Constructor Specialization

Torben Æ. Mogensen
DIKU, University of Copenhagen, Denmark
email: torbenm@diku.dk

Abstract

In the section on "challenging problems" in the proceedings from the first international workshop on partial evaluation and mixed computation [BEJ88] a question is stated: "Can PE be used to generate new specialized data types, in a way analogous to generating specialized functions". Since then little has been done to address this problem. In [Lau89], new types are indeed generated, but they are all simpler versions of the types in the original program. It is, e.g. not possible to have types with more constructors than the types in the original program.

I propose to alleviate this by means of *constructor specialization*. Constructors are specialized with respect to the static parts of their arguments, just like residual functions. I show how this is done and argue that it makes it possible to get good results from partial evaluation in cases where the traditional methods fail to produce satisfactory results. The discussion is centered around a small subset of Standard ML, but the idea applies equally well to other languages having user defined constructors, e.g. Haskell and Prolog.

1 Motivation

It is usually considered bad practice if all residual programs for a given initial program share some syntactic limits with the original program, when these limits are not present in the language as a whole. This is especially true when partially evaluating an interpreter: any limits in the target programs means that the target language is not fully exploited. An example is if the number of function definitions in a residual program cannot be greater than the number of function definitions in the original program. This limitation is lifted when polyvariant specialization of functions are used ([Bul88], [JSS85], [JSS89]). The literature shows other examples of early partial evaluators having such inheritable limitations which later systems avoid, e.g. the number of arguments to functions being no greater in the residual functions than in the original functions, which is solved by variable splitting ([Ses86], [Mog88], [Rom90], [Lau89]).

Basically one can look at features that the programming language allows unlimited use of, but which any particular program uses only finitely. In addition to the above, languages with user definable constructors, usually only allow a finite number of such in any particular program. Partial evaluators of such languages (e.g. [Lau89]) usually allow only constructors from the original program to appear in the residual programs.

This can have the effect that data structures in the residual programs are represented suboptimally, as they will have to be forced into the mold of the data types from the original program. This is often the case in interpreters for typed languages: values of all types are represented using a single universal type, and this is carried over to the residual programs, where this ought no longer to be necessary.

2 Overview

I show a simple example of a program where normal specialization techniques yield unsatisfactory results, and show that constructor specialization handles this well. I then discuss how to modify binding time analysis and specialization algorithm to handle constructor specialization. I briefly touch upon possible refinements to these methods and some semantic issues that are raised by constructor specialization. I then revisit the example and show how a simple modification of the program improves the result obtained through constructor specialization. This is an instance of binding-time improvement [HH91]. Finally I conclude and discuss further work.

3 An example

Consider the Standard ML program shown in figure 1. It implements a simple parser for a restricted class of grammars using only one nonterminal. `EMPTY` corresponds to the empty string, `REC` is a recursive reference to the entire grammar (and is thus the only nonterminal). `SYMB` indicates a terminal symbol and `SEQ` and `ALT` indicates sequence and alternative. The function `parse` takes a grammar and a list of terminal symbols and determines if the list is in the language generated by the grammar. It uses a stack of pieces of the grammar, which will be used to match against the remainder of the terminal list. The data type `ToDo` implements this stack. The parser backtracks on alternatives.

Let us consider specialization of the parser with respect to a specific grammar. A simple binding time analysis will tell us that the grammar and the stack are static throughout, so we expect all to be well when specializing. Let's try the grammar $G \rightarrow (G)G \mid \epsilon$, which is represented in the data type by

```
ALT(SEQ(SYMB("("),SEQ(REC,SEQ(SYMB(")"),REC))),
    EMPTY)
```

Figure 1: Backtracking parser

```
datatype Gram = EMPTY
              | REC
              | SYMB of string
              | SEQ of Gram*Gram
              | ALT of Gram*Gram

datatype ToDo = DONE | DO of Gram*ToDo

fun parse(g,s) = p(g,s,g,DONE)

fun p(g,s,g0,todo) =
      case g of
        EMPTY       => nowdo(todo,s,g0)
      | REC         => p(g0,s,g0,todo)
      | SYMB c      => s<>[] andalso (hd s)=c andalso nowdo(todo,tl s,g0)
      | SEQ (g1,g2) => p(g1,s,g0,DO(g2,todo))
      | ALT (g1,g2) => p(g1,s,g0,todo) orelse p(g2,s,g0,todo)
end;

fun nowdo(todo,s,g) =
      case todo of
        DONE          => s=[]
      | DO (g1,todo') => p(g1,s,g,todo')
end;
```

To our (maybe not so great) surprise the specialization doesn't terminate. The reason is that the stack grows unboundedly during specialization. It is clear that, in general, the size of the stack depends on the input string, so it is not the fault of the partial evaluator *per se*. The usual solution to handle unbounded data structures is to generalize them, either during specialization or by modifying the binding times. This, however, has the unfortunate effect that, since the grammar is stored on the stack, it too becomes dynamic. The result is that no speedup can be obtained by partial evaluation. It is not obvious that the program can be rewritten to avoid this problem (unless one uses continuations like in [Mos93], but that is a major rewrite, and not an option if the language does not support higher order functions).

We take a closer look at why the generalizing the stack causes the grammar to be dynamic. When handling the sequence operator, the parser pushes a part of the grammar on the stack, using the constructor DO. Since the stack is generalized, DO is a dynamic constructor and the normal rules for binding times makes the arguments to it dynamic too. Thus, when we pop the grammar from the stack in nowdo, it is dynamic.

But what if we allow dynamic constructors to have static arguments? Then we can pop a static grammar from the stack, even when this is dynamic. "How," I hear you say "are we supposed to do this?". When we use DO to push the grammar on the stack, we must generate code in the residual program that does this (the stack is dynamic, remember), and similarly we must generate residual code to pop the stack in the residual versions of nowdo. Instead of pushing the actual grammar, we can *specialize* DO to the grammar and use the specialized constructor to do the pushing. When popping, we do a case over the specialized versions of DO and in each branch we can use the static grammar that the corresponding constructor is specialized with.

If we use the example grammar from above, we will during execution push the following bits of the grammar:

```
SEQ(REC,SEQ(SYMB(")"),REC))

SEQ(SYMB(")"),REC)

REC
```

We can thus make three specialized versions of DO, corresponding to each of these possibilities. Let us call these DO_0, DO_1 and DO_2. When doing a case on these (in nowdo), we can in the branch corresponding to DO_0 use the fact that the grammar (g1) is SEQ(REC,SEQ(SYMB(")"),REC)) etc. Thus the grammar will remain static in all calls to p, and we obtain the residual program shown in figure 2.

We have unfolded all calls to p, except the first and the one in the branch of the REC operator. The calls to nowdo are all left residual. It can be seen that all mention of the grammar has been evaluated away during specialization. It is clear that the residual parser is much more efficient than the original. However, there is still room for improvement: there are two calls to nowdo' with constructors in the arguments. These will immediately be tested upon. Unfolding these calls can remove some dynamic pushing and popping. Note that we still need to push DO_1 and test for this against DONE. We will later see how we can modify the parser slightly so the superfluous operations disappear from the residual parser.

4 A small language

First we describe a small subset of Standard ML [MTH90], which will be used in the sections on binding time analysis and specialization.

The syntax of the language is shown in figure 3. Note that the data type declarations are monomorphic. Polymorphic data types cause some extra bookkeeping during binding time analysis and partial evaluation, so for ease of presentation, these have been left out. For similar reasons

Figure 2: Residual parser using constructor specialization

```
datatype ToDo' = DONE | DO_0 of ToDo' | DO_1 of ToDo' | DO_2 of ToDo'

fun parse'(s) = p'(s,DONE)

fun p'(s,todo) =
      (s<>[] andalso (hd s)="(" andalso nowdo'(DO_0 todo,tl s))
      orelse nowdo'(todo,s)
end;

fun nowdo'(todo,s) =
      case todo of
        DONE       => s=[]
      | DO_0 todo' => p'(s,DO_1 todo')
      | DO_1 todo' => s<>[] andalso (hd s)=")" andalso nowdo'(DO_2 todo',tl s)
      | DO_2 todo' => p'(s,todo')
end;
```

Figure 3: Syntax of SML subset

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Datatype}^{*}\ \mathit{Function}^{*} \\[6pt]
\mathit{Datatype} & \rightarrow & \texttt{datatype}\ \mathit{Typeid} = \mathit{Contype}^{+}_{|} \\[6pt]
\mathit{Contype} & \rightarrow & \mathit{Conid}\ \texttt{of}\ \mathit{Typeid}^{*}_{*} \\[6pt]
\mathit{Function} & \rightarrow & \texttt{fun}\ \mathit{Funid}\ \mathit{Pattern} = \mathit{Exp}\ \texttt{end;} \\[6pt]
\mathit{Pattern} & \rightarrow & (\mathit{Varid}^{*}_{,}) \\[6pt]
\mathit{Exp} & \rightarrow & \mathit{Varid} \\
 & | & \mathit{Funid}\ \mathit{Exp} \\
 & | & \mathit{Basic}\ \mathit{Exp} \\
 & | & \mathit{ConidExp} \\
 & | & (\mathit{Exp}^{*}_{,}) \\
 & | & \texttt{case}\ \mathit{Exp}\ \texttt{of}\ \mathit{Branch}^{+}_{|} \\[6pt]
\mathit{Branch} & \rightarrow & \mathit{Conid}\ \mathit{Patt} => \mathit{Exp}
\end{array}
$$

the language is restricted to first order functions and no local definitions. Infix operators are assumed to be syntactic sugar for calls to basic functions, and so do not occur in the grammar. Similarly nullary constructors and functions are considered sugar for functions taking an argument of unit type. Constants are nullary basic functions. The notation $Nonterminal^{*}_{symbol}$ is shorthand for a (possibly empty) sequence of $Nonterminal$ separated by *symbol*. Similarly $Nonterminal^{+}_{symbol}$ stands for a non-empty sequence. We will later use $Nonterminal^{?}$ to stand for zero or one occurrence of $Nonterminal$.

5 Binding time analysis

A common way to describe binding times for functional languages is to use projections [Lau89], [Mog89], but there is no projection that naturally describes a data type where the constructors are unknown and their arguments known. So instead we will us a type-based approach like in [GJ91], [Mog92] and [Hen91]. For simplicity of presentation we'll use a simple monomorphic system, but the idea scales in the usual way to more complex systems.

The binding times are described as a set of type rules. Binding time analysis consists of type inference using these rules. Figure 4 shows the syntax of annotated programs. Underlining is used to describe dynamic parts of the program. It is assumed that the program initially contains no underlines, but that a minimal consistent underlining will be provided during a type inference based binding time analysis. Details can be found in [Hen91].

Three environments are used: ρ for variables, ϕ for functions and κ for constructors. All environments map their objects to binding time annotated types. $\oplus$ is used to join environments. In case of common names in ρ and ρ', ρ' takes precedence in $\rho \oplus \rho'$. An annotated type has the form

$$
\begin{array}{lcl}
\mathit{BTtype} & \rightarrow & \mathit{Typeid} \\
 & | & \underline{\mathit{Typeid}} \\
 & | & \mathit{BTtype}^{*}_{*} \\
 & | & \mathit{BTtype} \rightarrow \mathit{BTtype}
\end{array}
$$

where underlines denote dynamic types. For convenience of notation we will use the equivalences that the empty product type is the unit type () and that $\underline{BT_1 * \ldots * BT_n} = \underline{BT_1} * \ldots * \underline{BT_n}$. We will thus use $\underline{BT}$ for a completely dynamic type.

Note that since the language is first order, there is no such thing as a dynamic function. A constructor is considered a function from its argument types to its data type. If a constructor is dynamic, its data type is and *vice versa*, so there is no need to annotate the function space arrow.

We first present rules for a "traditional" binding type analysis and then modify these to handle constructor specialization. The "traditional" type rules are given in figure 5. The types for the different logical connectives are shown with the rules that have them in their conclusion.

Obviously, it is the rules involving dynamic constructors that must be modified to accommodate constructor specialization. The rule for constructor declaration states that the argument type of a constructor in a dynamic data type must be dynamic. We remove this side condition. Similarly the rule for dynamic constructor application says that the argument must be dynamic. We change this so it (like in dynamic function applications) can be of any binding time. There is no need to change the rule for `case`-expressions, as

Figure 4: Annotated syntax

$$\begin{array}{lll}
\mathit{AProgram} & \rightarrow & \mathit{ADatatype}^*\ \mathit{AFunction}^* \\
 & & \\
\mathit{ADatatype} & \rightarrow & \texttt{datatype}\ \mathit{Typeid} = \mathit{AContype}^{+}_{|} \\
 & | & \underline{\texttt{datatype}}\ \mathit{Typeid} = \mathit{AContype}^{+}_{|} \\
 & & \\
\mathit{AContype} & \rightarrow & \mathit{Conid}\ \texttt{of}\ \mathit{ATypeid}^{*}_{*} \\
 & & \\
\mathit{AFunction} & \rightarrow & \texttt{fun}\ \mathit{Funid}\ \mathit{APattern} = \mathit{AExp}\ \texttt{end;} \\
 & & \\
\mathit{APattern} & \rightarrow & (\mathit{AVarid}^{*}_{,}) \\
 & & \\
\mathit{AExp} & \rightarrow & \mathit{Varid} \\
 & | & \mathit{Funid}\ \ \mathit{AExp} \\
 & | & \mathit{Funid\_AExp} \\
 & | & \mathit{Basic}\ \ \mathit{AExp} \\
 & | & \mathit{Basic\_AExp} \\
 & | & \mathit{Conid}\ \ \mathit{AExp} \\
 & | & \mathit{Conid\_AExp} \\
 & | & (\mathit{AExp}^{*}_{,}) \\
 & | & \texttt{case}\ \mathit{AExp}\ \texttt{of}\ \mathit{ABranch}^{+}_{|} \\
 & | & \underline{\texttt{case}}\ \mathit{AExp}\ \underline{\texttt{of}}\ \mathit{ABranch}^{+}_{|} \\
 & & \\
\mathit{ABranch} & \rightarrow & \mathit{Conid}\ \mathit{APatt} \Rightarrow \mathit{AExp} \\
 & & \\
\mathit{ATypeid} & \rightarrow & \mathit{Typeid}\ |\ \underline{\mathit{Typeid}} \\
 & & \\
\mathit{AVarid} & \rightarrow & \mathit{Varid}\ |\ \underline{\mathit{Varid}}
\end{array}$$

it imposes no restrictions on the arguments of constructors. One can note, though, that it is now possible to have static pattern variables in dynamic `case`-expressions. The modified rules are shown in figure 6. The binding time analysis is thus no more complex than before the modifications, if anything it is simpler. If we had used an abstract interpretation approach, we would have had to modify our binding time domain. So in this respect the type/constraint based approach is more flexible.

6 Specialization

We use inference rules to specify a relation between the original program and the specialized program. Again, this leaves some room for different implementations. The rules are shown in figures 7 and 8.

The binding time environments κ and ϕ are used in the rules for specialization. In addition to these, χ maps constructors to the sets of the static arguments to which they are specialized and ψ maps function names to sets of static arguments. The environment ρ binds variables to partially static values. The static parts of these are values, and the dynamic parts are residual expressions:

$$\begin{array}{lll}
PVal & \rightarrow & Value \\
 & | & Exp \\
 & | & (PVal^{*}_{,}) \\
 & | & Conid(Pval)
\end{array}$$

We are a bit sloppy in that we use $(\cdot, \ldots, \cdot)$ both for tupling of values and for constructing tuple-expressions.

Specialized functions and constructors are identified by pairs of their original names and the static parts of their arguments. These pairs must be replaced by new names in the usual way.

ψ_0 contains the goal function and the arguments to which this must be specialized. The environment ψ ensures that all specialized functions that are called are also defined, and χ ensures that if a constructor is specialized, then all it's specialized versions are declared and occur in every residual `case`-expression where the original constructor appears in the corresponding original `case`-expression. This is handled in the rules for Function, Contype and Dynamic branch. The rules for Pattern and Varid are used to build specialization time environments and residual patterns. The function *new* creates new unique variable names. The rules for Exp specializes expressions. Static expressions produce values and dynamic expressions produce residual expressions. Partially static expressions produce a mixture of these, which must be taken apart (using $\nearrow$) when needed as arguments to residual function calls or dynamic constructor applications (note the similarity between the rules for these).

The most interesting case is the dynamic `case`-expression. Each branch in the original `case`-expression is made into several branches in the residual `case`-expression. These branches are produced by the rule for Dynamic branch; the possible static arguments are fetched from χ and each value is used to make a residual branch: the constructor name is specialized, the pattern is specialized, a new environment is built, in which the body of the branch is specialized. If constructor specialization is not used, only a single residual branch would be produced, and all pattern variables would be dynamic. Note the similarity between the rule for Dynamic branch and for Function. The main difference is that the rule for Dynamic branch is used in every dynamic `case`-expression, whereas the rule for Function is applied only once for each function.

Retyping similar to what is achieved through projection factorisation in [Lau89] is done by the rules for Type: the residual type depends on the static value. Note how these rules parallel the rules for splitting a partially static value into its static and dynamic components and for variable splitting. Static base types are replaced by the unit type and the corresponding dynamic expressions are replaced by the unit constant (). The $*$ tuple type constructor used in the rules for constructing residual types is assumed to be associative and has the unit type as unit element. New parameter lists (patterns) to functions and constructors are constructed, such that each dynamic component of a partially static value becomes a separate parameter. For formal parameters this is handled by the rules for Pattern and Variable splitting. For actual parameters the rules for splitting values into static and dynamic components are used. The operator $+\!\!+$ is used to join tuple expressions and patterns. It obeys the rules below. The rules are prioritized from top to bottom.

$$\begin{array}{lll}
() +\!\!+ x & = & x \\
x +\!\!+ () & = & x \\
x +\!\!+ y & = & (x, y) \\
(x_1, \ldots, x_n) +\!\!+ (y_1, \ldots, y_n) & = & (x_1, \ldots, x_n, y_1, \ldots y_n) \\
x +\!\!+ (y_1, \ldots, y_n) & = & (x, y_1, \ldots, y_n) \\
(x_1, \ldots, x_n) +\!\!+ y & = & (x_1, \ldots, x_n, y)
\end{array}$$

The operation $new(V)$ used in the rules for Variable splitting creates a new unique name from the name V.

Figure 5: Type rules for binding time analysis

Program: $\vdash \; AProgram \;\propto\; (Conid \rightarrow BTtype)/(Funid \rightarrow BTtype)$

$$\frac{\vdash D_1 \;\rhd \kappa_1 \;\ldots\; \vdash D_1 \;\rhd \kappa_n \quad \kappa = \kappa_1 \oplus \ldots \oplus \kappa_n \quad \kappa, \phi \vdash Fn_1 \succ \phi_1 \;\ldots\; \kappa, \phi \vdash Fn_m \succ \phi_m \quad \phi = \phi_1 \oplus \ldots \oplus \phi_m}{\vdash D_1 \ldots D_n \; Fn_1 \ldots Fn_m \propto \kappa/\phi}$$

Datatype: $\vdash \; ADatatype \;\;\rhd\;\; (Conid \rightarrow BTtype)$

$$\frac{}{\vdash \texttt{datatype}\; Ti = C_1 \;\texttt{of}\; BT_1 \mid \ldots \mid C_n \;\texttt{of}\; BT_n \;\;\rhd\;\; [C_1 \mapsto BT_1 \rightarrow Ti \;, \ldots ,\; C_n \mapsto BT_n \rightarrow Ti]}$$

$$\frac{}{\vdash \underline{\texttt{datatype}}\; Ti = C_1 \;\underline{\texttt{of}}\; \underline{BT_1} \mid \ldots \mid C_n \;\texttt{of}\; \underline{BT_n} \;\;\rhd\;\; [C_1 \mapsto \underline{BT_1} \rightarrow \underline{Ti} \;, \ldots ,\; C_n \mapsto \underline{BT_n} \rightarrow \underline{Ti}]}$$

Function: $(Conid \rightarrow BTtype), (Funid \rightarrow BTtype) \;\vdash\; AFunction \;\succ\; (Funid \rightarrow BTtype)$

$$\frac{\vdash P \rightsquigarrow \rho/BT_1 \quad \kappa, \phi, \rho \vdash E : BT_2}{\kappa, \phi \vdash \texttt{fun}\; F \; P = E \;\texttt{end;} \;\succ\; [F : BT_1 \rightarrow BT_2]}$$

Pattern: $\vdash \; APattern \;\rightsquigarrow\; (Varid \rightarrow BTtype)/BTtype$

$$\frac{}{\vdash (V_1, \ldots, V_n) \;\rightsquigarrow\; [V_1 \mapsto BT_1, \ldots V_n \mapsto BT_n]/BT_1 * \ldots * BT_n}$$

Exp: $(Conid \rightarrow BTtype), (Funid \rightarrow BTtype), (Varid \rightarrow BTtype) \;\vdash\; AExp \;:\; BTtype$

$$\frac{\rho V = BT}{\kappa, \phi, \rho \vdash V \;:\; BT}$$

$$\frac{\phi F = BT_1 \rightarrow BT_2 \quad \kappa, \phi, \rho \vdash E \;:\; BT_1}{\kappa, \phi, \rho \vdash F \; E \;:\; BT_2} \qquad \frac{\phi F = BT_1 \rightarrow \underline{BT_2} \quad \kappa, \phi, \rho \vdash E \;:\; BT_1}{\kappa, \phi, \rho \vdash F\_E \;:\; \underline{BT_2}}$$

$$\frac{B \;:\; T_1 \rightarrow T_2 \quad \kappa, \phi, \rho \vdash E \;:\; T_1}{\kappa, \phi, \rho \vdash B \; E \;:\; T_2} \qquad \frac{B \;:\; T_1 \rightarrow T_2 \quad \kappa, \phi, \rho \vdash E \;:\; \underline{T_1}}{\kappa, \phi, \rho \vdash B\_E \;:\; \underline{T_2}}$$

$$\frac{\kappa C = BT_1 \rightarrow Ti \quad \kappa, \phi, \rho \vdash E \;:\; BT_1}{\kappa, \phi, \rho \vdash C \; E \;:\; Ti} \qquad \frac{\kappa C = \underline{BT_1} \rightarrow \underline{Ti} \quad \kappa, \phi, \rho \vdash E \;:\; \underline{BT_1}}{\kappa, \phi, \rho \vdash C\_E \;:\; \underline{Ti}}$$

$$\frac{\kappa, \phi, \rho \vdash E_1 \;:\; BT_1 \quad \ldots \quad \kappa, \phi, \rho \vdash E_n \;:\; BT_n}{\kappa, \phi, \rho \vdash (E_1, \ldots, E_n) \;:\; (BT_1, \ldots, BT_n)}$$

$$\frac{\kappa, \phi, \rho \vdash E \;:\; Ti \quad \kappa, \phi, \rho, Ti \vdash B_1 \;:>\; BT_2 \quad \ldots \quad \kappa, \phi, \rho, Ti \vdash B_n \;:>\; BT_2}{\kappa, \phi, \rho \vdash \texttt{case}\; E \;\texttt{of}\; B_1 | \ldots | B_n \;:\; BT_2}$$

$$\frac{\kappa, \phi, \rho \vdash E \;:\; \underline{Ti} \quad \kappa, \phi, \rho, \underline{Ti} \vdash B_1 \;:>\; \underline{BT_2} \quad \ldots \quad \kappa, \phi, \rho, \underline{Ti} \vdash B_n \;:>\; \underline{BT_2}}{\kappa, \phi, \rho \vdash \underline{\texttt{case}}\; E \;\underline{\texttt{of}}\; B_1 | \ldots | B_n \;:\; \underline{BT_2}}$$

Branch: $(Conid \rightarrow BTtype), (Funid \rightarrow BTtype), (Varid \rightarrow BTtype), BTtype \;\vdash\; ABranch \;:>\; BTtype$

$$\frac{\kappa C = BT_1 \rightarrow Ti \quad \vdash P \rightsquigarrow \rho'/BT_1 \quad \kappa, \phi, \rho \oplus \rho' \vdash E \;:\; BT_2}{\kappa, \phi, \rho, Ti \vdash C \; P => E \;:>\; BT_2}$$

Figure 6: Modified type rules for binding time analysis

Datatype:

$$\frac{}{\vdash \underline{\texttt{datatype}}\ Ti = C_1\ \underline{\texttt{of}}\ BT_1 \mid \ldots \mid C_n\ \texttt{of}\ BT_n \ \ \rhd\ [C_1 \mapsto BT_1 \to \underline{Ti}\ , \ldots,\ C_n \mapsto BT_n \to \underline{Ti}]}$$

Exp:

$$\frac{\kappa C = BT_1 \to \underline{Ti} \quad \kappa, \phi, \rho \vdash E\ :\ BT_1}{\kappa, \phi, \rho \vdash C\_E\ :\ \underline{Ti}}$$

It is easy in an implementation of the specialization rules to maintain ψ: entries are added whenever residual function calls are made, and residual functions are generated for each entry until all entries have definitions. New entries can never cause earlier generated residual function definitions to be invalid, so the new definitions are simply added to the existing definitions. Things are not as simple with χ. A new specialized constructor requires all residual `case`-expressions of relevant type to test on this, even those `case`-expressions that are already generated. Thus either regeneration of old function definitions in a standard fixed-point iteration style or an ability to "backpatch" earlier generated `case`-expressions is needed. The first solution is simple, but inefficient. The second solution is potentially more efficient, as each piece of residual code is only generated once, but requires either destructive updating of the residual program, or a multi-pass algorithm that first generates "pieces" of code for the `case`-expression branches and then assembles these in the final residual program.

7 Refinements and semantic considerations

There is a lot of room for improvement of both binding time analysis and specialization as they are shown here. First of all, polyvariant binding time analysis ([GR92], [DNBDV91]) could be used. Specialized constructors add no fundamental extra complexities to this problem, except that projections, as used in [Lau89] and [DNBDV91] appear ill-suited for describing binding times involving specialized constructors.

Another problem (which is unrelated to polyvariant binding time analysis) is the fact that types are specialized monovariantly in the specialization definition presented in this paper. Thus the residual program can have no more data type declarations than the original program, something that is against our "guiding rule" of not letting limits be inherited. It also has some potential semantic problems: when specializing a constructor it is automatically assumed that it will be used in any residual version of `case`-expressions that use the original constructor. Consider an interpreter where a universal data type is used to represent arbitrary types. Consider this interpreter specialized with respect to a program that uses two types. The two types will in the residual program be represented by a single type, and the functions that operate on one of the types will also have to test on the constructors from the other type. This can in the worst case cause run-time errors during specialization, and at best there will be lots of "dead code" in the residual program. The solution to this is to perform polyvariant type specialization: the specialized constructors are not collected into a single type, but grouped corresponding to where they are generated and used in the residual program. It is not clear how best to do this, but one could assume that newly generated specialized constructors are in a type of their own, and only combine the types when necessary, *e.g.* when they occur as the result of different branches to the same `case`-expression. A union/find style algorithm would be suitable for this approach.

The rules for specialization say nothing about what happens if specialized constructors occur in the output of the program. The most obvious solution is to disallow this by adding constrains that enforce all parts of output types to be dynamic. Another solution would be to allow specialized constructors in the output, but letting the specializer in addition to the residual program also return a specification of how the specialized constructors correspond to original terms. This can be seen as a derivor mapping the specialized data types to the original data types. Such a derivor can also be allowed as part of the input to the specializer, allowing a program to be specialized with respect to the specialized data type of another program. This can be used to specialize the interface types of multi-pass algorithms. This is an similar what Ershov calls *mixed computation* [BEJ88]: both residual program and residual data are produced. In our case, though, it is not actual data, but a data specification that is produced.

8 The example revisited

In section 3 I promised to show a modification of the parser program from figure 1, which would allow better specialization. The problem was that, in the residual program, `p'` would call `nowdo'` with a constructor which immediately would be decomposed inside `nowdo'`. The problem occurs because all applications of the constructor `DO` are made residual. The reason we made them residual was to avoid infinite growth in the stack, but this only occurs because we have recursion in the grammer. If we can generalize the stack only when recursion occurs, we can make more of the stack static. A way to do this is to leave the type `ToDo` static, but encapsulate the stack in a dynamic constructor of another type whenever recursion is needed. The modified parser is shown in figure 9.
By making `ToDo1` dynamic, we can avoid infinite growth of the stack. Using the same grammar as in section 3, we will apply `GENERALIZE` to the following arguments

```
DONE

DO(SEQ(SYMB(")"),REC),GENERALIZED(<ToDo1>))
```

Note that the function `generalize` ensures that `GENERALIZE` will not be applied to an already generalized argument. This

Figure 7: Rules for specialization (Part 1)

Program: $(Conid \rightarrow BTtype), (Funid \rightarrow BTtype), (Funid \rightarrow \{Pval\}) \vdash AProgram \mathrel{\overline{\propto}} Program$

$$\frac{\kappa, \chi \vdash D_i \mathrel{\overline{\triangleright}} Ds_i \quad \text{for } i = 1, \ldots, n \quad \kappa, \chi, \phi, \psi \vdash Fn_j \mathrel{\overline{\succ}} Fns_j \quad \text{for } j = 1, \ldots, m \quad \psi_0 \subseteq \psi}{\kappa, \phi, \psi_0 \vdash D_1 \ldots D_n \ Fn_1 \ldots Fn_m \mathrel{\overline{\propto}} Ds_1 \ldots Ds_n \ Fns_1 \ldots Fns_m}$$

Datatype: $(Conid \rightarrow BTtype), (Conid \rightarrow \{Pval\}) \vdash ADatatype \mathrel{\overline{\triangleright}} Datatype^{?}$

$$\frac{}{\kappa, \chi \vdash \texttt{datatype}\ Ti = Cs \mathrel{\overline{\triangleright}} \epsilon}$$

$$\frac{\kappa, \chi \vdash C_1 \mathrel{\bar{\triangleright}} Cs_1 \quad \ldots \quad \kappa, \chi \vdash C_n \mathrel{\bar{\triangleright}} Cs_n}{\kappa, \chi \vdash \underline{\texttt{datatype}}\ Ti = C_1 | \ldots | C_n \mathrel{\overline{\triangleright}} \texttt{datatype}\ Ti = Cs_1 | \ldots | Cs_n}$$

Contype: $(Conid \rightarrow BTtype), (Conid \rightarrow \{Pval\}) \vdash AContype \mathrel{\bar{\triangleright}} Contype^{+}_{|}$

$$\frac{\chi C = \{v_1, \ldots, v_n\} \quad \kappa, BT \vdash v_1 \searrow T_1 \quad \ldots \quad \kappa, BT \vdash v_n \searrow T_n}{\kappa, \chi \vdash C \ \texttt{of}\ BT \mathrel{\bar{\triangleright}} (C, v_1)\ \texttt{of}\ T_1 \mid \cdots \mid (C, v_n)\ \texttt{of}\ T_n}$$

Type: $(Conid \rightarrow BTtype), BTtype \vdash Pval \searrow Type$

$$\frac{}{\kappa, \underline{T} \vdash v \searrow (T)}$$

$$\frac{}{\kappa, Ti \vdash v \searrow ()} \quad \text{if } Ti \text{ is a base type}$$

$$\frac{\kappa C = BT \rightarrow Ti \quad \kappa, BT \vdash v \searrow T}{\kappa, Ti \vdash C(v) \searrow T} \quad \text{if } Ti \text{ is a data type}$$

$$\frac{\kappa, BT_1 \vdash v_1 \searrow T_1 \quad \ldots \quad \kappa, BT_n \vdash v_n \searrow T_n}{\kappa, BT_1 * \ldots * BT_n \vdash (v_1, \ldots, v_n) \searrow T_1 * \ldots * T_n}$$

Function: $(Conid \rightarrow BTtype), (Conid \rightarrow \{Pval\}), (Funid \rightarrow BTtype), (Funid \rightarrow \{Pval\}) \vdash AFunction \mathrel{\overline{\succ}} Function^{*}$

$$\frac{\phi F = BT_1 \rightarrow BT_2 \quad \psi F = \{v_1, \ldots, v_n\} \quad \kappa, BT_1, P \vdash v_i \mathrel{\overline{\rightsquigarrow}} \rho_i / P_i \quad \kappa, \chi, \phi, \psi, \rho_i \vdash E \mathrel{\bar{\vdots}} E_i \quad \text{for } i = 1, \ldots, n}{\kappa, \chi, \phi, \psi \vdash \texttt{fun}\ F\ P = E\ \texttt{end;} \mathrel{\overline{\succ}} \texttt{fun}\ (F, v_1)\ P_1 = E_1\ \texttt{end;} \ldots \texttt{fun}\ (F, v_n)\ P_n = E_n\ \texttt{end;}}$$

Pattern: $(Conid \rightarrow BTtype), BTtype, APattern \vdash (PVal^{*}_{,}) \mathrel{\overline{\rightsquigarrow}} (Varid \rightarrow Pval)/Pattern$

$$\frac{\kappa, BT_1, V_1 \vdash v_1 \mathrel{\overline{\hookrightarrow}} v_1' / P_1 \quad \ldots \quad \kappa, BT_n, V_n \vdash v_n \mathrel{\overline{\hookrightarrow}} v_n' / P_n}{\kappa, BT_1 * \ldots * BT_n, (V_1, \ldots, V_n) \vdash (v_1, \ldots, v_n) \mathrel{\overline{\rightsquigarrow}} [V_1 \mapsto v_1', \ldots, V_n \mapsto v_n'] / P_1 \mathbin{++} \ldots \mathbin{++} P_n}$$

Variable splitting: $(Conid \rightarrow BTtype), BTtype, Varid \vdash (PVal^{*}_{,}) \mathrel{\overline{\hookrightarrow}} (Varid \rightarrow Pval)/Pattern$

$$\frac{V' = new(V)}{\kappa, \underline{T}, V \vdash () \mathrel{\overline{\hookrightarrow}} V'/V'}$$

$$\frac{}{\kappa, Ti, V \vdash v \mathrel{\overline{\hookrightarrow}} v/()} \quad \text{if } Ti \text{ is a base type}$$

$$\frac{\kappa C = BT \rightarrow Ti \quad \kappa, BT, V \mathrel{\overline{\hookrightarrow}} v'/P}{\kappa, Ti, V \vdash C(v) \mathrel{\overline{\hookrightarrow}} C(v')/P} \quad \text{if } Ti \text{ is a data type}$$

$$\frac{\kappa, BT_1, V \mathrel{\overline{\hookrightarrow}} v_1'/P_1 \quad \ldots \quad \kappa, BT_n, V \mathrel{\overline{\hookrightarrow}} v_n'/P_n}{\kappa, BT_1 * \ldots * BT_n, V \vdash (v_1, \ldots, v_n) \mathrel{\overline{\hookrightarrow}} (v_1', \ldots v_n')/P_1 \mathbin{++} \ldots \mathbin{++} P_n}$$

Figure 8: Rules for specialization (Part 2)

Exp: $(Conid \rightarrow BTtype), (Conid \rightarrow \{Pval\}), (Funid \rightarrow BTtype), (Funid \rightarrow \{Pval\}), (Varid \rightarrow Pval) \vdash AExp \mathrel{\overline{:}} PVal$

$$\frac{\rho V = v}{\kappa, \chi, \phi, \psi, \rho \vdash V \mathrel{\overline{:}} v}$$

$$\frac{\kappa, \chi, \phi, \psi, \rho \vdash E \mathrel{\overline{:}} v \quad v \vdash P \mathrel{\overline{\rightsquigarrow}} \rho'/P' \quad \kappa, \chi, \phi, \psi, \rho' \vdash E' \mathrel{\overline{:}} w}{\kappa, \chi, \phi, \psi, \rho \vdash F\ E \mathrel{\overline{:}} w} \quad \text{where } F \text{ is defined by } \texttt{fun}\ F\ P = E'\ \texttt{end};$$

$$\frac{\phi F = BT_1 \rightarrow \underline{BT_2} \quad \kappa, \chi, \phi, \psi, \rho \vdash E \mathrel{\overline{:}} v \quad \kappa, BT_1 \vdash v \nearrow v'/E' \quad v' \in \psi F}{\kappa, \chi, \phi, \psi, \rho \vdash F\_E \mathrel{\overline{:}} (F, v')\ E'}$$

$$\frac{\kappa, \chi, \phi, \psi, \rho \vdash E \mathrel{\overline{:}} v}{\kappa, \chi, \phi, \psi, \rho \vdash B\ E \mathrel{\overline{:}} b(v)} \qquad \frac{\kappa, \chi, \phi, \psi, \rho \vdash E \mathrel{\overline{:}} E'}{\kappa, \chi, \phi, \psi, \rho \vdash B\_E \mathrel{\overline{:}} B\ E'}$$

$$\frac{\kappa, \chi, \phi, \psi, \rho \vdash E \mathrel{\overline{:}} v}{\kappa, \chi, \phi, \psi, \rho \vdash C\ E \mathrel{\overline{:}} C(v)}$$

$$\frac{\kappa C = BT_1 \rightarrow \underline{Ti} \quad \kappa, \chi, \phi, \psi, \rho \vdash E \mathrel{\overline{:}} v \quad \kappa, BT_1 \vdash v \nearrow v'/E' \quad v' \in \chi C}{\kappa, \chi, \phi, \psi, \rho \vdash C\_E \mathrel{\overline{:}} (C, v')\ E'}$$

$$\frac{\kappa, \chi, \phi, \psi, \rho \vdash E_1 \mathrel{\overline{:}} v_1 \quad \ldots \quad \kappa, \chi, \phi, \psi, \rho \vdash E_n \mathrel{\overline{:}} v_n}{\kappa, \chi, \phi, \psi, \rho \vdash (E_1, \ldots, E_n) \mathrel{\overline{:}} (v_1, \ldots, v_n)}$$

$$\frac{\kappa, \chi, \phi, \psi, \rho \vdash E \mathrel{\overline{:}} C_i(v) \quad v \vdash P_i \mathrel{\overline{\rightsquigarrow}} \rho'/P' \quad \kappa, \chi, \phi, \psi, \rho \oplus \rho' \vdash E_i \mathrel{\overline{:}} w}{\kappa, \chi, \phi, \psi, \rho \vdash \texttt{case}\ E\ \texttt{of}\ C_1\ P_1 \texttt{=>} E_1\ | \ldots |\ C_n\ P_n \texttt{=>} E_n \mathrel{\overline{:}} w}$$

$$\frac{\kappa, \chi, \phi, \psi, \rho \vdash E \mathrel{\overline{:}} E' \quad \kappa, \chi, \phi, \psi, \rho \vdash B_i \mathrel{\underline{\overline{:>}}} Bs_i \quad \text{for } i = 1, \ldots, n}{\kappa, \chi, \phi, \psi, \rho \vdash \underline{\texttt{case}}\ E\ \underline{\texttt{of}}\ B_1 | \ldots | B_n \mathrel{\overline{:}} \texttt{case}\ E'\ \texttt{of}\ Bs_1 | \ldots | Bs_n}$$

Dynamic branch:
$(Conid \rightarrow BTtype), (Conid \rightarrow \{Pval\}), (Funid \rightarrow BTtype), (Funid \rightarrow \{Pval\}), (Varid \rightarrow Pval) \vdash ABranch \mathrel{\underline{\overline{:>}}} Branch^*_{|}$

$$\frac{\kappa C = BT \rightarrow Ti \quad \chi C = \{v_1, \ldots, v_n\} \quad \kappa, BT, P \vdash v_i \mathrel{\overline{\rightsquigarrow}} \rho_i/P_i \quad \kappa, \chi, \phi, \psi, \rho \oplus \rho_i \vdash E \mathrel{\overline{:}} E_i \quad \text{for } i = 1, \ldots, n}{\kappa, \chi, \phi, \psi, \rho \vdash C\ P \texttt{=>} E \mathrel{\underline{\overline{:>}}} (C, v_1)\ P_1 \texttt{=>} E_1\ |\ \ldots\ |\ (C, v_n)\ P_n \texttt{=>} E_n}$$

Splitting into static and dynamic parts: $(Conid \rightarrow BTtype), BTtype \vdash Pval \nearrow Val/Exp$

$$\frac{}{\kappa, \underline{T} \vdash E \nearrow ()/(E)}$$

$$\frac{}{\kappa, Ti \vdash v \nearrow v/()} \quad \text{if } Ti \text{ is a base type}$$

$$\frac{\kappa C = BT \rightarrow Ti \quad \kappa, BT \vdash v \nearrow v'/E}{\kappa, Ti \vdash C(v) \nearrow C(v')/E} \quad \text{if } Ti \text{ is a data type}$$

$$\frac{\kappa, BT_1 \vdash v_1 \nearrow v_1'/E_1 \quad \ldots \quad \kappa, BT_n \vdash v_n \nearrow v_n'/E_n}{\kappa, BT_1 * \ldots * BT_n \vdash (v_1, \ldots, v_n) \nearrow (v_1', \ldots, v_n')/E_1 \mathbin{++} \ldots \mathbin{++} E_n}$$

Figure 9: Modified parser

```
datatype Gram = EMPTY
              | REC
              | SYMB of string
              | SEQ of Gram*Gram
              | ALT of Gram*Gram

datatype ToDo = DONE | DO of Gram*ToDo | GENERALIZED ToDo1

datatype ToDo1 = GENERALIZE ToDo

fun parse(g,s) = p(g,s,g,GENERALIZED(GENERALIZE(DONE)))

fun p(g,s,g0,todo) =
      case g of
        EMPTY       => nowdo(todo,s,g0)
      | REC         => p(g0,s,g0,generalize(todo))
      | SYMB c      => s<>[] andalso (hd s)=c andalso nowdo(todo,tl s,g0)
      | SEQ (g1,g2) => p(g1,s,g0,DO(g2,todo))
      | ALT (g1,g2) => p(g1,s,g0,todo) orelse p(g2,s,g0,todo)
end;

fun generalize(todo) =
      case todo of
       GENERALIZED(t) => t
       _              => GENERALIZED(GENERALIZE(todo))
end;

fun nowdo(todo,s,g) =
      case todo of
        DONE          => s=[]
      | DO (g1,todo') => p(g1,s,g,todo')
      | GENERALIZED t => case t of GENERALIZE todo' => nowdo(todo',s,g)
end;
```

Figure 10: Residual modified parser

```
datatype ToDo1' = GENERALIZE_0 | GENERALIZE_1 of ToDo1'

fun parse'(s) = p'(s,GENERALIZE_0)

fun p'(s,todo) =
      (s<>[] andalso (hd s)="(" andalso p'(tl s,GENERALIZE_1(todo)))
      orelse
      (case todo of
         GENERALIZE_0        => s = []
       | GENERALIZE_1(todo') => (s<>[] andalso (hd s)=")" andalso p'(tl s,todo'))
end;
```

would have lead to extra dynamic construction and decomposition.

We get the residual program shown in figure 10. All calls to `nowdo` and all calls to `p` except the initial call and the call in the `REC` branch are unfolded. It is now clear from the residual program that the depth of the stack is the number of outstanding parenthesis. Note that the `todo` argument of `p` has been retyped: where it originally was `ToDo` it is now a specialized instance of `ToDo1`, which is the type of the dynamic argument to a constructor (`GENERALIZED`) of the original type.

The "trick" of generalizing unbounded structures only at particular places, guarded by constructors like `GENERALIZE` and `GENERALIZED` corresponds closely to a similar "trick" which is used to generalize continuations in Similix, where the guards are eta-conversions ([Bon91], [Mos93]).

9 Conclusion and further work

I first got the idea of constructor specialization in 1987 when I was writing an interpreter for a functional language in a term rewrite language. This interpreter used a stack and partial evaluatially evaluating it caused the same problems with unbounded stacks as shown in the example here. I considered ways of handling this problem and found that constructor specialization would help. I didn't follow up on the idea, as my attention turned elsewhere. But I have always thought the idea had merit, so I have now tried to formalize the idea and present them to a greater public.

I believe constructor specialization will have most impact on two kinds of program: those that use a stack or similar structure (as shown in the example) and interpreters that use a single data type to encode a large family of types in the implemented languages. The latter problem has been investigated in [Lau91b] and [HL91] in the context of self-appliction, where a double level of coding is needed. Constructor specialization does not handle the problem of large space requirements that these papers have different solutions to, but instead improves the quality of the residual programs (e.g. compiled programs) by "inventing" suitable data types for each residual program. As mentioned in section 7, the presented specialization method needs a further refinement to handle this really well. In higher order languages the problem of unbounded stack can sometimes be solved by using continuations. This is the case with the parser shown in the example, and it is also the solution chosen in [Mos93]. There is no similar approach that will work for coding of data-structures in interpreters. It is in general possible to replace arbitrary algebraic types with higher order functions, but it doesn't yield the desired result and will in most cases cause problems in typed languages, when recursive data types (legal) are replaced by recursive function types (illegal).

In the abstract I said that the ideas are applicable to other languages and mentioned Haskell and Prolog. Haskell is sufficiently similar to ML that this is no surprise. Prolog has two properties that can cause problems: it is untyped and it allows unification of terms containing unbound variables. The untyped nature is not much of a problem, as specialization is mostly at the constructor level rather than the type level anyway. Unification can be a problem, though. It boils down to what happens when two differently specialized versions of the same constructor are unified. If the static arguments to which they are specialized are unifiable, the unification should succeed. But if different names are given to the specialized constructors, unification will fail. A solution to this problem is to make sure that the static parts are only unifiable if they are in fact equal. If they are ground, this is cartainly true. It would therefore seem reasonable to require the parameters to which constructors are specialized to be ground. In Logimix [MB93], predicates are (for similar reasons) only specialized with respect to ground parameters, so it is not unreasonable that the same should apply to constructors. Another problem is what happens when specialized constructors are unified with values supplied at runtime to the residual program. This problem is similar to the problem of specialized constructors in input and output mentioned in section 7, and can indeed be solved in a similar fashion.

At present (March 1993) I have no implementation of constructor specialization except for a prototype binding time analysis, so this is obviously the next step. When the basic method is implemented the refinements mentioned in section 7 can be considered. It is possible that constructor specialization can be added to Similix's partially static data-type feature [Bon93]. I will discuss this with Anders Bondorf.

References

[BEJ88] D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987.* Amsterdam: North-Holland, 1988.

[Bon91] A. Bondorf. Compiling laziness by partial evaluation. In S.L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 9–22, Berlin: Springer-Verlag, 1991.

[Bon93] A. Bondorf. *Similix Manual, System Version 5.0.* Technical Report, DIKU, University of Copenhagen, Denmark, 1993.

[Bul88] M.A. Bulyonkov. A theoretical approach to polyvariant mixed computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 51–64, Amsterdam: North-Holland, 1988.

[DNBDV91] A. De Niel, E. Bevers, and K. De Vlaminck. Program bifurcation for a polymorphically typed functional language. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 142–153, New York: ACM, 1991.

[GJ91] C.K. Gomard and N.D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[GR92] M. Gengler and B. Rytz. A polyvariant binding time analysis handling partially known values. In M. Billaud et al., editors, *WSA '92, Static Analysis, Bordeaux, France, September 1992. Bigre vols 81–82, 1992*, pages 322–330, Rennes: IRISA, 1992.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 448–472, ACM, Berlin: Springer-Verlag, 1991.

[HH91] C.K. Holst and J. Hughes. Towards binding-time improvement for free. In S.L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 83–100, Berlin: Springer-Verlag, 1991.

[HL91] C.K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218, Glasgow University, 1991.

[JSS85] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140, Berlin: Springer-Verlag, 1985.

[JSS89] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[Lau89] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, University of Glasgow, November 1989. Revised version in [Lau91a].

[Lau91a] J. Launchbury. *Projection Factorisations in Partial Evaluation.* Cambridge: Cambridge University Press, 1991.

[Lau91b] J. Launchbury. A strongly-typed self-applicable partial evaluator. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 145–164, ACM, Berlin: Springer-Verlag, 1991.

[MB93] T. Mogensen and A. Bondorf. Logimix: a self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *LOPSTR 92. Workshops in Computing*, Berlin: Springer-Verlag, January 1993.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347, Amsterdam: North-Holland, 1988.

[Mog89] T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *TAPSOFT '89. Proc. Int. Conf. Theory and Practice of Software Development, Barcelona, Spain, March 1989 (Lecture Notes in Computer Science, vol. 352)*, pages 298–312, Berlin: Springer-Verlag, 1989.

[Mog92] T. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 116–121, New Haven, CT: Yale University, 1992.

[Mos93] C. Mossin. Partial evaluation of general parsers. In *PEPM '93: ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New York: ACM, 1993.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Rom90] S.A. Romanenko. Arity raiser and its use in program specialization. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990 (Lecture Notes in Computer Science, vol. 432)*, pages 341–360, Berlin: Springer-Verlag, 1990.

[Ses86] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985 (Lecture Notes in Computer Science, vol. 217)*, pages 236–256, Berlin: Springer-Verlag, 1986.