Evolution of Partial Evaluators: Removing Inherited Limits

Torben Æ. Mogensen

DIKU, University of Copenhagen, Denmark

Abstract. We show the evolution of partial evaluators over the past ten years from a particular perspective: the attempt to remove limits on the structure of residual programs that are inherited from structural bounds in the original programs. It will often be the case that a language allows an unbounded number or size of a particular features, but each program (being finite) will only have a finite number or size of these features. If the residual programs cannot overcome the bounds given in the original program, that can be seen as a weakness in the partial evaluator, as it potentially limits the effectiveness of residual programs. The inherited limits are best observed through specializing a self-interpreter and examining the object programs produced by specialisation of this. We show how historical developments in partial evaluators gradually remove inherited limits, and suggest how this principle can be used as a guideline for further development.

1 Introduction

Much has happened in the field of partial evaluation in the past ten years. The evolution has taken many different paths, focusing on different problems. We will show below that many seemingly unrelated developments serve similar purposes: removal of *inherited limits*. More often than not, the developments have been introduced for solving immediate practical problems observed through experiments, without explicit awareness of the general underlying principle.

In this paper we first introduce the concept of inherited limits, and discuss why we think it may be a problem. We then, as a case study, define a small first-order functional language and show how successive refinements of an initially very simple specialisation method each remove an inherited limit, until our final version can be seen to have none. We then round up by discussing how to use the principle of inherited limits to suggest further studies in partial evaluation methods, and noting some important developments in partial evaluation that can not be related to removal of inherited limits.

2 Inherited Limits

It has been stated (I forget by whom) that in computer science (as well as many other areas) there are only three reasonable limits: zero, one and infinity.

This principle has often been used in language design, such that there typically is no arbitrary bounds on nesting depth, size of program, number of variables etc, unless that bound is uniformly set to one, as e.g. the nesting depth of function definitions in C.

But while the language as a whole imposes no such bounds, each program (being finite) will normally be limited to a finite depth, size or number of these features. For example, all Pascal programs will have a finite nesting depth of functions, a finite number of variables and a finite dimension of arrays, even though the language definition imposes no bound on these features.

The fact that any single program only uses a bounded number or size of a feature does not usually cause any problem, and there really is no natural way to avoid it, even if it did.

It is when new programs are derived by specialisation or other transformations from a single original programs that we may encounter a problem: the derived programs may not be able to exceed the limits imposed by the original program. Why is this a problem? Philosophically speaking, we can see that the derived programs can not use the full potential of the programming language: a better result may be obtained by using one more level of nesting, one more variable or one more dimension in an array. It is less clear to what extent this is a practical problem, though. I will argue that if partial evaluators are to be used for compiling by specialising interpreters, then it is indeed a problem.

When compiling by partial evaluation, each object program is a specialised version of the interpreter. The interpreter, being a single program, will have finite bounds on features that the language itself has no limitation for. If the residual programs can not exceed these bounds, only a subset of the target language is exploited. If the source language has uniform limits for these features, it is no problem. But if not, in particular if the source language is an extended version of the target language, this is a problem.

In the "challenging problems" collection from the '87 PEMC workshop, (Neil Jones 1987) suggested that a partial evaluator is "strong enough" (later the term *optimal* is used) if it is possible to completely remove the interpretation overhead of a self-interpreter by specialisation. The case of self-interpretation is interesting, as it is easy to judge to what extent the interpretation overhead has been removed by comparing the source and target programs, as these are in the same language. If the target program is identical to the source program (modulo renaming and reordering) it is safe to conclude that all interpretation overhead has been removed.

Note that for the test above to be a test on the quality of the partial evaluator, not the self-interpreter, we must not insist on a particular self-interpreter. We must argue that no matter how a self-interpreter is written, we can not obtain optimality. This is where considering inherited limits becomes a useful yardstick: if the partial evaluator inherits a bound from the original program relating to a feature that the language has no limit for, then we can argue that optimality can not be achieved.

Inherited limits are only interesting for features where it costs something to

$$\begin{array}{lll}
\mathit{Program} & \rightarrow & \mathit{TypeDecl}^{*}\ \mathit{FunDecl}^{+} \\[1ex]
\mathit{TypeDecl} & \rightarrow & \texttt{data}\ \mathit{TypeId}\ =\ \mathit{ConDecl} \\[1ex]
\mathit{ConDecl} & \rightarrow & \mathit{ConId}\ \mathit{TypeId}^{*} \\
 & \mid & \mathit{ConId}\ \mathit{TypeId}^{*}\ \mid\ \mathit{ConDecl} \\[1ex]
\mathit{FunDecl} & \rightarrow & \mathit{FunId}\ \mathit{VarId}^{*}\ =\ \mathit{Expr} \\[1ex]
\mathit{Expr} & \rightarrow & \mathit{Varid} \\
 & \mid & \mathit{FunId}\ \mathit{Expr}^{*} \\
 & \mid & \mathit{ConId}\ \mathit{Expr}^{*} \\
 & \mid & \texttt{case}\ \mathit{Expr}\ \texttt{of}\ \mathit{Branch}^{+} \\[1ex]
\mathit{Branch} & \rightarrow & \mathit{ConId}\ \mathit{VarId}^{*}\ \texttt{=>}\ \mathit{Expr}
\end{array}$$

Fig. 1. Syntax of a small functional language

```
data int  = Zero | Succ int
data list = Nil | Cons int list

f a b = case a of
          Nil      => b
          Cons n l => Cons (g n) (f b l)

g n = case n of
        Zero   => Zero
        Succ m => m
```

Fig. 2. A small program.

simulate an unbounded number/size with a bounded number/size. This is the usual case, but one can argue that one-dimensional arrays can simulate multi-dimensional arrays at no cost, since the code generated for the explicit index calculation using one-dimensional arrays is the same as the code generated by a compiler for index calculation in multi-dimensional arrays. Such arguments are, however, often implementation-specific and not always convincing.

As a case study we will successively refine a partial evaluator, removing inherited limits as we go along. To do so, we define a small first-order functional language and a self-interpreter for this language. We will test how well the partial evaluator can compile a small test program by specialising the self-interpreter to this program, and suggest changes to the partial evaluator from observed

```
data num   = Z | S num
data univ  = Con num ulist
data ulist = Un | Uc univ ulist
data funs  = Fn | Fun exp funs
data exp   = Var num | Fap num elist | Cap num elist | Case exp elist
data elist = En | Ec exp elist

append a b = case a of
               Un     => b
               Uc v a => Uc v (append a b)

hd vs = case vs of
          Un    => Con Z Un
          Uc v vs => v

tl vs = case vs of
         Un    => Un
         Uc v vs => vs
```

Fig. 3. Type declarations and utility functions used by self-interpreter.

deficiencies.

3 A Small Language

We have chosen a very simple language as our test-bed. We must not be too simple, however, as we must have some limits to remove. An example of too simple a language is the pure lambda calculus, where a five-line partial evaluator is able to remove all interpretation overhead when specialising a three-line self-interpreter as seen in (Mogensen 1992). A suitable compromise is the language whose syntax is described in Fig. 1. A small program in the language is shown in Fig. 2.

We have non-nested function definitions working over simple algebraic data types. Programs are assumed to be type-correct (monomorphically). Case-expressions are exhaustive. A self-interpreter for the language is shown in Fig. 4. The self-interpreter uses some type declarations and utility functions shown in Fig. 3. We will not show these in the programs below unless they are changed.

The type `univ` is a universal data type, used to represent values of any type. Elements of `univ` consists of a constructor number and a list of values. The types `funs`, and `exp` represent syntax of functions, expressions. The types `ulist`, `funs` and `elist` represent lists of values, function definitions and expressions. Due to the monomorphic restriction, separate types are required for these. Variables are referred to by their expected position in the environment, and functions by their position in the program. The type declarations of the source program are not represented in the self-interpreter, as they have no operational significance.

```
run p args = apply Z p args p

apply f fs args p =
  case fs of
    Fn       => Con Z Un
    Fun e fs => case f of
                  Z   => eval e args p
                  S f => apply f fs args p

eval e vs p =
  case e of
    Var n     => lookup n vs
    Fap n es  => apply n p (evallist es vs p) p
    Cap n es  => Con n (evallist es vs p)
    Case e es => case (eval e vs p) of
                   Con n vs1 => eval (select n es) (append vs1 vs) p

evallist es vs p =
  case es of
    En      => Un
    Ec e es => Uc (eval e vs p) (evallist es vs p)

lookup n vs =
  case n of
    Z   => hd vs
    S n => lookup n (tl vs)

select n es =
  case es of
    En      => Cap Z En
    Ec e es => case n of
                 Z   => e
                 S n => select n es
```

Fig. 4. Self-interpreter for a small functional language.

The function `apply` takes as arguments a function index (`f`), a list of function definitions (`fs`), a list of arguments to the function (`args`) and the representation (`p`) of the entire program. `apply` uses the index to find the corresponding function definition and passes the body expression, the arguments and the program to `eval`. `eval` evaluates the expression in the environment defined by `vs`. It uses the program representation `p` when interpreting function applications.

```
data num   = Z | S num
data univ  = Con num ulist
data ulist = Un | Uc univ ulist
data funs  = Fn | Fun exp funs
data exp   = Var num | Fap num elist | Cap num elist | Case exp elist
data elist = En | Ec exp elist

Fun (Case (Var Z)
          (Ec (Var (S Z))
          (Ec (Cap (S Z) (Ec (Fap (S Z) (Ec (Var Z) En))
                         (Ec (Fap Z (Ec (Var (S (S (S Z))))
                                    (Ec (Var (S Z)) En)))
                          En)))
           En)))
(Fun (Case (Var Z)
           (Ec (Cap Z En)
           (Ec (Var Z)
            En)))
 Fn)
```

Fig. 5. Representation of the test program as data.

4 Specialisation Methods

We now study a number of specialisation methods for the small language introduced above. We will deliberately avoid details about the implementation of these methods, we will describe *what* is to be done rather than *how*. In particular, we will try not to make the description specific to online or offline specialisation, nor will we consider self-application.

Since we want to state properties about the partial evaluators independently of a particular self-interpreter, we will allow ourselves to rewrite the self-interpreter to get maximum benefit from each partial evaluator. In essence, we will perform binding time improvements (Jones *et al.* 1993, chapter 12) as we see fit.

4.1 Method 1: Constant Folding

Our first simple algorithm for partial evaluation is *constant folding*, essentially as found in many optimising compilers. While the language is different, the method closely resembles "global" constant propagation as described e.g. in (Kildall 1973).

The essence is: if a formal parameter to a function is given the same known value in all calls to a function, the parameter is eliminated from the definition and all calls and all further references to it are replaced by the value. Local reductions are performed.

```
run args = apply Z (Fun ...) args

apply f fs args =
  case fs of
    Fn       => Con Z Un
    Fun e fs => case f of
                  Z   => eval e args
                  S f => apply f fs args

eval e vs =
  case e of
    Var n     => lookup n vs
    Fap n es  => apply n (Fun ...) (evallist es vs)
    Cap n es  => Con n (evallist es vs)
    Case e es => case (eval e vs) of
                   Con n vs1 => eval (select n es) (append vs1 vs)

evallist es vs =
  case es of
    En      => Un
    Ec e es => Uc (eval e vs) (evallist es vs)

lookup n vs =
  case n of
    Z   => hd vs
    S n => lookup n (tl vs)

select n es =
  case es of
    En      => Cap Z En
    Ec e es => case n of
                 Z   => e
                 S n => select n es
```

Fig. 6. Residual program made by method 1.

This is a very primitive method, and indeed little reduction is done when specialising an interpreter. The result of specialising the self-interpreter in Fig. 4 to the test program in Fig. 2 is shown in Fig. 6.

The two occurrences of "(Fun ...)" actually refer to the entire representation of the test program, as shown in Fig. 5. Note that we have been able to remove the p parameter from both apply and eval. This is likely to give a small speedup, but nothing dramatic. The source program is still essentially interpreted in the residual program.

If we do not count the size of in-lined constants, the residual programs produced by constant folding are never larger than the original program. This is a very severe inherited limit, which we remove in method 2.

4.2 Method 2: Adding Unfolding

In order to remove the inherited limit of program size, we introduce unfolding of function calls. This resembles loop-unrolling and inlining as done by optimising compilers.

In our extended specialisation method, we essentially apply constant folding *after* call unfolding and local reduction of case-expressions. This means that we can eliminate a parameter if it is identical in all *non-unfolded* calls, even if it is non-constant in unfolded calls.

The decision of which function calls to unfold can be made off-line by program annotation, inserted manually as in (Jones *et al.* 1985) or automatically as in (Bondorf 1993), or it can be done on-line by studying the computation history as in e.g. (Berlin and Weise 1990). For simplicity of presentation, we have chosen the manual off-line approach.

Before:

```
... eval (select n es) (append vs1 vs) p

select n es =
  case es of
    En      => Cap Z En
    Ec e es => case n of
                 Z   => e
                 S n => select n vs
```

After:

```
... eselect n es (append vs1 vs) p

eselect n es vs p =
  case es of
    En      => Con Z Un
    Ec e es => case n of
                 Z   => eval e vs p
                 S n => eselect n es vs p
```

Fig. 7. Applying "The Trick" to selection of case-branches.

We choose to unfold all calls to `eval`, `evallist`, `lookup` and `select` and all directly recursive calls to `apply`. It is safe also to unfold `hd` and `tl`, but doing so will make the residual program much larger and less readable (though a bit faster, since we eliminate some calls). The reason for not unfolding these is hence mostly expository.

Unfortunately, our present interpreter doesn't yield good results with this method. The reason is that we do not know which case-branch is selected (the result of `select` is unknown) when interpreting a case-expression. This means

```
run args = apply Z args

apply f args =
 case f of
   Z   =>
    case (hd args) of
      Con n vs1 =>
        case n of
          Z   => hd (tl args)
          S n =>
           case n of
             Z   =>
              Con (S Z)
                  (Uc (apply (S Z)
                             (Uc (hd (append vs1 args)) Un))
                  (Uc (apply Z
                             (Uc (hd (tl (tl (tl (append vs1 args))))) Un)
                             (Uc (hd (tl (append vs1 args))) Un)))
                   Un))
             S n => Con Z Un
   S f =>
    case f of
      Z   =>
        case (hd args) of
          Con n vs1 =>
            case n of
              Z   => Con Z
              S n =>
               case n of
                 Z   => hd (append vs1 args)
                 S n => Con Z Un
      S f => Con Z Un
```

Fig. 8. Residual program made by method 2.

that the expression to evaluate is not always known, even with full unfolding of `eval`. We do, however, know that the expression must be one of the case-branches, so we can apply "The Trick", a commonly used binding time improvement described in (Bondorf 1993). We just move the recursive call to `eval` into the `select` function, obtaining a new function `eselect`. In the new version, all calls to `eval` will have known expression arguments. The transformation is shown in Fig. 7.

The result of specialising the modified self-interpreter with respect to the test program is shown in Fig. 8.

The resulting target program is about the same size as the original interpreter[1],

[1] It would have been larger, had we unfolded `hd` and `tl`.

so this in itself doesn't show that the inherited limit has been removed. However, it is easily seen that the size of the residual program grows with the size of the source program. Closer inspection shows that the dispatch on syntax of expressions has been eliminated, removing a large part of the interpretation overhead.

The result is, however, still a far cry from being identical to the source program. Looking at inherited limits we quickly note that any target program will have exactly 3 functions: `run`, `apply` and `append`. This is less than the number of functions in the interpreter, and this is indeed the case regardless of which interpreter is chosen, as the specialiser can only eliminate functions and parameters, not invent new functions.

4.3 Method 3: Polyvariant Specialisation

```
run args = apply_0 args

apply_0 args =
 case (hd args) of
   Con n vs1 =>
     case n of
       Z   => hd (tl args)
       S n =>
        case n of
          Z   =>
           Con (S Z)
               (Uc (apply_1
                          (Uc (hd (append vs1 args)) Un))
               (Uc (apply_0
                          (Uc (hd (tl (tl (tl (append vs1 args))))) Un)
                          (Uc (hd (tl (append vs1 args))) Un)))
                Un))
          S n => Con Z Un

apply_1 args =
  case (hd args) of
    Con n vs1 =>
      case n of
        Z   => Con Z
        S n =>
         case n of
           Z   => hd (append vs1 args)
           S n => Con Z Un
```

Fig. 9. Residual program made by method 3.

As most readers will have guessed by now, the limit on the number of functions can be removed by using *polyvariant specialisation*, allowing each original function to be specialised to several different versions, depending on the known input. This idea was essential for the success of the MIX-project described in (Jones *et al.* 1985), but had earlier been applied to imperative languages e.g. by Itkin (1983) and Bulyonkov (1988).

Stated shortly, our third method is:

Unfoldable calls are unfolded, and local reductions are made. In non-unfoldable calls, the values of the known arguments are used to create specialised variants of the function (one for each value of the known arguments). Each specialised variant has a unique name, and the known parameters are eliminated.

Since the number of variants of a function depends on the number of different known arguments to it (which may be unbounded), we have removed the inherited limit. The residual program obtained by applying polyvariant specialisation to the interpreter with respect to our test program is shown in Fig. 9.

We note that there now is a specialised version of `apply` for each function in the source program. We have, however, not significantly reduced the interpretation overhead compared to the previous version. For some applications, though, polyvariant specialisation is essential to obtaining good results.

Looking at Fig. 9 we note that all specialised versions of `apply` have exactly one argument: `args`. It is easy to see that a specialised variant of a function can have no more parameters than the original, since a parameter is either eliminated (if known) or retained (if unknown). This is the next inherited limit we will remove.

4.4 Method 4: Partially Static Values

There are several ways to remove the limitation on the number of parameters. In the MIX project (Jones *et al.* 1985), a post process was at one stage used, which guided by user annotations would split a residual parameter into several. This was automated by Romanenko (1990). We will, instead, use the approach from (Mogensen 1988), where we keep track of known parts of values that may contain unknown components. When specialising a function it is specialised with respect to the known parts of the partially known arguments, and each unknown part becomes a separate argument. The number of parameters to a specialised function hence depends on the number of unknown components of a partially known value. Since this can be unbounded, we have removed the limit.

The prime targets for partially known values in our interpreter are the parameter `args` in `apply` and the parameter `vs` in `eval`, `evallist` and `lookup`.

While the elements of the `arg` are unknown at specialisation time, the structure (length) is known (assuming it is known in the initial call to `run`, which is not unreasonable), since it is built by `evallist`, and hence gets the same length as the list of expressions `es`.

We might expect the same to be true for `vs`, but unfortunately this is not so. The reason is that the contents of a constructed value is appended to `vs` when interpreting a case-expression. Following the idea from our previous binding time

Before:

```
data exp   = Var num | Fap num elist | Cap num elist | Case exp elist

... eselect n es (append vs1 vs) p

eselect n es vs p =
  case es of
    En      => Con Z Un
    Ec e es => case n of
                 Z   => eval e vs p
                 S n => eselect n es vs p
```

After:

```
data exp   = Var num | Fap num elist | Cap num elist | Case exp blist
data blist = Bn | Bc num exp blist

... bselect n bs vs1 vs p

bselect n bs vs1 vs p =
  case bs of
    Bn        => Con Z Un
    Bc m e bs => case n of
                   Z   => append_eval m vs1 Un vs e p
                   S n => bselect n bs vs p

append_eval m vs1 vs2 vs e p =
  case m of
    Z   => b
    S m => case vs1 of
             Un       => eval e (rev vs2 vs) p
             Uc v vs1 => append_eval m vs1 (Uc v vs2) vs e p

rev vs2 vs =
  case vs2 of
    Un       => vs
    Uc v vs2 => rev vs2 (Uc v vs)
```

Fig. 10. Binding time improvements for partially static values.

improvement, we will move the appending into the `eselect` function, so we know which constructor was applied when we do the appending. Unfortunately, we do not with our present program representation have sufficient information to know the length of the list we append to `vs`, even when we know the constructor. So we need yet another binding time improvement, this time affecting our program representation: we must explicitly indicate the arity of constructors in case expressions. The required changes are shown in Fig. 10. All the shown functions are unfolded during specialisation.

```
run args_0 args_1 = apply_0 args_0 args_1

apply_0 args_0 args_1 =
 case args_0 of
   Con n vs1 =>
     case n of
       Z   => args_1
       S n =>
        case n of
          Z   =>
           case vs1 of
             Un        => Con Z Un
             Uc v0 vs1 =>
               case vs1 of
                 Un        => Con Z Un
                 Uc v1 vs1 =>
                   Con (S Z)
                     (Uc (apply_1 v0)
                     (Uc (apply_0 args_1 v1)
                      Un))
          S n => Con Z Un

apply_1 args_0 =
  case args_0 of
    Con n vs1 =>
      case n of
        Z   => Con Z
        S n =>
         case n of
           Z   => case vs1 of
                    Un        => Con Z Un
                    Uc v vs1 => v
          S n => Con Z Un
```

Fig. 11. Residual program made by method 4.

The binding time improvement could be simplified somewhat if we could return a partially static value across a dynamic conditional (which is possible with CPS-based specialisation as described in (Bondorf 1992)), as we then just would have needed to add an extra controlling argument to append.

Now we will also unfold append, hd and tl, as we can eliminate these now the structure (spine) of the environment is static.

Applying specialisation with partially static values to the new version of the interpreter and the test program, we get the target program shown in Fig. 11.

We can see that the two specialised versions of apply have different numbers of parameters, corresponding to the parameters of the functions in the source program. While neither of these are larger than the original number of parameters to apply, other source programs can cause that. The target program is also both smaller and faster than the previous, having replaced the nested case-expressions used for variable lookup by direct references to new variable names. The case switching on vs1 comes from append_eval.

The goal function run now has two parameters, which are of a different type to the args argument of the original run function. Strictly speaking, this means that the residual program is not correct with respect to the usual mix-equation (Jones *et al.* 1993). But there is a well-defined correspondence between the two new arguments and the single old argument, so this is not a problem in practice.

While our target program now fits a page, it is still far larger than the test program, and a good bit slower. A source of inefficiency is that all values in the interpreted program are represented using a single uniform representation. More specifically, the constructors in the residual program are always a subset of the constructors in the original program. Since the language as a whole puts no limit on the number of different constructors, we clearly have an inherited limit.

4.5 Method 5: Constructor Specialisation

We removed the inherited limit on functions by introducing polyvariant specialisation of functions. We will use a similar solution to the constructor problem: polyvariant specialisation of constructors as introduced in (Mogensen 1993).

The observation is that we sometimes lose information during specialisation. This is for example when a partially static value is returned from a non-unfolded function. Instead of converting such partially static values to (completely dynamic) residual expressions, we can specialise the topmost constructor with respect to the static parts of its arguments, leaving the dynamic parts as separate arguments to the new constructor, exactly as we did for function calls in method 4. In addition to noting the new name and the types of the residual arguments in the type declaration, all case expressions that branch on the original constructors must now branch on the specialised constructors. The branches of these case expressions can use the known parts of the arguments to the constructor, just like residual functions can use the known parts of the arguments to the function.

In our interpreter, the Con constructor of the univ type is our target for constructor specialisation. We can specialise this with respect to its tag value and the structure of the list of arguments. Hence, each specialised version of Con will have the same number of parameters as the source-program constructor it represents.

An interesting consequence of using constructor specialisation in our interpreter is that the binding time improvements we introduced along the way are no longer necessary: the original version from Fig. 4 yields exactly the same residual programs as the binding-time improved one.

The target program obtained by constructor specialisation is shown in Fig. 12.

Zero and Succ are represented by the specialised constructors Con_0 and Con_1, and Nil and Cons by Con_0 and Con_2. The specialised constructors

```
data univ  = Con_0 | Con_1 univ | Con_2 univ univ

run args_0 args_1 = apply_0 args_0 args_1

apply_0 args_0 args_1 =
 case args_0 of
   Con_0       => args_1
   Con_1 v0    => Con_0       /* unreachable */
   Con_2 v0 v1 => Con_2 (apply_1 v0) (apply_0 args_1 v1)

apply_1 args_0 =
  case args_0 of
    Con_0       => Con_0
    Con_1 v0    => v0
    Con_2 v0 v1 => v0         /* unreachable */
```

Fig. 12. Residual program made by method 5.

have the same number of arguments as those that they represent, so we have not only removed the limit on the number of different constructors, but also on the number of arguments to these. There is a similar problem with respect to the mix-equation, as the one noted in Sect. 4.4, but again this is not a problem in practice.

The number of operations performed by the residual program when running is the same as for the source program, except for a single call from `run` to `apply_0`. But the target program is not just a renaming of the source program: there are "dead" branches in both case-expressions and there is only a single type definition. Both of these problems are caused by having a single residual type for all specialised constructors from a single source type.

In our example the dead branches are relatively harmless, but it is conceivable that dead branches can be a sizable fraction of the target program or cause errors/looping during specialization. Also, only if we assume constant time branch selection, irregardless of the number of branches, have we eliminated all interpretation overhead.

4.6 Method 6: Type Specialisation

In order to remove the inherited limit of the number of types in the residual program, we allow different specialised versions of the same original constructor to be in different residual types. The system described by Dussart (1995) does this. The following method is not exactly the same, but shows the essential idea:

When a constructor is specialised, a new type is created for the specialised instance. Only when necessary (e.g. when occurring in different branches of the same case-expression) are types joined. The same constructor applied to the same static arguments may occur in several types, if the uses are independent.

```
data univ_0  = Con_0 | Con_1 univ_0
data univ_1  = Con_2 | Con_3 univ_0 univ_1

run args_0 args_1 = apply_0 args_0 args_1

apply_0 args_0 args_1 =
 case args_0 of
   Con_2       => args_1
   Con_3 v0 v1 => Con_2 (apply_1 v0) (apply_0 args_1 v1)

apply_1 args_0 =
  case args_0 of
    Con_0       => Con_0
    Con_1 v0    => v0
```

Fig. 13. Residual program made by method 6.

Hence, the residual name of a specialized constructor depends not only on the static arguments, but also on the type it will eventually be part of.

The residual program obtained by type specialisation is shown in Fig. 13.

Now we have two specialised versions of the `univ` type. `univ_0` corresponds to the `int` type of the source program, and `univ_1` corresponds to `list`. `Con_0` and `Con_2` both correspond to `Con Z Un`, but are given different names because the residual types are different.

With the sole exception of the superfluous `run` function, the residual program is just a renaming of the source program. We can see that (after removing `run` and with some simple assumptions about how types are joined) the target program will *always* be a renaming of the source program. Hence, we can reasonably conclude that all interpretation overhead has been removed. Similarly, we can conclude that all inherited limits have been removed.

5 Other Evolution and Natural Selection

Our focus on a single aspect of the evolution of partial evaluation is in some way similar to the way palaeontologists may focus on the evolution of the thigh bone from early salamanders to humans. While it shows an important aspect of the adaptation to land-based life, it is not the whole story.

One may choose to focus on entirely different aspects to show how the field has developed. Indeed, Olivier Danvy (1993) has suggested that reducing the need for manual binding time improvement is a driving force behind much of the developments of partial evaluators – something we have explicitly chosen to ignore in this study.

There may also be different ways to remove inherited limits than the ones shown above. We noted that the limit on number of parameters has been handled differently, by a post-process, e.g. in (Romanenko 1990). Similarly, it is possible that limits on types and constructors can be handled by post processing. When several methods exist, a form of "natural selection" may determine which method survives. By all appearances, the use of partially static structures has ousted post-processing as a means of removing the bound on parameters. It is too early yet to tell for the later developments. Also, some inherited limits may be considered unimportant (as they have little significance for effectivity), so no method for removing them will be widely used.

As we saw in Sect. 4.5, some methods for removing inherited limits may also reduce the need for binding time improvements. This is true also for partially static structures, and was indeed one of the original motivations for introducing these. This may have been a major reason for the success of this approach over arity raising (Romanenko 1990), which only solves the variable splitting problem, and bifurcation (Mogensen 1989), (De Neil *et al.* 1990), which only solves the binding time improvement aspect.

6 Further Work

By removing all inherited limits in our example language, we may have given the impression that the problem is now solved. This is far from the case. Our language is very simple, and has few features. Every time a new feature is added to a language, a new inherited limit may potentially be added to a partial evaluator, even when it has been extended to handle the new feature. Below, we list a number of common features of programming languages, and discuss the principle of removing inherited limits that these features may introduce.

Nesting of Scopes. Our simple language has flat scoping, but it is common to have unbounded nesting of functions or procedures in programming languages (a notable exception being C). It turns out to be simple to allow arbitrary nesting in the residual program, just by unfolding calls to non-locally defined functions/procedures. This has been done in lambda-mix (Gomard and Jones 1991) and ML-mix (Birkedal and Welinder 1993). Some non-trivial interactions between nested scoping and higher-order functions have been studied by Malmkjær and Ørbæk (1995).

Array Dimensions. I doubt any existing partial evaluator can take a source program using (at most) two-dimensional arrays and produce a residual program using three-dimensional arrays. Nor is it easy to see how to do this in a natural fashion. A post-process may detect that access patterns corresponds to higher-dimensional arrays, but it is hard to see how "dimension raising" may be done as an integral part of the specialisation process.

Modules. When specialising a modular language, it should, by our principle, be desirable to create residual programs with more modules than the original program. Though we are not aware of such work, it should be possible to do so using techniques similar to polyvariant function specialisation.

Classes. In object oriented languages, it is natural to generate specialised instances of objects within a class. But our principle extends to creating entirely new class hierarchies in residual programs, which is less clear how to achieve.

Patterns. Languages with pattern matching usually allow arbitrarily complex patterns. If a partial evaluator can not generate patterns that are more complex than those found in the original program, this is an inherited limit. A solution to this problem is to combine several residual patterns into fewer, but larger patterns. It might be argued that a compiler will separate the pattern matching into nested tests anyway, so the limit may have no significance for the speed of residual programs. This argument is, however, implementation dependent.

Other. Similar problems may occur for features like exceptions, constraints, processes, streams etc. It is not clear to the author if moving to a polymorphic language will introduce a new potentially inherited limit.

7 Conclusion

What we have tried to impart to the reader is an awareness of the idea that a partial evaluator may cause residual programs to inherit limits from the original programs, and that this may be a problem.

The intention is to inspire developers of partial evaluators to consider if their partial evaluators exhibit such inherited limits, and decide if they think these important enough to deal with. In other words, the awareness of the problem may be used as a guideline for refining specialisation technology.

Identifying the problem is, however, only the first step. Knowing that a limit is inherited does not always suggest a solution. A good example of this is the array dimension limit, which has no obvious solution.

It may not be obvious which limits, if any, are inherited. A test for this may be done by attempting to write a self-interpreter and specialise this. If the residual programs are renamings of the original, it is evidence that no inherited limits exist. Otherwise, the problem may be in the interpreter or the specialiser, and the experiment may suggest which.

Note that we have only required that a self-interpreter exists which allows the residual programs to be identical to the source programs. Hence, we have not out-ruled the possibility of writing self-interpreters where partial evaluation would inject the residual programs in some sub-language, e.g. CPS style. This ability is important, and we should aim to keep this.

Not all improvements of partial evaluators are instances of removing inherited limits. Important subjects such as improved binding time analysis, control of termination, automatic binding time improvements, speed of specialisation, self-application etc. are all important developments that are orthogonal to the removal of inherited limits.

References

Berlin, A., Weise, D.: Compiling scientific code using partial evaluation. IEEE Computer vol. 23, no. 5, (1990), 25–37.

Birkedal, L. Welinder, M.: Partial evaluation of Standard ML. Masters Thesis, DIKU, University of Copenhagen, Denmark, (1993).

Bondorf, A.: Improving binding times without explicit CPS-conversion. LFP '92, Lisp Pointers, vol. V, no. 1, ACM Press (1992), 1–10.

Bondorf, A.: Similix manual, system version 5.0. Tech. Report, DIKU, University of Copenhagen, Denmark, (1993).

Bulyonkov, M. A.: A theoretical approach to polyvariant mixed computation. Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avernæs, Denmark, 18-24 October, 1987, North-Holland (1988), 51–64.

Danvy, O.: On the evolution of partial evaluators. DART technical report, University of Aarhus, Denmark (1993).

De Niel, A., Bevers, E., De Vlaminck, K.: Program bifurcation for a polymorphically typed functional language. PEPM '91, ACM Press (1991), 142–153.

Dussart, D., Bevers, E., De Vlaminck, K.: Polyvariant constructor specialisation. PEPM '95, ACM Press (1995), 54–65.

Gomard, C. K., Jones, N. D.: A partial evaluator for the untyped lambda-calculus. Journal of Functional Programming, vol. 1, no. 1, January 1991, 21–69.

Itkin, V.E.: On partial and mixed program execution. Program optimization and transformation, Novosibirsk Computing Center, (1983), 17–30.

Jones, N. D., Sestoft, P., Søndergaard, H.: An experiment in partial evaluation: the generation of a compiler generator. Rewriting Techniques and Applications, Springer LNCS 202, Springer-Verlag, (1985), 124–140.

Jones, N. D. (collector): Challenging problems in partial evaluation and mixed computation. Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avernæs, Denmark, 18-24 October, 1987, North-Holland (1988), 1–14.

Jones, N. D., Gomard, C. K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, 1993

Kildall, G. A.: A unified approach to global program optimization. POPL'73, ACM Press (1973), 194–206.

Malmkjær, K., Ørbæk, P.: Polyvariant specialisation for higher-order, block-structured languages. PEPM '95, ACM Press (1995), 66–76.

Mogensen, T. Æ.: Partially static structures in a self-applicable partial evaluator. Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avernæs, Denmark, 18-24 October, 1987, North-Holland (1988), 325–347.

Mogensen, T. Æ.: Separating Binding Times in Language Specifications. FPCA '89, Addison-Wesley (1989), 14–25.

Mogensen, T. Æ.: Self-applicable partial evaluation for pure lambda calculus. PEPM'92, Yale tech. report YALEU/DCS/RR-909 (1992), 116–121.

Mogensen, T. Æ.: Constructor specialization. PEPM '93, ACM Press (1993), 22–32.

Romanenko, S. A.: Arity raiser and its use in program specialization. ESOP '90, LNCS 432, Springer-Verlag (1990), 341–360.

This article was processed using the LaTeX macro package with LLNCS style