Code generation techniques for functional languages*

Jon Fairbairn and Stuart C. Wray
University of Cambridge Computer Laboratory,
United Kingdom

Abstract

This paper describes techniques used to generate fast, fully lazy code for the normal order functional language Ponder. The techniques include compile-time reduction, improved supercombinator abstraction, and strictness analysis.

The termination of compile-time reduction of expressions is ensured by use of the strong normalisation property of a simple type-system. The method of supercombinator abstraction is derived from Hughes' original algorithm, but produces larger combinators, which are therefore faster. A novel algorithm is used to analyse the strictness properties of the supercombinator programme. We present the disappointing result that a higher order strictness analyser gives no improvement in performance over a first order analyser that copes with curried functions.

The paper includes comparisons between the performance of programmes compiled with and without the various optimisations.

1 Introduction

This paper describes some of the techniques used to generate fast code for the lazy functional language Ponder [Fairbairn 83, Fairbairn 85, Tillotson 85]. The front end of the compiler performs type checking, and then converts the programme to lambda expressions; all of the optimisations described act on the lambda expression version of the programme. In the back end, super-combinators [Hughes 82] are produced from the lambda expressions, and stack machine intermediate code from the super-combinators.

The techniques described here are divided into four classes: super-combinator implementation techniques, compile-time reduction, optimisations in the super-combinator abstraction algorithm and strictness analysis.

*Work funded by the Science and Engineering Research Council of Great Britain

There are two ways to generate code for super-combinators: 'tree building' and 'direct code'. The method used in Ponder is to generate 'direct code', because this is a prerequisite to obtaining performance gains from strictness analysis. Both methods are described in the next section and their effects on performance are measured.

Following this is a description of the compile-time reduction pass, and then a section on super-combinator abstraction. Hughes' original algorithm is unnecessarily pessimistic, producing smaller combinators than needed, and the optimisations described in this section give larger combinators, and thus faster code.

The Ponder compiler uses a strictness analyser to annotate the super-combinator programme and thus produce better machine code. The penultimate section describes this and gives details of the resulting performance improvements. Of particular interest is the rather surprising result that higher order strictness analysis gives no improvement in performance over a first order analysis that deals with curried functions properly.

We conclude the paper with a comparison of the performance of programmes written in Ponder and in two conventional languages, LISP and BCPL. Appendix B gives the algorithm for generating intermediate code from super-combinators. This algorithm exploits the strictness annotations of the super-combinator programme in order to generate direct code whenever possible, and also avoids evaluating expressions repeatedly.

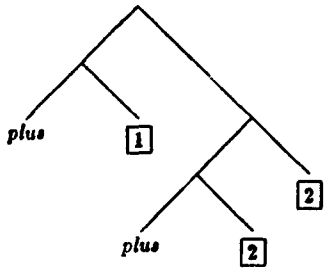
2 Implementing super-combinators

2.1 Tree Building

In tree building code the super-combinators are encoded as trees of applications which are then used as templates. Places are marked in these trees where arguments are to be substituted. When a super-combinator is applied to some arguments, a copy of this 'template tree' is made, with the arguments substituted into these places. Thus the combinator

$$F \triangleq \lambda x.\ \lambda y.\ (plus\ x\ (plus\ y\ y))$$

would be converted into the template



The value returned by the combinator is a copy of this tree with [1] and [2] replaced by the first and second arguments given to the combinator. This tree is then traversed by the evaluator, to perform any further reductions.

2.2 Direct Code

An alternative to copying a template tree would be to compile the body of the combinator into code for a stack machine that would build the tree node by node. This has the advantage that the arguments can be put into the tree directly, without the need for comparisons with numbers in the template. With this method F would be compiled into the code

```
Push_arg 2;
Push_arg 2;
Push Plus;
Fap 2;
Push_arg 1;
Push Plus;
Fap 2;
Return 2;
```

where the instruction Fap replaces the top two elements of the stack by a pointer to an application node containing them (this abstract machine code is described in Appendix A). The resulting tree would still need to be traversed by the evaluator to perform the remaining reductions.

An obvious optimisation of this approach is to avoid building a tree when the value can be calculated immediately. In the case of F we know that plus evaluates its arguments, so there is no need to build any trees at all. The code for F would now be

```
Push_arg 2;
Push_arg 2;
Call Plus;
Push_arg 1;
Call Plus;
Return 2;
```

The abstract machine code used in the Ponder system is quite similar to G-code used in the Lazy ML compiler [Johnsson 83].

An additional benefit of generating direct code is that conditionals can now be implemented much more efficiently. The tree building approach to conditionals is to make two trees, one for the Then part and one for the Else part. Depending on whether the condition evaluates to true or false, one or the other of these trees is discarded. This is a waste of time — not only is it futile building a tree and throwing it away, it is not even necessary to make a tree for the part that is needed. Code can be generated to produce the values returned by the Then and Else parts, just as it is for a function body, by direct execution of compiled code rather than interpretation of trees. The appropriate code fragment can be jumped to after testing the condition of the If.

Even when compiled into direct code, some combinators will still return application trees constructed so as to incorporate unevaluated arguments, and these trees will need further evaluation, but there is a worthwhile performance improvement over the tree template approach, as is shown by the timings in the table below.

Programme	Tree building	Direct	Gain
Nfib 20	30.00s	8.12s	269%
Tak 18 12 6	99.20s	32.47s	206%
Quicksort	4.70s	2.59s	81%
Dragon 10	100.72s	76.44s	32%
Lambda-parser	37.74s	20.12s	88%

As with all other performance figures given in the paper, the timings were taken on an 8MHz 68000. 'Gain' is calculated as

$$\frac{(\text{original time} - \text{new time})}{\text{new time}} \times 100.$$

Nfib is a standard performance test for functional languages, based on the Fibonacci series. A call of Nfib 20 executes 21 891 recursive calls. Tak is the "Takeushi" benchmark. Quicksort is the eponymous sorting algorithm used on a list of 100 random integers. Dragon 10 produces a picture of a fractal 'Dragon curve' [Mandelbrot 83] of depth 10 on a graphics terminal. This has 1024 line segments and involves a considerable amount of list processing, 'bit-shuffling' and arithmetic. Lambda-parser is a programme that parses a large lambda-expression, by using a parser made from higher-order functions. Nfib and Tak are rather atypical functions, but performance improvements for the other three benchmarks are similar to those for 'realistic' programmes. Note that these timings are for programmes compiled with the improved version of Hughes' algorithm and compile time reductions. Without these they are much slower.

3 Compile Time Reduction

For most programming languages there is an obvious benefit in reducing (evaluating) expressions at compile time. The same applies to functional languages in general, but in the case of Ponder it is essential because there are no built in control structures. All of these, including If and Case expressions, are built up using higher order functions or data structures and would involve a great overhead at run time if left alone.

For example the simultaneous declaration

$$\text{Let } a, b \triangleq c, d; e \quad \text{is equivalent to} \quad U\ (\lambda a.\ \lambda b.\ e)\ (P\ c\ d)$$

where $U\ f\ p \Rightarrow f\ (L\ p)\ (R\ p)$, $L\ (P\ a\ b) \Rightarrow a$ and $R\ (P\ a\ b) \Rightarrow b$. This can certainly be simplified to $(\lambda a.\ \lambda b.\ e)\ c\ d$, and possibly to $e[c/a][d/b]$ if this actually reduces the amount of work to be done at run time. This example is particularly significant since the strictness detection algorithm does not handle pairs. If e is strict on a or b, we will only detect this if the redundant pairing operation has been removed.

Hudak and Kranz have described [Hudak & Kranz 84] a compiler in which all reductions except recursive functions are performed at compile time. They discount the risk of non-terminating reduction sequences on the grounds that programmes rarely contain such things without the use of recursion. We are obliged to take a different view; a Ponder programme can validly include expressions such as $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$. Note that such programmes need not be non-terminating in themselves— the non termination might only occur for certain values of input data. Indeed, it is perfectly possible for a programme to include a new definition of the fixed point combinator, which would make it impossible to detect whether or not a given expression involved

recursion. We consider it desirable that the compiler should always terminate, even if particular programme does not, since it otherwise becomes impossible for a user to distinguish between a fault in the compiler and a fault in a programme.

Fortunately the type-system comes to our assistance. Although the Ponder type system is rich enough to permit a well typed expression for the fixed point combinator, it has a subset for which the strong normalisation theorem [Fortune, Leivant & O'Donnell 83] may be proved. The type system in Ponder only has constructors for functions, polymorphic types and for recursive types. Without the recursive type constructor the remaining subset of the type system is equivalent to a subset of the theory of constructions [Coquand & Huet 85], for which the strong normalisation theorem has already been proved.

If an expression can be typed in the simplified type system, we know that it has a terminating reduction sequence (we shall say that such an expression or its type "is SN"). Notice that it is not sufficient to check that the type of the expression belongs to this subset (the type of the fixed point combinator does!), nor is it sufficient to check that all the variables in the expression have appropriate types (if S and I have their usual meanings, then they can be well typed in the subset, but $(S\ I\ I)\ (S\ I\ I)$ can only be well typed in the full system). It is, however, sufficient to see that all subexpressions of the expression have types belonging to the subset. The front end of the ponder compiler annotates the lambda-expressions in the intermediate code to indicate whether they satisfy this condition.

If one wanted to apply this optimisation to a language that did not have a suitable type system, it would be possible to use a simple type inferencer based on Milner's [Milner 78] to infer some of the strong normalisation properties of expressions. This would not be quite as effective since some expressions that are well typed in the strongly normalisable Ponder subset cannot be typed in Milner's system. In particular Ponder allows the polymorphic use of bound variables to functions. A related observation is that even when the type given to an object by the user is not SN the object may still be. An example would be the list construction function *cons*, which will typically be given the type $\forall T.T \rightarrow \mathsf{List}[T] \rightarrow \mathsf{List}[T]$ involving the recursive type List, when it has the SN type $\forall T,U.T \rightarrow U \rightarrow \mathit{Option}[T \times U]$.

If an expression belongs to the strongly normalisable subset, we can reduce it at compile time without fear of non-termination. Unfortunately it is not a good idea to reduce all SN expressions, since this would lead to a loss of laziness. If, for example, an argument is an application as in $(\lambda x.\ F\ x\ x)\ (y\ z)$, it is possible that it will become a redex at run time, so that reducing the above expression to $F\ (y\ z)\ (y\ z)$ could result in whatever $(y\ z)$ becomes at run time being computed twice. This loss of laziness can be avoided by restricting compile time reduction to those cases where at least one of the following conditions holds:

1. The bound variable appears textually at most once in the body,
2. The argument is a single object i.e. a name, constant, or combinator,
3. The argument is an application of a primitive function to too few arguments each of which satisfies either (2) or (3), for example (int-plus-int 3). Such expressions contain no redexes, and so cannot result in a loss of laziness when copied.

Some non-SN expressions may be reduced. The application of an SN function to a non SN argument is safe provided that one of the above conditions is met, since the only risk of non-termination would come from applications of the argument, but these will not be reduced. Even applications of non-SN functions may be reduced if we are slightly more careful. The rules in this case are

1. The bound variable appears textually at most once in the body,
2. The argument is a single object i.e. a name or constant, but not a combinator,
3. The argument is an application of a primitive function to too few arguments each of which satisfies either (2) or (3)

The following table shows the effect of using the strong normalisation information to decide when to reduce expressions.

Programme	Without SN	SN	Gain
Nfib 20	2.97s	2.97s	0%
Tak 18 12 6	6.70s	6.70s	0%
Quicksort	2.56s	2.45s	4.5%
Dragon 10	70.92s	68.44s	3.5%
Lambda-parser	21.58s	18.58s	16%

The column headed "SN" gives the time for code taking advantage of all the optimisations. The column "Without SN" shows the slower times obtained by running programmes that have been compiled with the strong normalisation information ignored. Note that this does not mean that no compile-time reduction has been performed. The strong normalisation information makes the most difference for programmes that make some use of higher order functions. The effect is to expand the programme out into a version that has few higher order functions.

In the present version of the compiler only straightforward β reductions are performed, that is $(\lambda x.\ \ldots x \ldots x \ldots)\ X$ becomes $\ldots X \ldots X \ldots$. It would be possible to increase the number of reductions if we also allowed β reductions *around* abstractions. Suppose that $(\lambda x.\ \lambda y.\ \ldots x \ldots y \ldots)\ X\ Y$ cannot be reduced because X meets none of the conditions that would permit it. In the present system this would prevent the reduction involving Y even if it was safe. What we would like to do is convert the above expression to $(\lambda x.\ \ldots x \ldots Y \ldots)\ X$. We can see no objection to this in principle, but have not tried it yet.

4 The abstraction algorithm

The algorithm used to convert programmes to combinators is based on the supercombinator technique originated by John Hughes [Hughes 82]. This original algorithm produces unnecessarily many combinators, resulting in poor performance. This observation was made independently by Hudak and Goldberg [Hudak & Goldberg 85]. Their approach to the problem is quite similar to ours, but sufficiently different in detail that it is worthwhile describing our method here.

We require that the abstraction algorithm should preserve the laziness textually present in the source programme. What we want is that whenever a reducible instance of an expression is generated it is reduced at most once at run-time. If $F \triangleq \lambda x.\ \lambda y.\ x \times x + y \times y$, and we put $G \triangleq F\ 3$, G should be equivalent to $\lambda y.\ 3 \times 3 + y \times y$. However 3×3 is a redex, with value 9, so when G is applied for the first time, it should become equivalent to $\lambda y.\ 9 + y \times y$.

In Hughes' algorithm this is achieved by 'lifting' out the expression $x \times x$ and converting F to $\lambda x.\ F_1\ (x \times x)$ where $F_1 \triangleq \lambda xsq.\ \lambda y.\ xsq + y \times y$. Now G can be reduced to $F_1\ (3 \times 3)$, and then to $F_1\ 9$. Hughes' algorithm will convert F in this way irrespective of the uses of F. This is over cautious — if F is always given two arguments at the same time, so that F applied to one argument is never shared, there is no point in changing the definition of F. Indeed, such a change imposes the overhead of calling a second combinator, so it is advantageous not to do it.

Another way in which redundant combinators are introduced is when the expression lifted out cannot be a redex. There is no point in lifting out an expression such as *product* x when the function *product* needs two arguments before it can be reduced.

4.1 Hughes' Algorithm

Analogous to the concept of free variable, a free *expression* is an expression that contains no occurrences of bound variables. For instance, in $\lambda y.\ x \times x \times 3 + y$, x is a free expression, as are $x \times x$ and $x \times x \times 3$. A *maximal* free expression is a free expression that is not a subexpression of any larger free expression. In the above example the only maximal free expression is $x \times x \times 3$.

Hughes converts a lambda abstraction to combinators by lifting out the maximal free expressions and replacing them by variables. Hence $\lambda y.\ x \times x \times 3 + y$ becomes $(\lambda c.\ \lambda y.\ c + y)\ (x \times x \times 3)$. Now $(\lambda c.\ \lambda y.\ c + y)$ has no free variables, and is therefore a combinator.

The freeness of an expression is calculated by assigning 'levels' to variables. One can think of the level of a variable as indicating the order in which values can arrive: in $\lambda x.\ \lambda y.\ \lambda z.\ E$ the variable x has level one, y two and z three. When substituting into E, a value for x will arrive first, then one for y, the value for z arriving last. The level of an expression is defined as to be the greatest of the levels of all its free variables. If E is $x \times x + y \times y + z \times z$, then the level of $x \times x$ is one, of $y \times y$ is two, of $x \times x + y \times y$ is two and the level of E is three.

So the level of an expression can be thought of as the 'time' by which values for all of its variables will be available — if an expression has level one, all of its values will arrive when the first variable is instantiated, and an expression at level two must certainly wait for the second variable. Free expressions are then lifted out to a position where they can be instantiated as soon as possible.

4.2 Maximal Reducible Free Expressions

The first improvement to this algorithm arises from the observation that a maximal free expression is not necessarily reducible. The reason for lifting an expression out is so that its value may be preserved when it is reduced — if it cannot be reduced, then there is no point in lifting it out. Suppose that *product* is a combinator that takes two arguments. In $\lambda x.\ \lambda y.$ *product* $(x + x)\ y +$ *product* $y\ y$ the expression *product* $(x + x)$ is a maximal free expression of $\lambda y.\ \ldots$, but only $(x + x)$ can be reduced.

The optimisation is simply to take account of the number of arguments needed by combinators; when an expression is a candidate for lifting, see if it is a combinator applied to too few arguments. If so, just lift out the maximal reducible free expressions of the arguments. This means that the above expression will be abstracted into $\lambda x.F_1\ (x + x)$ where $F_1 \triangleq \lambda xx.\ \lambda y.$ *product* $xx\ y +$ *product* $y\ y$, rather than into into $\lambda x.F_2\ (\textit{product}\ (x + x))$ where $F_2 \triangleq \lambda px.\ \lambda y.\ px\ y +$ *product* $y\ y$. This has the advantage that *product* $xx\ y$ can be compiled into direct code, whereas $px\ y$ must be built as a tree.

4.3 Groups of arguments

The level numbering of Hughes' algorithm allows for the possibility of partial applications of curried functions. If $F \triangleq \lambda a.\ \lambda b.\ a \times a + b$, and we have $G \triangleq F\ 3$ and G is used more than once, we want to preserve the value of $a \times a = 3 \times 3 = 9$ to avoid recalculating it at each application of G. So F is compiled to $\lambda a.\ F'\ (a \times a)$ (where $F' \triangleq \lambda aa.\lambda b.aa + b$), so that $G = F'\ (3 \times 3) = F'\ 9$. In practice it is common for curried functions to be defined that are never applied to fewer than n arguments. If F was always applied to two arguments, there would be no need to split it into two combinators. Notice that this cannot be detected naïvely from the textual occurrences of the functions: even if F is always supplied with two arguments in the text of the programme its arguments may not arrive at the same time. Consider

$$F \triangleq \lambda a.\ \lambda b.\ \ldots;$$
$$G \triangleq \lambda a.\ \lambda b.\ \ldots F\ a\ b \ldots;$$
$$H \triangleq G\ 1;$$
$$\ldots H \ldots H \ldots$$

Here the only textual occurrence of F is given two arguments, but G is shared, so we ought to compile G for the case when one argument arrives first, which means that $F\ a$ should be lifted out, which will create an instance of F with only one argument.

As with Hudak and Goldberg, our solution to this problem attempts to detect the shared partial applications of each function. What we shall describe here is an algorithm that approximates this, and produces the result in a form that can immediately be used by the abstraction algorithm.

In Hughes' algorithm the level numbers of bound variables are calculated independently of their use. We want to assign levels so that groups of bound variables that are always instantiated together have the same level. We can do this by considering the levels of the arguments supplied at each instance of a function. If the instance is $F\ e_1\ \ldots\ e_n$ then the level numbers of consecutive bound variables of F can be the same whenever the levels of consecutive arguments $e_i\ \ldots\ e_j$ are the same (provided that all other instances agree with this). This is an understatement of the case: if a distant argument arrives before a closer one it can be counted as having arrived at the same time. So the rule is: for each instance $F\ e_1\ \ldots\ e_n$, install a change in level between the i^{th} and j^{th} bound variables of F for which *level* $e_i <$ *level* e_j. If the number of bound variables is greater than n, we must assume the worst about all the remaining bound variables and give them increasing level numbers.

To get this off the ground we must begin by installing level changes whenever a lambda-abstraction is bound to a name N — this takes into account the possibility that the application of N to no arguments is shared (as indeed it will be, since all names that occur just once will have been reduced out by the previous pass of the compiler).

As an example consider the sequence of definitions

$$E \triangleq \lambda e_1.\ \lambda e_2.\ \ldots$$
$$F \triangleq \lambda f_1.\ \lambda f_2.\ \ldots E\ f_2\ f_1 + E\ f_2\ 9$$
$$G \triangleq \lambda g_1.\ \lambda g_2.\ \ldots F\ g_1\ g_2 \ldots$$
$$G\ 1$$

Initially all the bound variables will be given level 1. Level changes are installed starting with the expression $G\ 1$. This

will force a change in level between the bound variables g_1 and g_2, so g_2 will have level 2. Now we look at the body of G and discover that there is a increase in level between the first and second arguments given to F, so the levels of f_1 and f_2 become 1 and 2 respectively. In the body of F both instances of E are given arguments in order of decreasing level, so no change is needed for E.

Recursive definitions

The algorithm so far described works from the main expression backwards. Although this would not work for recursive definitions it is a simple matter to extend it to cope. When a level change is inserted in the bound variables of a recursive function we can propagate the change into its body, and so on until no more changes are inserted. This process obviously terminates since no more changes can be inserted when all the bound variables have different levels.

Abstracting

Once the all the bound variables in a programme have been given levels it is quite a simple matter to abstract out the appropriate combinators. The algorithm is very similar to Hughes' original, except that his algorithm cannot deal with the case when several variables have the same level. The only modification needed is that whenever we want to abstract out a group of arguments at the same level we do them all at once.

4.4 A minor refinement

Occasionally a user defined expression will already be a combinator — this typically arises from the partial application of a function — but not a redex. In such circumstances it is helpful to η-expand the expression until it contains a redex. If a programme contained the definition $double \triangleq product\ 2$ better code can be generated if this is expanded to be $double \triangleq \lambda x.product\ 2\ x$.

4.5 Results

Programme	Ordinary Hughes'	With improvements	Gain
Nfib 20	2.97s	2.97s	0%
Tak 18 12 6	114.70s	6.70s	1612%
Quicksort	2.79s	2.45s	14%
Dragon 10	74.28s	68.44s	8.5%
Lambda-parser	25.28s	18.58s	36%

The most marked effect is on Tak, a strict function of three arguments. This is because without the 'sharing' optimisation the code has to allow for the possibility that Tak may be applied to one or two arguments.

4.6 Improvements

As remarked above, the sharing algorithm is pessimistic about possible sharings of expressions that are passed as arguments. If we examine the expression

$$(\lambda f.G\ (f\ 9\ 9)\ (f\ 8\ 8))\ (F\ x)$$

we can see that although F applied to one argument is shared, F applied to two arguments is never shared since f is always given two arguments. In this case it would be quite easy to change the algorithm so that expressions bound immediately by lambdas are treated in the same way as function definitions. The search for sharing could be taken still further — the sharing properties of F after the third argument depend on how G treats its first and second arguments. This information could be propagated backwards from the definition of G and through the lambda binding into F, but it is questionable as to whether the resulting improvement in performance would justify the extra effort.

5 Strictness analysis

5.1 The strictness analyser

In lazy functional languages it is often the case that an argument to a function will necessarily be evaluated by that function. It could thus have been evaluated before being passed to the function (*eagerly*) rather than after (*lazily*). The aim of strictness analysis is to find out where eager evaluation would have the same effect as lazy evaluation, and so allow a compiler to produce more efficient code. When an expression is passed eagerly, direct code can be compiled for that expression and the value it returns given to the function. When an expression is passed lazily, an application tree must be made, so that if the value of the expression is subsequently required this tree can be evaluated. Evaluating a tree is slower than executing direct code, and there is the additional overhead of creating the tree and garbage-collecting it later.

The first work on strictness analysis for functional languages was done by Mycroft [Mycroft 81]. This style of strictness analyser, using *abstract interpretation* has been extended to higher order functions [Hudak & Young 85, Burn et al 85] and more recently to lists [Kieburtz & Napierala 85, Wadler 85].

The strictness analyser used in the Ponder compiler is not based on Mycroft's work. It differs from Mycroft's analyser in several ways:

- Mycroft's original analyser only works on first order functions, so functions that take functions as arguments or return functions as results are forbidden. The Ponder strictness analyser allows both higher order functions and curried functions. Two versions of the analyser can be used in the Ponder compiler: "first order", which loses some strictness information when handling higher order functions, and "second order", which loses information only when the functional argument of a higher order function is itself a higher order function. No information is lost for curried functions in either case.

- As well as the classifications strict and lazy, the Ponder analyser also has *absent* (first suggested in [Hughes 85]) and *dangerous*. If the analyser finds that a function will definitely evaluate an argument, that argument is said to be strict. If the analyser finds that a function will definitely *not* evaluate an argument, that argument is said to be absent. The classification lazy is not the opposite of strict, but instead represents a position of uncertainty between strict and absent. A lazy argument might be evaluated, but the strictness analyser could not predict that it certainly would be. If the analyser finds that a function needs to know the value of an argument, but the function is found not to terminate, that argument is said to be dangerous. An argument that is absent or dangerous need not be evaluated, since it can never be used.

  Absence information could also be carried around at runtime and might give performance gains for both tree build-

ing and direct code. However, this is not done in the Ponder system. The evaluator would have to be changed considerably, since the strictness properties of a function would have to be determined by the evaluator before any trees were built for the function's arguments.

- The analyser in the Ponder compiler can be used safely on any super-combinator programme. Strictness analysers based on Mycroft's work will not terminate when called to analyse some kinds of self application.

The Ponder strictness analyser works as follows: A *mode* is assigned to every combinator in a programme, describing the manner in which that combinator uses its arguments (lazily, strictly and so on). There are four different ways in which a function can use an argument, and for each argument that a function takes there is a *usage* in the the mode of the function which specifies one of these ways. The four possible usages are S for strict, L for lazy, A for absent and D for dangerous. The syntax of a mode is:

$$mode = \begin{cases} usage \rightarrow mode \\ m_L \\ m_D \end{cases}$$

The syntax of a usage is

$$usage = \begin{cases} S \\ L \\ D \\ A \end{cases}$$

If the mode of a function ends in m_D or contains the usage D this means that it will not return a value — its execution will not terminate.

Built-in functions have predefined modes that are not changed by the strictness analyser. For example the built-in function to add two integers together has the mode

$$S \rightarrow S \rightarrow m_L$$

The two Ss indicate that the function is strict on both its arguments. The m_L after the second arrow indicates the the result does not have mode m_D, and thus may terminate. Other than this m_L gives no information about an object.

The analyser determines the mode of a combinator by recursively examining the sub-expressions of the combinator's body. For each sub-expression the analyser derives a mode and a *usage set*. The usage set of an expression is a set of associations between bound variables and their usages in that expression. For example, if a and b are bound variables and F is a function with mode $S \rightarrow L \rightarrow m_L$, then in the expression $(F\ a\ b)$, a has usage S and b has usage L. This is because a is the first argument to F, so its usage is the same as the first usage in the mode of F. Similarly b's usage is the same as the second usage in the mode of F. The usage set of $(F\ a\ b)$ is $\{a{:}S,\ b{:}L\}$.

To find the mode and usage set of an expression the analyser combines the modes and usage sets of its sub-expressions. To determine the mode of a combinator it is necessary to know the modes of all of the combinators mentioned in its body. For recursive definitions an approximation to the combinator's own mode must be used when analysing its body. The mode m_D is used for this purpose. By iterating to obtain better approximations the analysis converges to a suitable mode for each recursive combinator. A complete exposition of this new analyser, including the second order version, can be found in [Wray 86].

5.2 First order strictness

Strict arguments to super-combinators are evaluated directly in machine code rather than by using trees. The tables below summarise the improvement arising from the use of first order strictness analysis compared with direct code without strictness analysis.

Programme	Without strictness	With strictness	Gain
Nfib 20	8.12s	2.97s	173%
Tak 18 12 6	32.47s	6.70s	385%
Quicksort	2.59s	2.45s	5.5%
Dragon 10	76.44s	68.44s	12%
Lambda-parser	20.12s	18.58s	8%

The last three programmes involve much list manipulation, so the improvement for them is rather small. Nfib shows a considerable improvement — it runs at more than double the speed. This is the best improvement that can be hoped for, since the intermediate code for the strict version of Nfib is 'perfect' — as good as handwritten code for the recursive function. The effect on Tak is similarly impressive.

The improvement due to strictness analysis is worth having, but not as large as might have been hoped for. It is quite good 'value for money', since in the Ponder compiler the code to perform first order strictness analysis is about 300 lines, out of a total of 10000 and takes about 5% of the compilation time.

5.3 Second order strictness

The performance of the code produced using the second order strictness analyser is exactly the same as that produced using the first order analyser. To check this, several Ponder programmes were selected and the intermediate code produced using each analyser was compared. In about 10000 lines of intermediate code there were only ten differences, and only five of these would have had any noticeable effect on performance. In particular lambda-parser uses higher order functions extensively and there were no differences in all its 3000 lines of intermediate code.

The analyser is only "second order", so it might appear that a full higher order analyser could find significantly more strictness. However, this is unlikely, since the reason for the poor performance seems to be that higher order functions have a very bad effect on strictness analysis. Consider the application

map function list

where *map* is the standard map function, *function* is strict on its argument, and *list* is some list. *map* will apply *function* to each element of *list* and it will return a list of the results. A first order analyser will find that *map* is lazy on *function* and strict on *list* (although it will be lazy on the elements of *list*). With second or higher order analysis there will be no change, so there could be no performance improvement.

One way to produce an improvement would be to alter the implementation so that there were two versions of *map*, one expecting a strict function, the other expecting a lazy function. As it stands, *map* must be pessimistic and assume that its functional argument is a lazy function, because it sometimes will be. If there were two versions of *map*, the appropriate one could be selected after determining the strictness properties of *function*. Care would have to be taken that only a small number of alternative versions of higher order functions were made, otherwise there would be an explosion in the size of the programme.

For this scheme to yield significant results it would be necessary to use list strictness analysis as well. This is because in the body of *map* there is the expression

cons (*function* (*head list*)) (*map function* (*tail list*))

and (*function* (*head list*)) would still have to be constructed as an application tree if the standard lazy *cons* were used. With list strictness analysis [Wadler 85, for example] it would be possible to determine that (say) the whole output list of *map* would be required, so a strict version of *cons* could be used. However, to do this there would have to be even more versions of *map*, each with a different kind of *cons*, or else strictness information would have to be passed around at run-time to enable map to take appropriate action. We believe that the first alternative may lead to an unacceptable increase in code size, but the second scheme appears to be quite promising.

5.4 The "Applied" optimisation

When a function is found to be strict on an argument we can generate code for the function under the assumption that the argument will have been evaluated. In cases where the function is called directly this is forced to be the case, either because the argument can be compiled as direct code or by forcing evaluation with an 'Eval_top' instruction. When the function is built into a tree of applications we cannot guarantee that strict arguments are evaluated in advance, so functions are provided with a second entry point with 'Eval_arg' instructions for all the strict arguments.

The "applied" optimisation avoids generating 'Eval_arg' or 'Eval_top' instructions when they are not needed. If a strict argument is applied as a function, such an application must be built as a tree which is then evaluated — this means that the argument would be traversed by the evaluator twice, once before entry to the function, and again afterwards. This can be avoided by looking at the types of strict arguments. If an argument has a function type, then we can leave out the 'Eval_'s.

In practice this optimisation has little effect on performance — at most a gain of 2%, but frequently much less than this. This is because most of the cases in which this optimisation could come into play will not have evaluation instructions anyway, because the abstract machine code generation algorithm (Appendix B) leaves them out when the object is itself obviously a function.

6 Conclusions

The largest average increase in performance was obtained when going from tree building code to direct code — this is in any case a prerequisite if any performance gains are to be obtained from strictness analysis. After this, the other optimisations give a similar performance gain — often about 20%, but with a wide variation depending on the particular programme.

It is interesting to observe the interaction between the improvements to the generation of super combinators (including normalisation) and the speed up obtained from strictness analysis. The first table below gives timings for code with no improvements at the combinator level with and without strictness, and the second table compares the gain in this case with that when the improvements are on.

Programme	Without strictness	With strictness	Gain
Nfib 20	40.35s	34.31s	18%
Tak 18 12 6	186.23s	186.15s	0%
Quicksort	6.42s	5.81s	10%
Dragon 10	122.18s	112.62s	8.5%
Lambda-parser	54.30s	49.80s	9%

Programme	Gain without improvements	Gain with improvements
Nfib 20	18%	173%
Tak 18 12 6	0%	385%
Quicksort	10%	5.5%
Dragon 10	8.5%	12%
Lambda-parser	9%	8%

Higher order strictness analysis is gives no improvement over first order analysis for speeding up real programmes, unless other code generation techniques are also used. We have proposed two ways of using higher order strictness information: generating several copies of the code for some super-combinators and carrying strictness information at run-time, but neither of these has been implemented.

The techniques described in this paper have also been used in conjunction with those described by Stoye [Stoye et al 84] to generate micro-coded supercombinators for the SKIM-II graph reduction engine, with impressive performance gains [Elworthy 85].

For comparison the benchmark programmes have also been written in Cambridge LISP and BCPL. Their timings (also on an 8Mz 68000) are shown in the table below, with the percentage improvement over the best times obtained for Ponder.

Programme	Ponder	LISP	Gain	BCPL	Gain
Nfib 20	2.97s	2.96s	0.5%	1.52s	95%
Tak 18 12 6	6.70s	5.57s	20%	4.56s	47%
Quicksort	2.44s	0.55s	344%		
Dragon 10	68.08s	27.33s	149%	15.66s	335%

Quicksort was not written in BCPL because it needs to use a garbage collected heap to be a fair comparison with Ponder, and BCPL does not have a heap. Lambda parser was not written in BCPL or LISP because it makes extensive use of higher order functions, so it would be difficult to translate and any comparison of performance would be meaningless.

Nfib is about twice as fast in BCPL. Although the intermediate code for the Ponder Nfib is 'perfect', the translation from intermediate code into 68000 machine code is far from perfect. Techniques such as register slaving and peephole optimisation on the target code are not used in the Ponder compiler. The factor of two difference in speed indicates that all Ponder programmes could be made to run perhaps twice as fast by using better conventional code generation techniques. If we discount this factor of two and imagine that these techniques had been used when compiling Dragon in Ponder, the BCPL version would still be more than twice as fast. Presumably this difference is due to the use of lazy lists in Ponder. This explanation is certainly borne out by comparison with the LISP quicksort, in which the lists are of course strict.

References

[Burn et al 85]
G. L. Burn, C. L. Hankin & S. Abramsky,
The Theory and Practice of Strictness Analysis of Higher Order Functions,
Department of Computing,
Imperial College London,
Research Report Doc 85/6, April 1985.

[Coquand & Huet 85]
Thierry Coquand & Gérard Huet,
Constructions: A Higher Order Proof System for Mechanising Mathematics,
INRIA Rapport de Recherche Number 401,
1985. Also appears in Springer LNCS 203

[Elworthy 85]
David Elworthy,
Implementing a Ponder Cross-compiler for the SKIM processor,
Cambridge University Computer Laboratory,
Diploma Dissertation 1985.

[Fairbairn 83]
Jon Fairbairn,
Ponder and its Type System,
University of Cambridge Computer Laboratory,
Technical Report No. 31, 1983.

[Fairbairn 85]
Jon Fairbairn,
Design and Implementation of a Simple Typed Language Based on the Lambda-Calculus,
University of Cambridge Computer Laboratory,
Technical Report No. 75, May 1985.

[Fortune, Leivant & O'Donnell83]
Steven Fortune, Daniel Leivant & Michael O'Donnell,
The Expressiveness of Simple and Second-Order Type Structures,
JACM Vol 30 Number 1,
January 1983, pp 151–185

[Hudak & Goldberg 85]
Paul Hudak & Benjamin Goldberg,
Serial Combinators: "Optimal" Grains of Parallelism,
Yale University,
Department of Computer Science,
1985

[Hudak & Kranz 84]
Paul Hudak & David Kranz,
A combinator based compiler for a functional language,
11th ACM Symposium on Principles of Programming Languages,
ACM Jan 1984, pp 121–132

[Hudak & Young 85]
Paul Hudak & Jonathan Young,
A Set-Theoretic Characterization of Function Strictness in the Lambda Calculus,
Yale University,
Department of Computer Science,
Research Report YALEU/DCS/RR-391 (April 1985)

[Hughes 82]
John Hughes,
Graph Reduction with Super-Combinators,
Oxford University Programming Research Group,
Technical Monograph PRG-28 (1982)

[Hughes 85]
John Hughes,
Short talk at Workshop on Functional Programming,
Aspenäs, Sweden,
February 1985.

[Johnsson 83]
Thomas Johnsson,
The G-Machine: An Abstract Machine for Graph Reduction,
Proceedings of SERC Declarative Programming Workshop at UCL,
April 1983

[Kieburtz & Napierala 85]
Richard B. Kieburtz & Maria Napierala,
A studied laziness—strictness analysis with structured data types,
Oregon Graduate Centre,
Extended Abstract, July 1985.

[Mandelbrot 83]
Benoit B. Mandelbrot,
The fractal geometry of nature,
W. H. Freeman, New York 1983.

[Milner 78]
Robin Milner,
A Theory of Type Polymorphism in Programming,
Journal of Computer and System Sciences Vol 17 Number 3,
December 1978

[Mycroft 81]
Alan Mycroft,
Abstract Interpretation and Optimising Transformations for Applicative Programs,
PhD Thesis, University of Edinburgh, December 1981.

[Stoye et al 84]
W.R. Stoye, T.J.W. Clarke and A.C. Norman,
Some Practical Methods for Rapid Combinator Reduction,
1984 ACM Symposium on Lisp and Functional Programming,
ACM August 1984.

[Tillotson 85]
Mark Tillotson,
Introduction to the Functional Programming Language "Ponder",
University of Cambridge Computer Laboratory,
Technical Report No. 65, May 1985.

[Wadler 85]
Phil Wadler,
Strictness Analysis on Non-Flat Domains,
Programming Research Group, Oxford University,
November 1985.

[Wray 86]
S. C. Wray,
Implementation and Programming Techniques for Functional Languages,
Submitted as PhD Thesis, University of Cambridge, January 1986.

A The Ponder abstract machine

Ignoring output, the state of the machine can be considered as an element of Instruction-sequence × Stack × Dump × Heap
where Stack = List [Object], Object = Tag × Int, Dump = List [(Stack × Instruction-sequence)],
together with functions *nargs*, *code* and *eval-code* that give the number of arguments and the code for each combinator. The code returned by *code* c expects the strict arguments of the combinator c to have been evaluated already. The code returned by *eval-code* c does not expect its strict arguments to have been evaluated, and it has Eval_arg instructions to ensure that they get evaluated before entering the body of c.

In addition to the combinators there are selectors: U_n^w selects the n^{th} of its w arguments. Selectors are usually used to determine which part of a conditional to execute, but if they are applied as functions, *nargs* and *eval-code* give the following results:

nargs	$U_1^1 \triangleq$	1	*nargs*	$U_n^1 \triangleq$	2	*nargs*	$U_n^w \triangleq$	1
eval-code	$U_1^1 \triangleq$	Push_arg 1; Tail_eval 1	*eval-code*	$U_n^1 \triangleq$	Push_arg 1; Push U_{n-1}^1; Fap 1; Return 2	*eval-code*	$U_n^w \triangleq$	Push U_{n-1}^{w-1}; Return 1

So U_n^w selects the n^{th} of w arguments, but in order to preserve laziness it does this in stages.

The operations of the Ponder abstract machine are defined by means of productions of the form
<I_1, S_1, D_1, H_1>⇒<I_2, S_2, D_2, H_2>
where I_1 and I_2 are instruction sequences, S_1 and S_2 are stacks, D_1 and D_2 are dumps and H_1 and H_2 are heaps. When more than one production applies, the first takes priority.

The notation H[n⇝ p] means a heap H in which n indexes the element *p*. Integers which can be used as such indexes are tagged: Ap#n points to an application and Pa#n points to a pair. For operations on lists, :: denotes infix cons and ⧺ denotes infix append.

Push instructions

<Push x; I_{rest}, S, D, H>⇒<I_{rest}, (x :: S), D, H>
<Push_arg n; I_{rest}, S, ((a₁ :: ... :: aₙ :: ..., I) :: D), H>⇒<I_{rest}, (aₙ :: S), ((a₁ :: ... :: aₙ :: ..., I) :: D), H>

Form Application

<Fap 0; I_{rest}, S, D, H>⇒<I_{rest}, S, D, H>
<Fap (m+1); I_{rest}, (a :: b :: S), D, H>⇒<Fap m; I_{rest}, (Ap#n :: S), D, H[n⇝(a, b)]>

Call and Return

<Call c; I_{rest}, S, D, H>⇒<*code* c, (), ((S, I_{rest}) :: D), H>
<Return n; I_{rest}, (e :: S_1), ((a₁ :: ... :: aₙ :: S_2, I) :: D), H>⇒<I, (e :: S_2), D, H>
<Tail_call c, n; I_{rest}, S_1, ((a₁ :: ... :: aₙ :: S_2, I) :: D), H>⇒<*code* c, (), ((S_1 ⧺ S_2, I) :: D), H>

Selections

<Switch *n* (I_1 .. I_w ... I_n); I_{rest}, (U_n^w :: S), D, H>⇒<I_w; I_{rest}, S, D, H>

Evaluate

<Eval_arg n; I_{rest}, S, ((a₁ :: ... :: aₙ :: ..., I) :: D), H>⇒<Eval_top; !replace_arg n; I_{rest}, (aₙ :: S), ((a₁ :: ... :: aₙ :: ..., I) :: D), H>
<!replace_arg n; I_{rest}, (e :: S), ((a₁ :: ... :: a_{n-1} :: aₙ :: ..., I) :: D), H>⇒<I_{rest}, S, ((a₁ :: ... :: a_{n-1} :: e :: ..., I) :: D), H>
<Tail_eval n; I_{rest}, (e :: ()), ((a₁ :: ... :: aₙ :: S, I) :: D), H>⇒<Eval_top; I, (e :: S), D, H>
<Eval_top; I_{rest}, (e :: S), D, H>⇒<!eval, (e :: ()), ((S, I_{rest}) :: D), H>
<!eval, (Ap#n :: S), D, H[n⇝(f,a)]>⇒<!eval, (f :: Ap#n :: S), D, H[n⇝(f,a)]>
<!eval, (U_n^w :: ()), ((S, I) :: D), H>⇒<I, (U_n^w :: S), D, H>
<!eval, (U_1^1 :: Ap#n :: ()), D, H[n⇝ (U_1^1, e)]>⇒<!eval, (e :: ()), D, H[n⇝ (U_1^1, e)]>

<!eval, (c :: Ap#n_1 :: ... :: Ap#n_m :: S), D, H[n_1 ⇝ (f_1, a_1)]...[n_m ⇝ (f_m, a_m)] >⇒
<*eval-code* c, (), ((a₁:: ... a_m :: Ap#n_m :: S), !update) :: D), H[n_1 ⇝ (f_1, a_1)]...[n_m ⇝ (f_m, a_m)] >
where m = *nargs* c and c is a combinator or selector.
<!eval, (c :: Ap#n_1 :: ... :: Ap#n_m :: ()), ((S, I) :: D, H>⇒ < Ap#n_m :: S, D, H>
where m < *nargs* c and c is a combinator or selector.
<!eval, (e :: ()), ((S, I) :: D), H>⇒<I, (e :: S), D, H>

<!update, (e :: Ap#n :: S), D, H[n⇝p]>⇒<!eval, (e :: S), D, H[n⇝ (U_1^1, e)]>

B Code generation

The input

The input is a sequence of supercombinator definitions followed by an expression.

```
Programme ::=  Definition; Programme |
               Expression

Definition ::= C-name ≙ λarg1{Usage1}. ... λargn{Usagen}. Expression |
               A-name ≙ Expression

Expression ::= argn{Usage} |
               Constant |
               C-name |
               A-name |
               Expression1 {Usage} Expression2 |
               [selection n [Expression0] Expression1 ... Expressionn]

Usage ::=  S | L | D | A
```

Compiling code

For each supercombinator definition of the form ⟦ C-name ≙ λarg1{Usage1} ... λargn{Usagen} Expression ⟧ the compiler produces the definitions of *nargs* C-name, *code* C-name and *eval-code* C-name which will be used by the abstract machine. Supercombinators of the form ⟦ A-name ≙ Expression ⟧ are simply built as application trees on the heap that is given to the abstract machine along with the program

The function *arity* gives the arity of an expression. This information is derived from the original Ponder program. Note that *arity* c and *nargs* c are not necessarily equal. For instance if ⟦ c ≙ λx. int-plus-int x ⟧, then *arity* c = 2, but *nargs* c = 1. The function *dangerous* returns *true* if the mode of its argument, as determined by the strictness analyser was m_D — that is, if it definitely does not terminate.

The *nargs* definitions are produced first:

```
Nargs-Programme ⟦ Definition; Programme ⟧ =  Nargs-Definition ⟦Definition⟧;
                                             Nargs-Programme ⟦Programme⟧
Nargs-Programme ⟦ Expression ⟧ = <empty>
Nargs-Definition ⟦ C-name ≙ λarg1{Usage1}. ... λargn{Usagen}. Expression ⟧ = nargs C-name ≙ n
Nargs-Definition ⟦ A-name ≙ Expression ⟧ = nargs A-name ≙ 0
```

Then the definitions of *code* and *eval_code* are produced:

```
Compile-Programme ⟦ Definition; Programme ⟧ =  Compile-Definition ⟦Definition⟧;
                                               Compile-Programme ⟦Programme⟧
Compile-Programme ⟦ Expression ⟧ = Compile-strict ⟦ Expression ⟧ 0 (arity Expression)
Compile-Definition ⟦ C-name ≙ λarg1{Usage1}. ... λargn{Usagen}. Expression ⟧ =
      eval-code C-name ≙  If Usage1 = S Then Eval_arg 1; Fi
                          ...
                          If Usagen = S Then Eval_arg n; Fi
                          Compile-strict-expression ⟦Expression⟧ 0;
                          Return n
           code C-name ≙  Compile-strict-expression ⟦Expression⟧ 0;
                          Return n
Compile-Definition ⟦ A-name ≙ Expression ⟧ =  If dangerous Expression
                                              Then put the expression (*trap* *trap*) on the heap
                                              Else just build Expression as a tree on the heap
                                              Fi
Compile-strict-expression ⟦ Expression ⟧ m =  If dangerous Expression
                                              Then call *trap*
                                              Else Compile-strict ⟦ Expression ⟧ m (arity Expression)
                                              Fi
Compile-strict ⟦argn{S}⟧ 0 a = Push_arg n;
Compile-strict ⟦argn{L}⟧ 0 a =  Eval_arg n;
                                Push_arg n;
Compile-strict ⟦argn{A}⟧ 0 a = Push *trap*;
Compile-strict ⟦argn{D}⟧ 0 a = Push *trap*;
```

```
Compile-strict [arg_n{Usage}] m a =  Push_arg n;
                                     Fap m; If a = 0 Then Eval_top Fi
Compile-strict [Constant] 0 a = Push Constant;
Compile-strict [C-name] m a =  If nargs C-name ≤ m
                               Then  Call C-name;
                                     If (m − nargs C-name) > 0
                                     Then  Fap (m − nargs C-name);
                                           If a = 0 Then Eval_top Fi
                                     Fi
                               Else  Push C-name;
                                     Fap m
                               Fi
Compile-strict [A-name] m a =  Push A-name;
                               Fap m;
                               If a = 0 Then Eval_top Fi
Compile-strict [Expression_1 {Usage} Expression_2] m a =  If Usage = S
                                                          Then Compile-strict-expression [Expression_2] 0;
                                                          Elif Usage = D or Usage = A
                                                          Then Push *trap*
                                                          Else Compile-lazy-expression [Expression_2]
                                                          Fi
                                                          Compile-strict [Expression_1] (m+1) a;
Compile-strict [selection n [Expression_0] Expression_1 ... Expression_n] m a =
    Compile-strict-expression [Expression_0] 0;
    Switch n (Compile-strict-expression [Expression_1] m, ..., Compile-strict-expression [Expression_n] m);
Compile-lazy-expression [ Expression ] =  If dangerous Expression
                                          Then Push *trap*
                                          Else Compile-lazy [ Expression ]
                                          Fi
Compile-lazy [arg_n{Usage}] = Push_arg n;
Compile-lazy [constant] = Push constant;
Compile-lazy [C-name] = Push C-name;
Compile-lazy [A-name] = Push A-name;
Compile-lazy [Expression_1 {Usage} Expression_2] =  If Usage = A or Usage = D
                                                    Then Push *trap*;
                                                    Else  Compile-lazy Expression_2;
                                                          Compile-lazy Expression_1;
                                                          Fap 1
                                                    Fi
Compile-lazy [selection n [Expression_0] Expression_1 ... Expression_n] =  Compile-lazy [Expression_n];
                                                                           ...
                                                                           Compile-lazy [Expression_1];
                                                                           Compile-lazy [Expression_0];
                                                                           Fap n;
```

Peephole optimisation

After code generation, Faps are expanded out:

Fap (n+1) ⇒ Fap 1; Fap n

Tail_eval and Tail_call instructions may then be produced:

Eval_top; Return n ⇒ Tail_eval n

Call c; Return n ⇒ Tail_call c, n

Then Return, Tail_eval and Tail_to are migrated into Switches, and Faps are migrated out of Switches:

Switch n (I_1, ..., I_n); Return m ⇒ Switch n (I_1; Return m, ..., I_n; Return m)

Switch n (I_1, ..., I_n); Tail_call c, m ⇒ Switch n (I_1; Tail_call c, m, ..., I_n; Tail_call c, m)

Switch n (I_1, ..., I_n); Tail_eval m ⇒ Switch n (I_1; Tail_eval m, ..., I_n; Tail_eval m)

Switch n (I_1; Fap 1, ..., I_n; Fap 1) ⇒ Switch n (I_1, ..., I_n); Fap 1

Then the Faps are compacted again:

Fap n; Fap m ⇒ Fap (n+m)

Finally, redundant Eval_args are removed:

Switch n (I_1; Eval_arg m; I'_1, ..., I_n; Eval_arg m; I'_n) ⇒ Eval_arg m; Switch n (I_1; I'_1, ..., I_n; I'_n)

I_1; Eval_arg m, I_2; Eval_arg m; I_3; ⇒ I_1; Eval_arg m; I_2; I_3;