way we can obtain the best of two worlds, efficiency and generality. We study the problem, known as partial evaluation, in the context of Standard ML.

1.1 Computation in One Stage or More

Turning a general program into a specialized program can be conceived of as turning a one-stage program into a two-stage program. For instance, an interpreter for a language L is a general program which is parameterized on an L source program and the input to the L program. The interpreter is a one-stage program as the interpretation is performed in one step. A compiler on the other hand is a two-stage program. It takes the L program as input and produces a target program, which afterwards can be executed on the input and give the result. The target program can be conceived of as a specialized version of the interpreter in the sense that it can only be used for problems specified by the L program. Moreover, it is well-known that it is much more efficient to execute a compiled program than interpreting the original program, and that it is easier to write an interpreter than it is to write a compiler. The methods developed in this paper are sufficient to *automatically* turn a one-stage program like an interpreter into a two-stage program like a compiler.

1.2 Generating Program Generators

A compiler is a program generator as it returns programs as results. So what we are looking for is a program, which we will call `cogen` for historical reasons, that generates program generators. To simplify matters we assume without loss of generality that the general programs we want to specialize all take two arguments. The first one which describes the particular problem we call static, the other we call dynamic. The situation can then be pictured as in Figure 1, which is borrowed from [21]. The `cogen` program accepts a general program `p`, and turns it into a

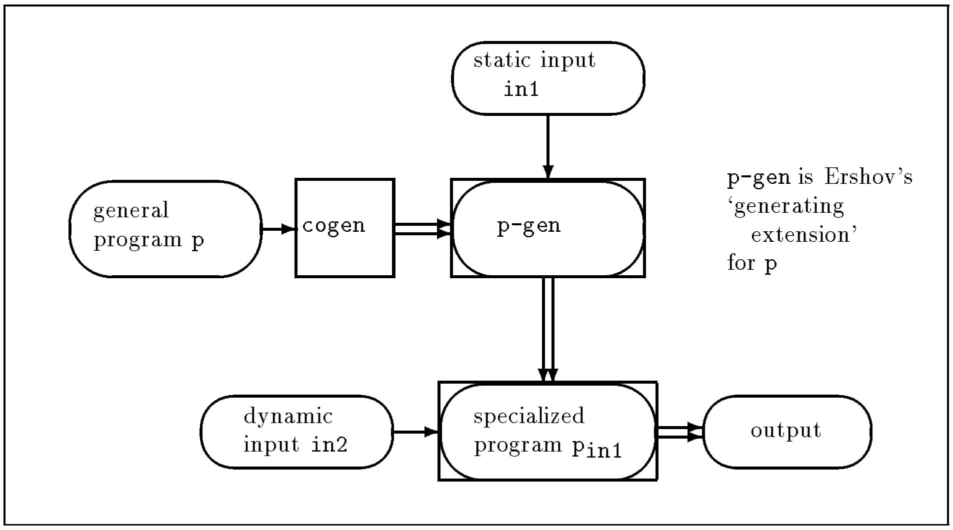

Figure 1: A Generator of Program Generators

program generator, `p-gen`. When `p-gen` is applied to the static argument of the general program, it produces a specialized program. This specialized program yields, when applied to the dynamic argument, a result which is equal to the result `p` would return when applied to both the static and the dynamic arguments. Notice how the one-stage program `p` has been turned into a two-stage program `p-gen`. The two-stage program `p-gen` is called the *generating extension* of `p`. In their recent book [21, Page 11] Jones, Gomard, and Sestoft remark that

> *It would be wonderful to have a program generator generator, but it is far from clear how to construct one.*

In this paper we report on the development of such a program generator generator for programs written in the Standard ML Core Language [28,27].

We will call the hand-writing of `cogen` for *the direct approach* to program generator generators. Traditionally, the `cogen` and `p-gen` boxes in Figure 1 have been merged into one box, called `mix`. The reason has been to simplify matters [21]. The `mix` program is called a *partial evaluator* and in fact it is possible to obtain a program generator generator indirectly by self-application of `mix` [13,22]. We will call this for *the indirect approach* to program generator generators.

A natural question then is: why consider developing a program generator generator directly if it is simpler to write `mix`? The reasons are that

– there are certain problems with the program generator generators obtained by self-application, in particular when it comes to specializing programs written in a statically typed language; and
– it turns out that with the methods we present, it is just as simple or difficult to write `cogen` as it is to write `mix`.

1.3 Outline

In Section 2 we discuss the problems with the traditonal indirect approach and how they can be avoided by using the direct approach. In Section 3 we show the basic principles of how to construct a program generator generator by presenting a program generator generator for a skeletal language. Thereafter, in Section 4, we report on the development of a `cogen` for essentially all of the SML Core Language, and on some experiments with a prototype implementation. Finally, we conclude and give directions for future work.

2 Program Generator Generators — What?

In this section we study program generator generators on an abstract level using equations to specify relationships between different programs. We start out by considering the indirect approach and discuss some problems with this approach. Then we consider the direct approach and argue that the direct approach avoids some of the problems with the indirect approach. In both cases we shall be concerned with program generator generators for programs written in a statically typed language since we want the principles to work not only for dynamically typed programming languages.

2.1 The Indirect Approach to Program Generator Generators

A partial evaluator can be characterized by whether it is *on-line* or *off-line.* An on-line partial evaluator decides on the fly which operations can be performed at specialization time (i.e., when `mix` is executed), and which must be deferred to runtime. The partial evaluators described in [26,25,31] are all on-line. In off-line partial evaluators it is decided *a priori* to the specialization proper which operations to perform at which time. This is done by a *binding-time analysis.* The partial evaluators described in [7,15,2,23] are all off-line and self-applicable.

It has been argued that off-line specialization is superior when the goal is self-application of the specializer [10]. It is in fact the case that most successfully implemented self-applicable partial evaluators have been off-line. In this section we shall only be concerned with self-applicable partial evaluation as our goal is to be able to obtain a program generator generator. Moreover, we shall assume that a program to be specialized is first binding-time analysed, then annotated on the basis of the binding-time analysis and that the annotated program is then given to the partial evaluator as input. It is well-understood how to add such annotations (e.g., [21,2,16]) and we will not discuss it here.

Notation. We need some notation in order to distinguish between program texts, values, program representations, and program meanings as we shall deal with all four concepts.

For any program text, `p`, written in language L we let $[\![\mathtt{p}]\!]_L$ denote the meaning of the program according to the semantics of the language L. Alternatively we may say that $[\![\,\cdot\,]\!]_L$ is an interpreter (as an abstract mathematical function) for the language L. We will often omit the subscript when there is no chance for misunderstandings.

An interpreter written in a statically typed language must represent the values of the language it interprets in some uniform data type Val. The reason is that the interpreter itself is a fixed program and has a bounded number of types. There is no upper limit on the number of different types that can occur in a program to be interpreted. This means that the interpreter must code at least some of the object types in a general type.

We consider self-application so we assume that the interpreter is written in the language it interprets. When `v` is a value of some type we let $\overline{\mathtt{v}}^{\mathrm{Val}}$ be the encoding of `v` into the value data type. Likewise, let $\overline{\mathtt{p}}^{\mathrm{Prg}}$ be the encoding of a program `p` into a data type Prg. We will assume that the function $\mathtt{p} \mapsto \overline{\mathtt{p}}^{\mathrm{Prg}}$ is injective, i.e., that we can remove the representation when we need to.

An Equational Specification of the Indirect Approach. We now reformulate the Futamura projections [13], in the setting of a statically typed language in order to explain the above mentioned problems. This has also been done by Andersen [1,2] and Launchbury [23].

Recall that a partial evaluator is a program, **`mix`**, which when evaluated with an annotated version, **`p-ann`**, of a given program, `p`, and part of that program's input data, `s`, as actual parameters yields a program which when evaluated with the rest of the programs input data, `d`, yields the same result as the original program `p` would have given when applied to all its input data. A partial evaluator, **`mix`**, can thus be characterized by the following so-called mix-equations. If

$$\overline{\mathtt{p}'}^{\mathrm{Prg}} = [\![\mathtt{mix}]\!]\left(\overline{\mathtt{p\text{-}ann}}^{\mathrm{Prg}}, \overline{\mathtt{s}_1}^{\mathrm{Val}}\right) \tag{1}$$

then

$$[\![\mathtt{p}]\!]\ \mathtt{s}_1\ \mathtt{d}_2 = [\![\mathtt{p}']\!]\ \mathtt{d}_2 \tag{2}$$

with the same termination properties of the two sides.

Using a self-applicable partial evaluator it is possible to obtain a compiler and a program generator generator. This is expressed by the equations 3–5 shown below, now known as the Futamura projections. Let `int` be an interpreter for some programming language L and let `p` be a program written in L. We then have the following provided the partial evaluator `mix` terminates.

$$\overline{\mathtt{target}}^{\mathrm{Prg}} = [\![\mathtt{mix}]\!] \left(\overline{\mathtt{int\text{-}ann}}^{\mathrm{Prg}}, \overline{\mathtt{p}}^{\mathrm{Val}}\right) \tag{3}$$

$$\overline{\mathtt{compiler}}^{\mathrm{Prg}} = [\![\mathtt{mix}]\!] \left(\overline{\mathtt{mix\text{-}ann}}^{\mathrm{Prg}}, \overline{\overline{\mathtt{int\text{-}ann}}^{\mathrm{Prg}}}^{\mathrm{Val}}\right) \tag{4}$$

$$\overline{\mathtt{cogen}}^{\mathrm{Prg}} = [\![\mathtt{mix}]\!] \left(\overline{\mathtt{mix\text{-}ann}}^{\mathrm{Prg}}, \overline{\overline{\mathtt{mix\text{-}ann}}^{\mathrm{Prg}}}^{\mathrm{Val}}\right) \tag{5}$$

The first projection states that compilation can be done by specialization of an interpreter with respect to a program. The next equation states that a stand-alone compiler can be generated by specializing the partial evaluator itself with respect to an interpreter, and finally, the last projection expresses that a stand-alone program generator generator can be generated by specializing the specializer with respect to the specializer. Given an interpreter, `cogen` produces a compiler. The Futamura projections are more thoroughly studied, albeit in an untyped setting, in [19].

The `cogen` obtained by self-application can, of course, be used not only on interpreters but also on other programs, so we have now seen how a program generator generator can be obtained by self-applying a partial evaluator. As already mentioned, there are some problems with this approach, however, which we discuss in the following section.

The Coding Problems. There are two problems related to the encoding of values and programs: double-encoding and unnecessary tagging and untagging operations in the generated two-stage programs. We now consider these problems in turn.

The double-encoding problem refers to the fact that the value arguments are doubly-coded when `mix` is self-applied, as can be observed from the typed Futamura projections above. To understand this problem let us as an example assume that the universal value data type for an SML-like language is defined this way

```
datatype Val =
  Int of int                      (* Base type *)
| ...                             (* Other base types *)
| Record of (string * Val) list   (* Records *)
| Cons0 of string                 (* Constructed values without arg. *)
| Cons of string * Val            (* Constructed values with arg. *)
| ...
```

Base types are represented by simply adding a tag, records as lists of labels and values, and constructed values as the name of the constructor plus its argument, if present. Programs are represented as their abstract syntax trees using some data type. For example we would represent the two-element list `[10,20]` (which in Standard ML is a syntactically sugared way of writing the expression `(op::){1=10,2=(op::){1=20,2=nil}}`) by

```
Cons("::",Record[("1",Int 10),
                ("2",Cons("::",Record[("1",Int 20),
                                      ("2",Cons0 "nil")]))])
```

The doubly-coded version of the list is so large that we will not show it here, and it requires little imagination to see that a doubly-coded program (the list might be a tiny part of a program) is a gigantic constructed value. Experience shows that naïve double-encoding is prohibitively expensive for self-application [23].

There are ways out of the double-coding problem. In [1] and [23] the trick is to enhance the universal type with a subtype, program, at the same level as the ground types instead of having programs represented as constructed values. In [12] the trick is to treat all values (and thus programs as well) as having one and the same "black-box" type inherited from the implementation language of the partial evaluator. But then, of course, the implementation language must be dynamically typed.

The use of the described encoding gives rise to another problem in traditional partial evaluation: some unnecessary encoding and decoding operations may be left in the residual program. As an example [1, Section 6.1] consider specialization of a self-interpreter with respect to the power function which can be defined as follows in SML.

```
fun pow (n:int) (x:int) =
  if n=0 then
    1
  else
    (x * pow (n-1) x)
```

We might end up with something like this

```
fun pow (n:Val) (x:Val) : Val =
  if val_to_int n = 0 then
    int_to_val 1
  else
    int_to_val (val_to_int x *
                (val_to_int (pow (int_to_val (val_to_int n - 1)) x)))
```

where `int_to_val` injects an integer into the universal type[1] and `val_to_int` is its partial inverse.[2] We would have liked the resulting power program to be essentially the same as the original as it would have been for an optimal **mix**, see [21, Section 6.4]. To overcome this problem, an untagging analysis is created in [1].

[1] So if we used the above data type it would simply be the `Int` tag.

[2] Which with the above data type would be `(fn (Int i) => i)`.

Recall that any traditional partial evaluator has an embedded self-interpreter. From the example above we therefore learn that two-stage programs (e.g., compilers) generated by self-application of `mix` can contain a lot of unnecessary tagging and untagging operations; of course, it depends on `mix`, but to the best of our knowledge, there is no implemented self-applicable partial evaluator for a statically typed language which does not have this problem.

2.2 The Direct Approach to Program Generator Generators

In this section we give an equational description of a hand-written `cogen` and argue that the above mentioned problems are avoided. Moreover, we shall see that there are other benefits with the direct approach to program generator generators.

The idea of hand-writing `cogen` instead of writing a partial evaluator `mix` seems to have originated with [4] in which the authors report on a so-called partial evaluation compiler, *RedCompile*, which is a hand-written compiler generator for a subset of Lisp. *RedCompile* is semi-automatic: it relies on user annotation of functions to ensure preservation of semantics.

In [17] Carsten Kehler Holst describes what he calls "syntactic currying." It is basically the same as hand-writing a program generator generator. The language used is a minimal subset of Scheme without higher-order functions and side effects. The system is implemented in Scheme using macros. Perhaps due to the choice of language, Holst did not realize the real value of his findings when used on a typed language. However, he and Launchbury did that later in the working note [18] where it was argued that hand-writing `cogen` and performing partial evaluation in two stages is advantageous when it comes to partial evaluation of statically typed languages.

To the best of our knowledge, our hand-written `cogen` for most of the SML Core Language is the first succesfully implemented hand-written `cogen` for a statically typed language.

An Equational Specification of the Direct Approach. Assume we have written a program generator generator `cogen` directly. To compare with the traditional approach lets consider how specialization of a program `p` with respect to the static part of input, `s`, proceeds. The equations are as follows (where `d` is the dynamic input).

$$\overline{\texttt{p-gen}}^{\mathrm{Prg}} = [\![\texttt{cogen}]\!]_{L_1}\ \overline{\texttt{p-ann}}^{\mathrm{Prg}} \tag{6}$$

$$\overline{\texttt{p-res}}^{\mathrm{Prg}} = [\![\texttt{p-gen}]\!]_{L_2}\ \texttt{s} \tag{7}$$

$$\texttt{result} = [\![\texttt{p-res}]\!]_{L_3}\ \texttt{d} \tag{8}$$

and the correctness criterion is again that

$$\texttt{result} = [\![\texttt{prg}]\!]_{L}\ \texttt{s d} \tag{9}$$

with equal termination properties. Notice that `cogen`, `p-gen`, and `p-res` do not have to be written in the same language. However, things are simpler if L_2 and L_3 are the same language as the one subjected to partial evaluation (or very close to).

Advantages of the Direct Approach.

1. As is apparent from the equations, a hand-written program generator generator does not have to manipulate values of the object program — all it does is to export each part of the annotated program syntax tree to either the generating extension or to the residual program, or rather, as the residual program does not exists when `cogen` is run, arrange for the generating extension to export part of the syntax tree to the residual program.
   Hence the coding problem is avoided in the direct setting and so is the the double-coding problem as there is no self-application involved. Moreover, in the direct approach no universal data type is used for values during specialization, so there is no need for an untagging analysis. For instance, a compiler generated by the direct approach will typically be more efficient than a compiler generated using the indirect approach.[3]
2. `p-gen` can use all features of the L_2 language, i.e., there are no restrictions on the use of language features during specialization. This is opposed to the indirect approach, where the specializer, `mix`, must only use those features of the language, which it treats (due to self-application).
3. There is no restriction on the language in which the program generator generator is written. For instance, it is perfectly possible to write a `cogen` in C which treats SML programs.
4. A hand-written `cogen` can be (but need not be) more efficient than one generated by self-application, since a `cogen` generated by self-application typically performs time consuming environment manipulations (the environment is typically inherited from the specializer) whereas a hand-written `cogen` does not need an enviroment since it only manipulates syntax trees.
5. The specialization process in the direct approach, see equation (7), is likely to be faster than the corresponding specialization process in the indirect approach, see equation (3). This is because the former corresponds to running a compiled program while the latter corresponds to interpreting a program. Andersen [2] reports the difference to be an order of magnitude.
6. When using the direct approach, there is no need to write a self-interpreter, as in the indirect approach. This is important in practice because it requires some effort to write a self-interpreter for realistic languages like SML [5].

3 Program Generator Generators — How

We have now argued that it is advantageous to use the direct approach and hand-write a program generator generator. In this section we will demonstrate that it is in fact not much more difficult to hand-write a `cogen` than to hand-write a specializer. We do this by presenting a program generator generator for programs written in a small subset of Standard ML. We only consider a small subset of SML to ease the presentation and due to space limitations. Essentially, the small subset

[3] Of course, it again depends on the specializer but to the best of our knowledge no implemented specializer for a statically typed language has avoided this problem.

is an extended simply typed call-by-value lambda calculus. Moreover, we will not be concerned with how to generate an annotated program. Instead we define a two-level language in which annotations are made explicit in the syntax [30,15] and develop a `cogen` for this particular language. The reader is referred to the literature for a description of how to generate a two-level language on the basis of a binding-time analysis, see e.g. [15,11,2,6].

3.1 Two-level Mini ML

The grammar of the Two-level Mini ML language we consider is shown in Figure 2. We use $\langle\ \rangle$ to enclose optional phrases, *var* ranges over a set of variables, i ranges over integer constants, and *op* ranges over base operations like `+` and `-`. There is only one base type, integer, in Two-level Mini ML, and `lift` is used, as usual, for a static expressions of base type in a dynamic context. Underlined value bindings are used for precisely those functions which are to be specialized (memoization must take place for these functions) — all other functions are to be unfolded; in particular, all calls to lambda abstractions (`fn` *2match*) are to be unfolded and hence we only have one kind of lambda abstraction in the syntactic class *2exp* of two-level expressions. Otherwise, the convention is that underlined phrases are to be residualized and that non-underlined phrases are to be executed during specialization, i.e., when the generating extension is run. Note that specialization functions take precisely one static and one dynamic argument; this is the reason for the occurrence of both "`fn`" and "$\underline{\texttt{fn}}$". Two-level Mini ML has simple pattern matching to choose between several branches (as in SML); underlined patterns are only used for dynamic arguments of specialization functions, and to ease the presentation we will assume that there is exactly one variable in the dynamic argument match rules. We only consider specialization of *well-annotated* two-level

$$
\begin{array}{lcl}
\mathit{2dec} & ::= & \texttt{val rec}\ \mathit{2valbind} \\[4pt]
\mathit{2valbind} & ::= & \mathit{var}\ \underline{\texttt{=}}\ \texttt{fn}\ \mathit{var}\ \texttt{=>}\ \underline{\texttt{fn}}\ \mathit{2match} \\
 & | & \mathit{var}\ \texttt{=}\ \texttt{fn}\ \mathit{2match} \\
 & | & \mathit{2valbind}_1\ \texttt{and}\ \mathit{2valbind}_2 \\[4pt]
\mathit{2exp} & ::= & \mathit{var} \mid \underline{\mathit{var}} \mid i \mid \texttt{fn}\ \mathit{2match} \mid \mathit{2exp}_1\ \texttt{@}\ \mathit{2exp}_2 \mid \mathit{2exp}_1\ \underline{\texttt{@}}\ \mathit{2exp}_2 \\
 & | & \mathit{2exp}_1\ \mathit{op}\ \mathit{2exp}_2 \mid \mathit{2exp}_1\ \underline{\mathit{op}}\ \mathit{2exp}_2 \mid \texttt{lift}\ \mathit{2exp} \\[4pt]
\mathit{2match} & ::= & \mathit{2mrule}\ \langle\ \texttt{|}\ \mathit{2match}\rangle \mid \mathit{2mrule}\ \langle\ \underline{\texttt{|}}\ \mathit{2match}\rangle \\[4pt]
\mathit{2mrule} & ::= & \mathit{2pat}\ \texttt{=>}\ \mathit{2exp} \mid \mathit{2pat}\ \underline{\texttt{=>}}\ \mathit{2exp} \\[4pt]
\mathit{2pat} & ::= & \mathit{var} \mid \underline{\mathit{var}} \mid i \mid \underline{i}
\end{array}
$$

Figure 2: Two-level Mini ML Grammar

programs [15]. Type rules for checking well-annotatedness are simple to write down (see e.g., [15,6]) and have been omitted for space reasons.

3.2 A Program Generator Generator for Two-level Mini ML

Let us begin with two simple examples.

First consider the Two-level Mini ML expression `2 + 3`. This is a non-underlined expression and hence the expression should be evaluated when the generating extension is run. This is accomplished by simply copying the expression into the generating extension.

Next consider the Two-level Mini ML expression `d` $\underline{\texttt{+}}$ `lift 2`. The underlining tells us that the variable `d` will be bound to code when the generating extension is run, and that code must be emitted for the application of the `+` operator when the generating extension is run. To describe what code `cogen` then must generate for the expression, we shall use the following notation: $\lceil$`ml-code`$\rceil$ is an expression that evaluates to the (syntax tree of) `ml-code` except that dot-underlined structures inside are evaluated beforehand. For our example under consideration, we will generate the code: $\lceil$`d + lift 2`$\rceil$, which in a concrete implementation can be equal to: `build_add(d,lift 2)`. When the generating extension is run, this expression evaluates to a piece of code, consisting of an application of `+` to the piece of code that `d` is bound to and a piece of code which evaluates to 2, i.e., a syntax tree for 2.

A syntax-directed `cogen` for Two-level Mini ML based on the observations from these two examples is shown in Figures 3 and 4. It is defined by a collection of semantic functions, one function $\mathcal{C}_{2phrase}$ for each syntactic two-level phrase class.

Most of these semantic functions, except the one for declarations, are typed in a uniform manner: they take a *2phrase* and return either a *phrase* or an expression (a piece of code) which *evaluates to* a *phrase*. We use $\underline{phrase}$ to denote the type *exp* of expressions (code) which evaluates to *phrase* [20,21]. This choice has been made such that it is simple to ensure type-correctness of the generating extension. We conjecture, but thave not proved, that the generating extension and the residual program are always well-typed according to the static semantics of SML. For example, the type of $\mathcal{C}_{2exp}$ is $2exp \rightarrow exp/\underline{exp}$ meaning that it takes a *2exp* as argument and gives either an *exp* or an $\underline{exp}$ as result. In other words the result is an expression either at specialization time or at residual time. We generally omit injections and projections in the meta-language to avoid cluttering the notation.

The treatment of expressions, match rules, and patterns follow from the observations from our two small examples above (except for underlined variables, which we discuss below).

The most interesting case is the case for specialization functions. First the meta-variable *sp_seenB4list* is bound to a fresh variable name, and the meta-variable x_i is bound to the dynamic variable in the match rules. With the use of these meta-variables, code is then generated. The idea of this code is best explained by considering how the code is executed when the generating extension is run.

The variable *sp_seenB4list* is bound to an updatable, initially empty list. This variable is used to keep track of the static values with respect to which the specialization function has been specialized for so far (each static argument is paired with the name of the corresponding residual function) and corresponds to the usual

$$\mathcal{C}_{2exp} : \mathit{2exp} \rightarrow \mathit{exp}/\underline{\mathit{exp}}$$

$$
\begin{array}{lcl}
\mathcal{C}_{2exp}[\![\mathrm{i}]\!] & = & \mathrm{i} \\
\mathcal{C}_{2exp}[\![\mathit{var}]\!] & = & \mathit{var} \\
\mathcal{C}_{2exp}[\![\underline{\mathit{var}}]\!] & = & \lceil \mathit{var} \rceil \\
\mathcal{C}_{2exp}[\![\texttt{fn}\ \mathit{2match}]\!] & = & \texttt{fn}\ (\mathcal{C}_{2match}[\![\mathit{2match}]\!]) \\
\mathcal{C}_{2exp}[\![\mathit{2exp}_1\ \texttt{@}\ \mathit{2exp}_2]\!] & = & \mathcal{C}_{2exp}[\![\mathit{2exp}_1]\!]\ \mathcal{C}_{2exp}[\![\mathit{2exp}_2]\!] \\
\mathcal{C}_{2exp}[\![\mathit{2exp}_1\ \underline{\texttt{@}}\ \mathit{2exp}_2]\!] & = & \lceil \mathcal{C}_{2exp}[\![\mathit{2exp}_1]\!]\ \mathcal{C}_{2exp}[\![\mathit{2exp}_2]\!] \rceil \\
\mathcal{C}_{2exp}[\![\mathit{2exp}_1\ \mathit{op}\ \mathit{2exp}_2]\!] & = & \mathcal{C}_{2exp}[\![\mathit{2exp}_1]\!]\ \mathit{op}\ \mathcal{C}_{2exp}[\![\mathit{2exp}_2]\!] \\
\mathcal{C}_{2exp}[\![\mathit{2exp}_1\ \underline{\mathit{op}}\ \mathit{2exp}_2]\!] & = & \lceil \mathcal{C}_{2exp}[\![\mathit{2exp}_1]\!]\ \mathit{op}\ \mathcal{C}_{2exp}[\![\mathit{2exp}_2]\!] \rceil \\
\mathcal{C}_{2exp}[\![\texttt{lift}\ \mathit{2exp}]\!] & = & \texttt{lift}\ \mathcal{C}_{2exp}[\![\mathit{2exp}]\!]
\end{array}
$$

$$\mathcal{C}_{2match} : \mathit{2match} \rightarrow \mathit{match}/\underline{\mathit{match}}$$

$$
\begin{array}{lcl}
\mathcal{C}_{2match}[\![\mathit{2mrule}\ \langle\ |\ \mathit{2match}\rangle]\!] & = & \mathcal{C}_{2mrule}[\![\mathit{2mrule}]\!]\ \langle\ |\ \mathcal{C}_{2match}[\![\mathit{2match}]\!]\rangle \\
\mathcal{C}_{2match}[\![\mathit{2mrule}\ \langle\ \underline{|}\ \mathit{2match}\rangle]\!] & = & \lceil \mathcal{C}_{2mrule}[\![\mathit{2mrule}]\!]\ \langle\ |\ \mathcal{C}_{2match}[\![\mathit{2match}]\!]\rangle \rceil
\end{array}
$$

$$\mathcal{C}_{2mrule} : \mathit{2mrule} \rightarrow \mathit{mrule}/\underline{\mathit{mrule}}$$

$$
\begin{array}{lcl}
\mathcal{C}_{2mrule}[\![\mathit{2pat}\ \texttt{=>}\ \mathit{2exp}]\!] & = & \mathcal{C}_{2pat}[\![\mathit{2pat}]\!]\ \texttt{=>}\ \mathcal{C}_{2exp}[\![\mathit{2exp}]\!] \\
\mathcal{C}_{2mrule}[\![\mathit{2pat}\ \underline{\texttt{=>}}\ \mathit{2exp}]\!] & = & \lceil \mathcal{C}_{2pat}[\![\mathit{2pat}]\!]\ \texttt{=>}\ \mathcal{C}_{2exp}[\![\mathit{2exp}]\!] \rceil
\end{array}
$$

$$\mathcal{C}_{2pat} : \mathit{2pat} \rightarrow \mathit{pat}/\underline{\mathit{pat}}$$

$$
\begin{array}{lcl}
\mathcal{C}_{2pat}[\![\mathit{var}]\!] & = & \mathit{var} \\
\mathcal{C}_{2pat}[\![\underline{\mathit{var}}]\!] & = & \lceil \mathit{var} \rceil \\
\mathcal{C}_{2pat}[\![i]\!] & = & i \\
\mathcal{C}_{2pat}[\![\underline{i}]\!] & = & \lceil i \rceil
\end{array}
$$

Figure 3: Program Generator Generator for Two-level Mini ML — Part 1

done-list in partial evaluation [21]. For efficiency reasons it is best to have one list for each sp-function rather than a global list; it saves the use of a tag which would otherwise be needed in order to distinguish between different sp-functions, and makes the lookup operations in the list faster.

Thereafter, the variable *var* is bound to a function which performs the actual specialization given the static argument. This function works as follows. First it checks, by looking up in the SeenB4List, whether we have already specialized the function with respect to the given static value. If so, the variable `seen` will be true and `name` will be bound to the name of the residual function corresponding to the value of the static argument, and this variable name is returned as an expression.

On the other hand, if `seen` is false, then `name` will be bound to a fresh variable name, which is the name of the new residual function to be generated and the SeenB4List will be updated. The utility function `emit` is then called with two

$\mathcal{C}_{2dec} : \mathit{2dec} \to \mathit{dec}$

$\mathcal{C}_{2dec}[\![\texttt{val rec}\ \mathit{2valbind}]\!] \quad = \quad \texttt{val rec}\ \mathcal{C}_{2valbind}[\![\mathit{2valbind}]\!]$

$\mathcal{C}_{2valbind} : \mathit{2valbind} \to \mathit{valbind}$

$\mathcal{C}_{2valbind}[\![\mathit{var} = \texttt{fn}\ \mathit{var}' \Rightarrow \underline{\texttt{fn}}\ \mathit{2match}]\!] \qquad =$

let $\mathit{sp\_seenB4list}$ = fresh_var() and x_i = dynvar($\mathit{2match}$) in

```
val sp_seenB4list = ref (nil: (int * string) list)
val var = fn var' =>
  let val (seen, name) = seenB4 sp_seenB4list var'
      val _ = if seen then ()  else emit name
         (let val x_i = fresh_var() in C_2match[[2match]] end)
  in ⌈name⌉ end
```

$\mathcal{C}_{2valbind}[\![\mathit{var} = \texttt{fn}\ \mathit{2match}]\!] \qquad =$

$\quad \mathit{var} = \texttt{fn}\ \mathcal{C}_{2match}[\![\mathit{2match}]\!]$

$\mathcal{C}_{2valbind}[\![\mathit{2valbind}_1\ \textsf{and}\ \mathit{2valbind}_2]\!] \qquad =$

$\quad \mathcal{C}_{2valbind}[\![\mathit{2valbind}_1]\!]\textsf{and}\ \mathcal{C}_{2valbind}[\![\mathit{2valbind}_2]\!]$

Figure 4: Program Generator Generator for Two-level Mini ML — Part 2

arguments: the name and the body of the residual function. `emit` glues the name and the body together into a residual function which it appends to a list of residual functions. The body of the residual function is the result of the inner `let`-expression. The dynamic variable x_i is bound to a fresh variable as usual in program specializers, see e.g. [14], to avoid name-clashes in the residual program. As seen in the definition of $\mathcal{C}_{2exp}$, we generate the code $\lceil \mathit{var} \rceil$ for dynamic variables bound in specialization functions. This means that the variable will be evaluated when the generating extension is run — it evaluates to the fresh variable name — and an expression consisting of that fresh variable is then made. Likewise for $\mathcal{C}_{2pat}$, except that a pattern and not an expression will be made here.

Observations. The following points about the presented `cogen` for Mini ML are worth noting, c.f. Section 2.2.

- It only deals with abstract syntax trees; in particular it does not perform environment manipulations. Clearly `cogen` never loops.
- The generated generating extension makes use of SML features (for example references) that are not present in Mini ML.
- It can easily be implemented in any conventional programming language, and it is not more complicated than a program specializer.

Notice in particular that `cogen` itself is not very complicated; the point is that the creativity mainly is in the binding-time analysis and the annotation phase and that this holds both for the indirect and the direct approach.

The observations scale well to larger languages, although it is a bit more complicated to deal with more sofisticated specialization techniques, such as partially static structures and arity raising [6].

3.3 Example

In this section we give an example of how the shown program generator generator can turn a Two-level Mini ML version of the Ackermann function into a generating extension written in SML, which when executed can specialize the Ackermann function. Suppose given the following Two-level Mini ML annotated version of the Ackermann function.

```
val rec ack = fn m => fn n =>
  (fn 0 => n + (lift 1)
   | m' => (sp_ack m') @ n) m
and sp_ack = fn m' =>
  (fn 0 => ack (m' - 1) (lift 1)
   | n' => ack (m' - 1) (ack m' (n' - (lift 1))))
```

Note that the first parameter, `m`, is static and the second is dynamic. When we apply the above shown program generator generator to this two-level program we get the following generating extension

```
val ack_SeenB4List = ref (nil: (int * string) list)
val rec ack_gen = fn (m: int) => fn (n: codetype) =>
  (fn 0 => ⌈n + lift 1⌉
   | m' => ⌈(sp_ack_gen m') n⌉) m
and sp_ack_gen = fn m' =>
  let
    val (seen,name) = seenB4 ack_SeenB4List m'
    val _ =
     if seen then () else emit name
      (let val n' = fresh_var()
       in   ⌈fn 0 => ack_gen (m' - 1) (lift 1)
              | n' => ack_gen (m' - 1) (ack_gen m' ⌈(n' - (lift 1)⌉))⌉
       end)
  in
    ⌈name⌉
  end
```

Assume that we want to specialize with respect to static value $\texttt{m} = \texttt{3}$. We then call the `ack_gen` function with actual parameters `3` and $\lceil\texttt{n}\rceil$. This causes the following residual function definitions to be generated.

```
fun f3 0  = f2 1
    | n17 = f2 (f3 (n17 - 1))
and f2 0  = f1 1
    | n18 = f1 (f2 (n18 - 1))
and f1 0  = 1 + 1
    | n19 = f1 (n19 - 1) + 1
```

To obtain the value of `(ack 3 8)` we can now evaluate `(f3 8)`.

4 A Program Generator Generator for SML

The principles presented above have been extended to cover most of the language features of the Standard ML Core Language [6]. We have made a prototype implementation based on The ML Kit [5], an implementation of SML written in SML. The implementation includes an efficient binding-time analysis, an annotation phase, and a program generator generator. The implemented program generator generator treats general user defined monomorphic data types, partially static structures and arity raising, and general pattern matching. Techniques have been developed for exceptions, but they have not been implemented. We defer discussion of these issues to a later paper, though we remark that the direct approach to program generator generators has proven very promising compared to the indirect approach, especially for nested pattern matching. In this section we report on some experiments with our prototype.

4.1 Interpreter for Imperative Language

A naïvely written interpreter for a small imperative language appears in [21, Figure 3.3]. We have used this interpreter to test our prototype. From the same source we also borrowed the following program

```
1: if x = y goto 7 else 2
2: if x < y goto 5 else 3
3: x := x - y
4: goto 1
5: y := y - x
6: goto 1
7: return x
```

which calculates the greatest common divisor of `x` and `y` using a variant of Euclid's algorithm. Applying `cogen` to the interpreter yields a compiler in the traditional sense and applying the compiler to the above program gives

```
fun f3 b = if b then 0 else 1
and f1 b = if b then 0 else 1;
fun f2 (x,y,true)  = x
  | f2 (x,y,false) = f4 (x,y,f3 (x<y) = 0)
and f4 (x,y,true)  = f2 (x,y-x,f1 (x=y-x) = 0)
  | f4 (x,y,false) = f2 (x-y,y,f1 (x-y=y) = 0);
fun main (x,y) = f2 (x,y,f1 (x=y) = 0);
```

except for trivial syntactic sugaring. At first sight this example is not as nice as one would usually expect from a partial evaluator, as some simple inlining and unfolding seemingly could improve the residual program. In existing partial evaluators a post-processing phase is often employed to do these simple optimizations, see, e.g., [9]. We have deliberately chosen not to include a post-processing phase in our prototype since such simple optimizations can just as well be done by the SML compiler used when the residual program is run (see [3] for a description of optimizations included in the most widely used SML compiler, SML/NJ).

Running the ML-code, using the values 1234567 and 7654321 is 125 times faster than running the interpreted program! Optimizations on the interpreter lower this factor to 85. The optimizations include removal of extremely naïve and inefficient lookup of the next command to be executed. The reason for this rather dramatic speedup is the effective removal of pattern matching, data structures, and environment lookup operations — a compiled version of the above residual program can execute almost entirely with values in registers, whereas the interpreter uses a lot of values which typically are either heap or stack allocated. When the interpreter is modified to use references (side effects) for both variables and statements, so that there is no explicit store representation, the interpreter is about five times faster than the original interpreter.

4.2 Ackermann's Function

Although Ackermann's function is completely useless for practical purposes it is traditionally used as an example. We have used a version like the annotated one in Section 3.3 for experiments. The actual generating extension is 134 lines long and was used to specialize Ackermann's function with respect to `m` = 3. The specialized program was as shown slightly sugared in Section 3.3. When computing the value of `(ack 3 8)` and then `(f3 8)` we observed a speed-up factor of 6.8.

5 Related Work

Earlier work on hand-writing `cogen` has been described in Section 2.2. Recently, Mark Leone and Peter Lee [24] have investigated an alternative and complement approach to compile-time program analysis and optimization, termed *deferred compilation.* They have written a code generator for stage-annotated terms, which permits them to consider very low-level optimizations including, e.g., register allocation. The idea of writing a code generator for stage-annotated terms is strongly related to our direct approach to program generator generators.

6 Conclusion and Future Work

In this paper we have argued that the direct approach to program generator generators has a number of advantages compared to the indirect approach. We have shown the basic principles of how to construct a program generator generator by presenting a program generator generator for a Two-level Mini ML language, and we have seen that it is just as easy or difficult to write `cogen` as it is to write `mix`. Moreover, we have reported on some promising experiments made with a prototype implementation of a program generator generator for most of the Standard ML Core Language.

Future work includes extending the prototype and making more practical experiments with it in order to obtain a better understanding of the implications of the design decisions for real Standard ML programs.

Another path of future work is to make the specialization phase stronger in the program generator generator for SML. It would be interesting to extend the prototype with continuation-based specialization [9]. Likewise, constructor specialization [29] might be considered, but the method suggested by Mogensen seems incompatible with the direct approach.

Acknowledgements

We wish to express our thanks for help with this article and with the work behind it to Peter Lee, Peter Sestoft, and Professor Neil D. Jones, and to the rest of the DIKU Topps Group for advice and helpful comments on this work.

References

1. Lars Ole Andersen. C program specialization. Technical Report 92/14, DIKU, Department of Computer Science, University of Copenhagen, May 1992.
2. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, Department fo Computer Science, University of Copenhagen, May 1994.
3. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
4. Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewall. A partial evaluator and its use as a programming tool. In *Artificial Intelligence 7*, pages 319–357. North-Holland Publishing Company, 1976.
5. Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit, Version 1. Technical Report 93/14, DIKU, University of Copenhagen, Denmark, 1993. The ML Kit is obtainable by anonymous ftp from `ftp.diku.dk` directory `pub/diku/users/birkedal`. This technical report is distributed along with the ML Kit.
6. Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Technical Report 93/22, DIKU, October 1993.
7. Anders Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, December 1990.
8. Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, March 1991.
9. Anders Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference in Lisp and Functional Programming, San Francisco, California. (Lisp Pointers, vol. V, no. 1, 1992)*, pages 1–10. ACM, 1992.
10. Anders Bondorf, Neil D. Jones, Torben Mogensen, and Peter Sestoft. Binding time analysis and the taming of self-application. Draft, 18 pages, DIKU, University of Copenhagen, Denmark, August 1988.
11. Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on partial evaluation*, 3, July 1993.
12. Anne de Niel. *Self-applicable Partial Evaluation of Polymorphically Typed Functional Languages.* PhD thesis, Katholieke Universiteit Leuven, January 1993.
13. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

14. Carsten K. Gomard. Higher order partial evaluation — HOPE for the lambda calculus. Master's thesis, DIKU, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, 1989.
15. Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.
16. Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. Springer-Verlag, 1991.
17. Carsten Kehler Holst. Syntactic currying. Student report, DIKU, 1989.
18. Carsten Kehler Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. Working Note, October 1992.
19. Neil D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Björner, A. P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.
20. Neil D. Jones. Efficient algebraic operations on programs. In *AMAST: Algebraic Methodology and Software Technology*, pages 245–267. University of Iowa, USA, 1991.
21. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Program Generation*. Prentice-Hall, 1993.
22. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
23. John Launchbury. A strongly-typed self-applicable partial evaluator. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science. ACM, Springer-Verlag, 1991.
24. Mark Leone and Peter Lee. Deferred compilation: The automation of run-time code generation. Technical Report CMU–CS–93–225, Carnegie Mellon University, December 1993.
25. Morten Marquard and Bjarne Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992.
26. U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 94–105. ACM, 1991.
27. Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.
28. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
29. Torben Æ. Mogensen. Constructor specialization. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–33, 1993.
30. Hanne Riis Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176, 1988.
31. D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 165–191. ACM, Springer-Verlag, 1991.