Partially strict non-recursive data types

ERIC NÖCKER AND SJAAK SMETSERS
Faculty of Mathematics and Computer Science, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
(*e-mail*: {eric,sjakie}@cs.kun.nl)

Abstract

Values belonging to lazy data types have the advantage that sub-components can be accessed without evaluating the values as a whole: unneeded components remain unevaluated. A disadvantage is that often a large amount of space and time is required to handle lazy data types properly. Many special constructor cells are needed to 'glue' the individual parts of a composite object together and to store it in the heap. We present a way of representing data in functional languages which makes these special constructor cells superfluous. In some cases, no heap at all is needed to store this data. To make this possible, we introduce a new kind of data type: *(partially) strict non-recursive data types*. The main advantage of these types is that an efficient call-by-value mechanism can be used to pass arguments. A restrictive subclass of (partially) strict non-recursive data types, *partially strict tuples*, is treated more comprehensively. We also give examples of important classes of applications. In particular, we show how partially strict tuples can be used to define very efficient input and output primitives. Measurements of applications written in Concurrent Clean which exploit partially strict tuples have shown that speedups of 2 to 3 times are reasonable. Moreover, much less heap space is required when partially strict tuples are used.

Capsule review

Ever since Augustsson and Johnsson first published their work on the G-Machine, it has been recognized that for good performance a functional language compiler must attempt to minimize heap access. In the G-machine this was achieved by using a value stack. Nöcker and Smetsers have shown how to extend and generalize Augustsson and Johnssons' work, and they report on the improvements to the execution time that the improvements give rise to.

In line with the Clean philosophy of functional programming, Nöcker and Smetsers have used annotations to mark those data structures that can be safely and usefully stored outside the heap. In the particular applications that they have considered (Parsing and the Fast Fourier Transform), they observe an approximately three-fold improvement in execution times.

1 Introduction

In pure functional languages, all expressions are referentially transparent. The value of an expression is not changed by evaluating that expression. As a consequence, there are (infinitely) many ways of denoting any given value. This fact is exploited by

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

lazy data types: types whose values may contain unevaluated components. Lazy data types allow infinite objects to be represented in a finite amount of space.

Functional languages handle (lazy) data types in a very elegant fashion, hiding all details of memory management or pointer manipulation. However, in comparison with strict data types, lazy data types have a serious disadvantage: a proper implementation consumes large amounts of both space and time. To store a possibly unevaluated expression, a complete representation including the environment needed to evaluate it, has to be created in memory. Although there exist compilation techniques which optimize memory management in many ways, the use of lazy objects remains an expensive affair. Both packing and unpacking lazy values requires significant execution time. Even worse, because of the dynamic memory behaviour a complex heap allocation mechanism, including a garbage collector, is necessary. If much memory is used, it is possible that more time is spent garbage collecting than executing the program itself. In the worst case more memory is needed than is available.

Strict data types have the advantage that the overhead introduced by (partially) unevaluated values disappears. Furthermore, when using strict basic types, such as strict integers or strict reals, we can avoid the use of any heap at all. Instead, values of these types can be kept on stacks or in registers, which significantly increases the efficiency of the code generated by the compiler.

Changing lazy data types into strict ones can be done by adding strictness information to a program. To a certain extent, this information can be derived automatically by strictness analysers. Implementations of such strictness analysers (e.g. Nöcker 1990) obtain fairly good results. Many functional programs have shown a remarkable speedup, for instance an efficient compilation of the well-known Fibonacci function leads to code that does not use any heap space. Even so, many functional programs still have large space and time requirements.

1.1 Partially strict data types

To increase the efficiency of functional programs, we believe that it is inevitable to appeal to the programmer. In this paper we present a new kind of data type for lazy functional languages which makes it easier for a programmer to specify efficient programs without losing the basic elegance of these languages. These new data types, called *partially strict data types*, are obtained by supplying types with additional strictness information. In a type definition this strictness information specifies which components of that type should be evaluated (the so called *evaluation context* of that component). In the type signature of a function the strictness information determines the evaluation contexts of the parameters and the result.

We show later that this strictness information enables the compiler to generate more efficient code, and we also present some important kinds of application that benefit from this new technique. In particular, we present a new, *efficient* way of handling IO in functional languages.

To discuss how partially strict data types can be implemented efficiently, we have chosen to employ the ABC-machine (Koopman *et al.*, 1990) as our abstract model.

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

This abstract machine is similar to G-machine variants (e.g. Johnsson, 1987; Peyton Jones and Salkild, 1989). The techniques presented in this paper are not restricted to the ABC-machine, however, but can be applied to any stack- or register-based (abstract) machine.

1.2 Structure of the paper

The next section briefly describes the abstract ABC-machine. Section 3 introduces partially strict data types and considers how they may be implemented. In section 4 we present some typical applications together with performance figures. Section 5 concludes. Though the techniques are developed and implemented in Concurrent Clean (Nöcker *et al.*, 1991), the program fragments in these sections are written in Miranda.[1]

2 The abstract ABC-machine

The abstract ABC-machine is a mixture of a graph-rewriting machine and a more traditional stack-based machine. Conceptually, evaluation of expressions is done using graph rewriting. A functional program is therefore represented as a set of graph rewriting rules. To increase efficiency these rewrite rules are not interpreted directly, but are instead compiled into ABC instructions. Expressions are represented by graphs and stored in the heap. Each node in the graph corresponds to either a function application or a value (the result of the evaluation of an application). In the latter case, we say that the graph that represents such a value is in *head normal form*. The ABC-machine has three stacks, of which two are used for argument passing (the third is used for storing code addresses, and is not relevant here): the A stack, containing addresses of nodes in the heap, and the B stack, containing values of basic types, such as integers or reals (see for example Fig. 1). Note that both stacks depicted

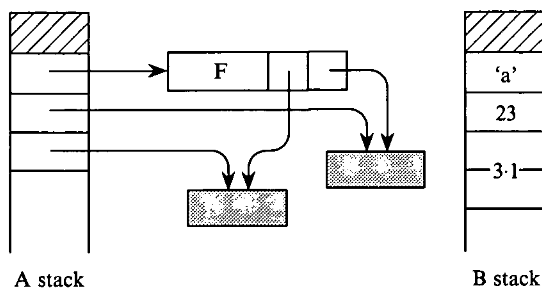

Fig. 1. A and B stacks of the ABC machine.

in Fig. 1 grow downwards. Shaded nodes indicate subgraphs whose precise structure is not relevant.

Basic values can be represented in two ways: as nodes in the heap, or as items on the B stack. In the latter case, a basic value may occupy more than one entry, for example, a 64-bit Real value needs two entries. As values are addressed relative to the

[1] Miranda is a registered trademark of Research Software Ltd. (Turner, 1985).

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

top of the stack, the sizes of all objects have to be known (preferably at compile time). To store a value in the heap, a node has to be allocated and filled with the value. Retrieving a value is relatively expensive because the node containing the value has to be unpacked. Clearly, storing values on the B stack is much more efficient. In an average ABC program, the efficiency increases by about a factor of eight when the B stack is used instead of the heap (Heerink, 1990).

Although this is not a complete description of the ABC machine, it should be sufficient to understand the optimizations below. In describing our implementation, we restrict ourselves to the use of the stacks and the heap.

Generating efficient machine code from an ABC program is not straightforward. At first sight, the use of stacks does not seem the most optimal way of using real machine resources (such as registers). However, it is possible to keep some of the top-most elements of the stacks in registers (which eliminates many push and pop operations). In the Concurrent Clean implementation this method is exploited as much as possible. The techniques presented in this paper lead to a more efficient use of the A and B stack. Consequently, the final machine code will also be more efficient. We will not consider machine code generation any further: this subject is treated more extensively in Smetsers *et al.* (1991).

The idea of having a special stack for processing basic values is nothing new, but we show here that this stack can also be used to keep (parts of) values of general composite data types. In composite data types (sometimes called algebraic data types) data constructors are defined that 'glue' the sub-components together. Generally, composite objects are represented directly in the heap. Typically, 50% of the nodes are used to represent the structure of the object rather than containing useful data. For example:

tree * ::= *Node* * (*tree* *)(*tree* *) |
Leaf *

A *tree* is normally represented by an object consisting of linked nodes with almost the same structure as a graphical representation.

Functions defined on complex data structures often use pattern matching to access the components of those structures. As a consequence, these functions may be strict in the corresponding argument. In such a case, it is sometimes possible to avoid building unnecessary graph structures. For example:

```
head :: [*]->*
head (f:r)         = f
head []            = error 'list was empty
fromto :: num  ->  num->[num]
fromto a b         = [],                     a > b
                   = a:(fromto (a+1) b),     otherwise
start :: num
start  =  head (fromto 2 10)
```

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

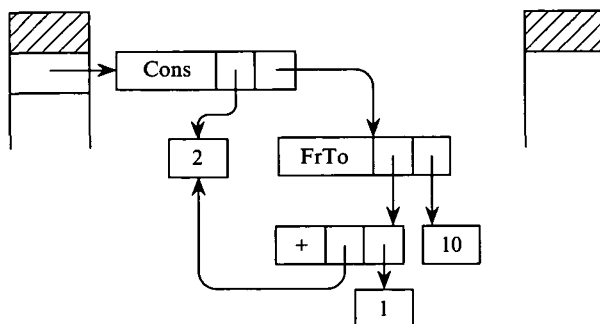

Fig. 2. The result of *fromto* passed to *head*.

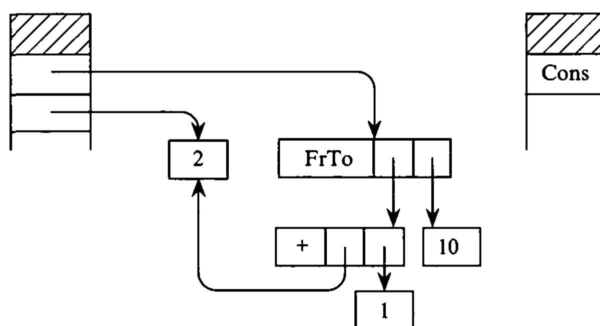

Fig. 3.

The function *head* uses pattern matching to access the head and the tail of the list. It is therefore strict in its argument. Usually, the result of *fromto* is passed to *head*, as shown in Fig. 2.

It is easy to see that the topmost node can be omitted. To distinguish between empty and non-empty lists, a code which identifies the constructor (called the *constructorid*) is pushed on the B stack (see Fig. 3). Obviously, for data types having only one constructor, such as tuples, a *constructorid* is not needed.

Although one node has been saved in this example, many nodes are still needed to represent the program structure.

3 Partially strict data types

How far a graph can be reduced at a certain moment depends on the evaluation contexts of the nodes in the graph. In lazy functional languages we can distinguish two kinds of contexts: lazy and head-normal form (hnf). Nodes in a hnf context can be reduced to head-normal form, whereas the lazy nodes should be left unevaluated. An example of a node which is always in a hnf context is the outermost node of the right-hand-side of a function definition.

The problem with data constructors is that the arguments of such constructor

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

nodes are always lazy, since the nodes containing the constructors are already in head-normal form. Often, however, the hnf context is too restrictive. For instance, when a function delivers more than one result these results might frequently be computed immediately. However, the constructor that is needed to glue these results together forces a lazy context for the arguments, which will postpone the evaluation of these arguments. For example:

$$complex \quad == \quad (real, real)$$

$$plusC \quad :: \quad complex \rightarrow complex \rightarrow complex$$

$$plusC\,(r1, i1)\,(r2, i2) \quad = \quad (r1+r2, i1+i2)$$

It is intended that the two parts of the result of *plusC* should be reduced immediately. However, the tuple constructor that is used to pack the real and imaginary parts of the complex number forces a lazy context for its arguments.

To avoid this, we introduce a new, more general notion of context that is defined by means of special data types.

3.1 Definition

A *partially strict data type* is a data type where it is specified (e.g. by the programmer) which parts of instances of that type should be evaluated or not. Expressed in terms of contexts, a partially strict data type is a data type for which the context of all the nodes of each possible instance is indicated explicitly. The default context is lazy. This default can be overruled by a strictness annotation which makes the indicated part partially strict. We use exclamation marks to denote strictness annotations. For example

$$some_tuple \quad == \quad (!(!num, bool), !num)$$

$$\begin{aligned} token \quad ::= \quad & Keyword \; !keywordkind \; | \\ & Identifier \; !entry \; | \\ & IdentName \; !char \; | \\ & Eof \end{aligned}$$

$$f \quad :: \quad num \rightarrow (bool, num) \rightarrow some_tuple$$

$$f\,a\,(b, c) \quad = \quad ((a, b), c).$$

An object of type *some_tuple* appearing in a strict context consists of a tuple of two elements, both of which are evaluated to at least head-normal form. The first element of this tuple is a tuple of two elements, of which the first one is a *num* value (evaluated) and the second one is a *bool* expression (possibly unevaluated). The second argument of the outermost tuple is an evaluated *num* value. An object of the (algebraic) type *token* can have various appearances, depending on the constructor. Note that, for instances of both types, graphs are built only if they occur in a lazy context.

Although it is possible to provide strictness annotations for non-strict positions, we

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

only allow arguments of partially strict types to be indicated as partially strict. This is because evaluation of a partially strict object in a non-strict context will force the evaluation of the surrounding non-strict object (for example, if we omit the first annotation in the definition of *some_tuple* then the first, strict numeral can only be evaluated if the tuple itself is present). So, this limitation does not imply any conceptual restrictions.

Annotations are allowed in type definitions as well as in type specifications of functions (see the next example). Since the right-hand-side of a function always appears in a strict context, we do not annotate the result type of the function explicitly. The strictness properties of the type *some_tuple* have a consequence for the function f: it becomes strict in both arguments; in fact, f is also strict in the second sub-argument c.

The rest of this section discusses the implementation of partially strict data types.

3.2. Implementation

The A and B stacks are used for passing parameters and returning results. The type of a function, including the strictness information, fully defines its calling convention. This also holds if one of the arguments, or the result, has a type that is partially strict. Consider, for example, the type of the following function (which is identical to the function given in the previous example, but with expanded type synonyms and full strictness information):

$$f \quad :: \quad !num \rightarrow !(bool, !num) \rightarrow (!(!num, bool), !num)$$

The first argument of f, which is strict, is expected on the B stack. The second argument of f is a partially strict tuple, of which the first argument, the *bool*, is passed via the A stack and the second argument, the *num*, via the B stack. The boolean might be unevaluated. The caller has to ensure that the layouts of both A and B stack are correct. The following picture shows the stack layouts of an example call to f (see Fig. 4).

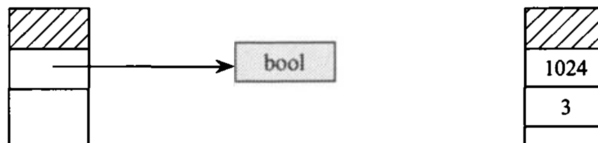

Fig. 4.

The result of f is handled in the same way. f returns a tuple consisting of a strict tuple and an evaluated *num* value. The latter is passed via the B stack. The innermost tuple contains an evaluated *num* and a possibly unevaluated *bool*. Consequently, the first argument of this tuple is placed on the B stack and the second one on the A stack.

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

Things become more complicated if algebraic types are involved. Consider the following example:

token	::=	*SpecialKeyword*	!*keyword* |
		Identifier	!*entry* |
		IdentName	!*char* |
		Eof	
keyword	::=	*OpenSyn* |	
		CloseSym |	
		DotSym |	
		LambdaSym	

env == (*File*, *IdentTable*)

getnexttoken :: ! *env* → (!*token*, !*env*)

parseexp :: !(!*token*, !*env*) → (!*syntaxtree*, !*env*)

parseexp (*Keyword OpenSym*, *e*) = ...

parseexp (*Keyword LambdaSym*, *e*) = ...

parseexp ...

The size of an object of type *token* in a strict context depends on the object's constructor. Hence, the corresponding *constructorid*, which is passed via the B stack, defines the layout of the rest of the stacks. The use of these strict constructors ensures that the function *getnexttoken* can return its result in an optimal way. Figure 5 shows a snapshot of both A and B stack just before *getnexttoken* delivers its result with the use of lazy types only (Fig. 5a) and when using partially strict types (Fig. 5b).

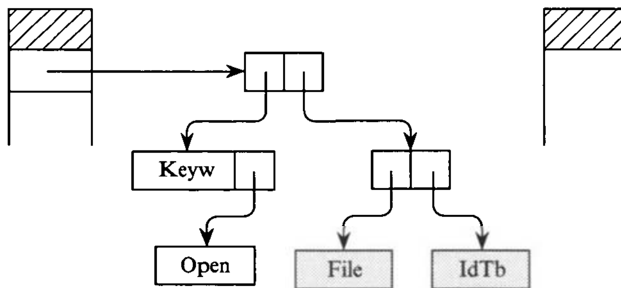

Fig. 5a

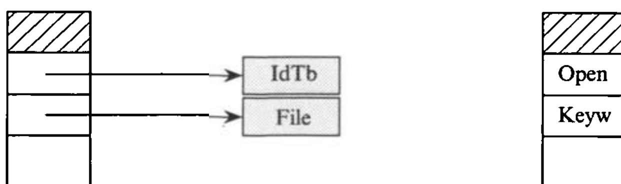

Fig. 5b

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

The function *parseexp* can use the result of *getnexttoken* immediately. However, it first has to analyse what instance of a token is actually present. A few switch statements select the code for the right alternative:

```
parseexp.1:
            switch 0 4 Keyword Identifier IdentName Eof
Keyword:
            switch 1 4 OpenSym CloseSym DotSym LambdaSym
OpenSym:
    ...     || code for the first alternative
CloseSym:
    ...
DotSym:
    ...
LambdaSym:
    ...     || code for the second alternative
Identifier:
    ...
```

3.3. Coercions

The layout of the stacks when a function is called is determined by the type of the function. When the actual layout of the stacks differs from the layout expected at the strict entry for a function, a conflict occurs. An example of such a conflict is:

```
f     ::  ([char], (num, [char]))
      ...
g     ::  !([char], !(!num, ![char])) -> num
      ...
start ::  num
start =   g f
```

Consider the application (*g f*). The way *f* delivers its result disagrees with the way *g* demands its parameter, as can be seen in Fig. 6.

In such a case, code has to be generated that converts the result into the form indicated by the type of the argument. We call such a conversion a *coercion*.

The most common coercion in lazy functional languages arises when an unevaluated expression appears in a strict context. The node representing the

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

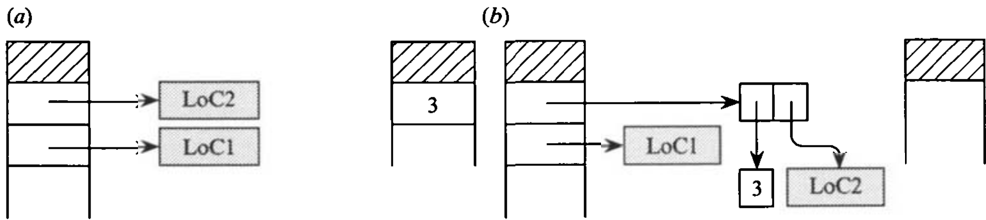

Fig. 6. (a) Demanded stacks. (b) Offered stacks.

unevaluated expression comprises an application of some function *f*. The arguments to *f* are also stored in the heap and must therefore be unpacked in such a way that they agree with the calling conventions of *f*. Similarly, when a node is updated with a result which has been divided over the stacks, these value have to be packed into a suitable form.

To give an idea of how much code may be involved in performing a coercion, we give the ABC code for the previous example:

```
start.1:
  jsr f             || evaluate f
                    || and now the coercion code
  push_a 1          || evaluate the second argument of the
  jsr_eval          || outermost tuple (which is a tuple itself)
  pop_a 1
  push_args 1 2 2   || push the arguments of the innermost tuple
  push_a 1          || on the A stack and reduce the second one
  jsr_eval          || (i.e. the second [char])
  pop_a 1
  jsr_eval          || evaluate the first argument of the second
  push_A 0          || tuple (i.e. the num) and push it on the B stack
  update_a 1 3      || update the A stack by overwriting the
                    || entry referring to the tuple with a
                    || reference to the second [char] and pop
  pop_a 2           || the superfluous entries from the A stack
                    || finally
  jmp g             || call g.
```

In some cases, large pieces of code may be necessary to perform the coercions. However, this does not necessarily increase execution time. In most cases, a coercion is done because the value of an object will be needed (in the case of unpacking), or was used (in the case of packing). But then the packed or unpacked value will be needed anyway, only the exact moment of packing or unpacking differs in the two

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

cases. Superfluous coercions may, however, occur when calling certain kinds of polymorphic functions. An example of this is the identity function applied to a strict tuple. This problem might be solved by generating code for all different kinds of applications of such functions. However, since such applications occur rarely, this optimization is not recommended.

3.4 On recursive data types

Unfortunately, this method of argument passing will not work for recursively defined data types. There are two reasons for this.

In contrast to non-recursive data-types, it is generally not possible to determine the size of instances of recursive data types at compile-time. However, to determine the positions of all the arguments or sub-arguments on the stacks it is necessary to know at compile-time how large each object will be. For example:

$$sum \;::\; ![num] \rightarrow !num \rightarrow num$$
$$sum\,[\,]\quad l = l$$
$$sum\,(a\!:\!r)\;l \;=\; sum\;r\;(l+a).$$

Suppose that [!*num*] indicates a fully evaluated list, of which the elements are passed via the B stack. To locate the second parameter, we need to know how many elements the list contains (see Fig. 7). A solution for this problem might be to place the actual

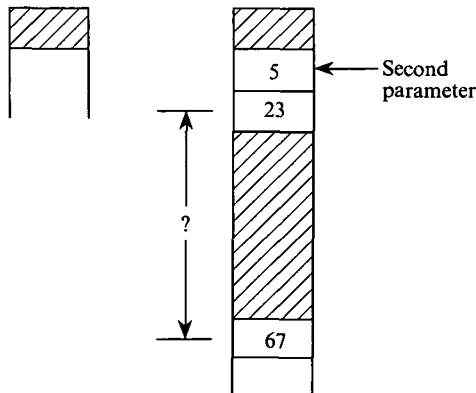

Fig. 7.

length of the list as an additional item on the B stack. But this is not the only problem. When a parameter is passed from one function to another the parameter may need to be copied, for instance when the called function expects it in a different position on the stack. This is particularly inconvenient when the parameter consists of a large number of stack entries. In some cases, especially when dealing with strict recursive data types whose size is unpredictable, this might lead to an unacceptable loss of

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

efficiency. Although such data types might be useful, they cannot be implemented efficiently using this technique.

To allow the compiler to choose between either a call-by-value or a call-by-reference mechanism (where the corresponding object is stored in the heap), we demand that the size of each object has to be known at compile time. Consequently, we restrict our attention to partially strict non-recursive data-types.

One further aspect of lazy recursive data types needs to be mentioned here. Because of their recursive structure, these objects are generally built dynamically using recursive functions. To be more specific, when a function builds a recursive object, it usually creates the recursive parts by means of one or more (possibly indirect) calls to the same function. Since these parts are glued together with a constructor, these recursive calls appear in a lazy context, and hence they have to be built in the heap instead of being evaluated immediately. Evaluating such an expression at a later stage involves unpacking the object, followed by a call to the evaluation code of the outermost function. However, if the evaluation code of the function no longer creates the root node of the result (as proposed in section 2) then, to preserve possible sharing, this has to be done after the evaluation has taken place. But then there is no gain over the old implementation in which the left-hand-side root node is overwritten by the result of the right-hand-side. Let us illustrate this with an example:

$$\begin{aligned} \mathit{fromto} &:: \; !\mathit{num} \rightarrow !\mathit{num} \rightarrow [\mathit{num}] \\ \mathit{fromto}\ a\ b &= \; [\,], (a > b) \\ &= \; a : (\mathit{fromto}\ (a+1)\ b). \end{aligned}$$

As argued earlier, a node containing the *fromto* application has to be built before evaluation. After evaluation this node will contain the root node of the rest of the list (either a *Cons* or a *Nil* node). But this is exactly the node that we intended to avoid building. Because of the lazy (indirect) recursive call of *fromto*, this node has been created in advance.

In addition, updating the node is hampered by the fact that it is necessary to examine the B stack in order to determine the constructor and arguments which should overwrite the node.

4 Applications and performance

In this section we discuss three examples in which partially strict (non-recursive) data types play an important role. They represent problems that cannot be implemented efficiently in standard implementations of lazy functional languages. Our main observation is that, thanks to the use of partially strict data types, both less execution time and less heap space are consumed. In practice, it would be almost impossible to derive this additional strictness information using only static analysis, so partially strict data types are clearly beneficial here.

The first example, the Fast Fourier Transform, is a well-known algorithm. Much computational work with complex numbers is involved. A complex number is defined

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

as a (strict) tuple of two real numbers. In the second example we present a new way of implementing IO in functional languages. The last example is a simple scanner/parser for lambda expressions, in which the new IO primitives are used. By using partially strict data types characters and tokens can be processed in an optimal way.

In the following sections only those program fragments are presented that are important for the discussion. The complete programs can be found in the Appendix. As mentioned earlier, the actual programs have been written in Concurrent Clean, and compiled by the Concurrent Clean system. The measurements have been done on a Macintosh IIfx, with a MC68030 processor running at 40 MHz. All timings are given in seconds of total execution time. Unless mentioned otherwise, all programs were executed with a 3MByte heap. The most recent code generator has been used for the compilation. This includes a smart register allocation mechanism (Smetsers *et al.*, 1991), and a very efficient heap management system (Groningen *et al.*, 1991).

4.1 The Fast Fourier Transform

The Fast Fourier Transform is a well-known algorithm for computing the discrete Fourier Transform of an array of complex numbers (e.g. Cooley and Tukey 1965). Consider an array $A[0 \ldots n-1]$ of n complex numbers. The Fourier Transform of A is an array $B[0 \ldots n-1]$, defined as

$$B[i] \quad = \quad \sum_{j=0}^{n} A[j] * r^{i*j}$$

where r is the nth principal root of 1. A straightforward implementation of this formula results in an algorithm with complexity $O(n^2)$. With the Fast Fourier Transform, array B is computed by first splitting array A into two parts, then deriving the Fourier Transforms of each part, and finally by merging the resulting arrays into a single array. The complexity of this algorithm is $O(n * \log(n))$.

The basic operations on complex numbers can easily be defined:

```
complex  ==  (!num, !num)

plusC  ::  !complex -> !complex -> complex
plusC (r1, i1) (r2, i2)  =  (r1 + r2, i1 + i2)

minC  ::  !complex -> !complex -> complex
minC (r1, i1) (r2, i2)  =  (r1 - r2, i1 - i2)

mulC  ::  !complex -> !complex -> complex
mulC (r1, i1) (r2, i2)  =  (r1 * r2 - i1 * i2,
                            r1 * i2 + i1 * r2)
```

Without strictness annotations all values would be lazy. However, because of the annotations in the type *complex* everything is strict and efficient code can be

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

generated. The strictness annotations on the arguments in these functions can also be derived by a strictness analyser. However, the type specifications are essential, since this is the only way to indicate that these functions return tuples that are strict in their arguments.

The actual work in the Fast Fourier Algorithm is done during the merging. The function *merge* merges two (converted) lists

```
merge  ::  ![complex] -> ![complex] -> !num -> [complex]
merge  even odd n
       =   low ++ high
           where
           (low, high)  =  merge' even odd 0
       ||  merge'  ::  ![complex] -> ![complex] -> !num -> (![complex], ![complex])
           merge' (e:re) (o:ro) i  =  (!ui : urest, !umi : umrest)
                                      where
                                      (urest, umrest)  =  !merge' re ro (i+1)
                                      ri    =  root i n
                                      prod  =  mulC ri o
                                      ui    =  plusC e prod
                                      umi   =  minC e prod
           merge' []     []     i  = ([], [])
```

The required values *ui* and *umi* of the new lists appear in a lazy context. This would normally mean that the whole expression containing the complex operations would have to be built in the heap. However, it is better to compute these values immediately. This is done by adding strictness annotations to force a strict context for the evaluation. Without these annotations, much more time and heap space is required. In practice, it is almost impossible to derive this information using current static analyses, though all elements of the list will certainly be used.

Table 1 shows timings for the Fast Fourier program. First, we note that due to the use of strict types the total execution time in the case of an array of 2^{11} elements is decreased by a factor of almost three. This is caused both by more optimal code and by lower heap use. The smallest amount of heap needed for the lazy version is around 2 Mbyte, whereas the strict version can easily run in 750 Kbyte. With lazy types the heap use becomes so large that it is impossible to run the program on an array of 2^{13} elements with a 3 MByte heap. The third row gives the execution time of a fairly optimal C version of the Fast Fourier algorithm. It is twice as fast as the Clean program. This can be completely attributed to the comparative heap behaviour of the two programs: the C solution uses just 600K of memory, and no garbage collections are required.

With respect to the efficiency of the Fast Fourier program the following notes are important:

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

Table 1. *Timings for the Fast Fourier Transforms*

	Execution time	Garbage collection time	Total time
Clean, strict types, array of 8K	5·5	4·9	10·4
Clean, lazy types, array of 8K	—	—	—
C, array of 8K	5·4	—	5·4
Clean, strict types, array of 2K	1·3	0·1	1·4
Clean, lazy types, array of 2K	3·0	0·8	3·8

- it seems as if the algorithm can be made faster by removing the call to *append* in the function *merge*. This can be done by defining several instances of the *merge′* function. In practice, this transformation appeared to have no effect.
- the algorithm written in an imperative language can be made faster by removing the splitting. This is done by performing a complex shuffle operation on the original array. Because lists elements cannot be accessed in constant time, this shuffling is very expensive in a functional language without constant access-time arrays.
- the root values that are needed can be computed in advance. This would reduce the number of computations. On the other hand, selecting the required root from the list means an expensive selection. So, this optimization is only useful if arrays are available.
- in all other respects, arrays do not improve efficiency. All other selections on lists concern the first element, which can be obtained by pattern matching. Also, the creation of new lists cannot be done more efficiently with arrays.

4.2 Fast IO in functional languages

One of the main disadvantages of lazy functional languages is their poor support for IO. For example, input is usually obtained via a lazy list of characters. Clearly, this repeated packing and unpacking of characters is not a very efficient way of passing input to the program. This inefficiency can be partly eliminated by supplying higher level functions that deliver integers, strings, etc., instead of characters. The effect of these functions is to divide the input into larger pieces before it is packed and passed to the program. However, such functions can only be used in particular cases. It would be better to have efficient low level functions with which the programmer himself can define these high level functions.

Our solution is based on partially strict tuples: we define a basic function *readchar* that returns a character, as well as a kind of continuation (called a *file*) that can be used for obtaining the rest of the input. *readchar* gets a *file* as input

$$readchar \quad :: \quad file \rightarrow (!char, !file)$$

Of course, a *file* might be implemented as a list of characters, or as a continuation function. However, that would again be very inefficient. A better solution is to use a more direct representation of a file, for example an index in the file table maintained

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

by the run-time system. To be able to distinguish between the various representatives of one physical file, we have to add a kind of version number to this representation. The easiest way is to use an integer that indicates the current position in the file. Thus, a *file* consists of a file position and a file index both represented by integers. We assume the following internal representation for files

$$
\begin{aligned}
\mathit{file} &== (!\mathit{file_position}, !\mathit{file_index})\\
\mathit{file_position} &== \mathit{num}\\
\mathit{file_index} &== \mathit{num}
\end{aligned}
$$

Clearly, both parts of the *file* value can be passed via the B-stack.

A problem with IO in functional languages is that many instances of one file can exist. Files used for reading cause no problems. The file pointer of the physical file can safely be adjusted if the current file position is checked on each access. For files used for writing, things are worse since if parts of a file were overwritten this would violate the property of referential transparency. We therefore only allow writes to the copy of the file with the highest version number. Consequently, a run-time error message will be given if an attempt is made to write to an old version of a file.

This kind of IO has been implemented in the Concurrent Clear System (Plasmeijer *et al.*, 1991). An example that shows the efficiency of this method is a program that copies a file to another file (character by character)

```
copyfile :: !file -> !file -> file
copyfile from to      = to, if endoffile from
                      = copy' (readchar from) to

copy' :: !(!char, !file) -> !file -> file
copy' (c, from) to   = copyfile from (putchar c to)

main = copyfile (openfile "in" "r") (openfile "out" "w")
```

All strictness annotations in the above definitions are derived by the strictness analyser. It takes 67 seconds to copy a file of 1 MByte. An equivalent C program is only 25% faster. This difference is due to the overhead of testing the file position markers in the Clean version. If strings are written instead of characters, however, the overhead becomes much less significant. In the current IO system of Concurrent Clean (Achten *et al.*, 1992) this testing is not needed anymore.

4.3 A simple scanner/parser

This example describes a simple scanner and parser for lambda expressions. With strict tuples, functions can accept and return their values efficiently. This allows faster scanners and parsers to be written. The function *readchar* is used for defining a function that determines the net token of an input file

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

```
token    ::=    SpecialKeyword    !keyword   |
                Identifier        !entry     |
                IdentName         !char      |
                Eof

keyword  ::=    OpenSym                      |
                CloseSym                     |
                DotSym                       |
                LambdaSym

  env           ==   (!file, !identtable)
  entry         ==   num

  getnexttoken   ::  !env -> (!token, !env)
  getnexttoken (f, t)  =  (token, (file, table))
                          where
                          (c, file)        =  skiplayout (readchar f)
                          token'           =  convertchartotoken c
                          (token, table)   =  putinidenttable (token', t)

  skiplayout   ::  !file -> (!char, !file)
  skiplayout f     =   ('EofChar', f)       , if endoffile f
                   =   skip (readchar f),   otherwise

  skip   ::  !(!char, !file) -> (!char, !file)
  skip (' ', f)    =  skiplayout (readchar f)
  skip ('\t', f)   =  skiplayout (readchar f)
  skip ('\n', f)   =  skiplayout (readchar f)
  skip x           =  x

  convertchartotoken   ::  !char   ->  token
  convertchartotoken '('           =   SpecialKeyword OpenSym
  convertchartotoken ')'           =   SpecialKeyword CloseSym
  convertchartotoken '.'           =   SpecialKeyword DotSym
  convertchartotoken '\\'          =   SpecialKeyword LambdaSym
  convertchartotoken 'EofChar'     =   Eof
  convertchartotoken c             =   IdentName c
```

All functions use strict types. The annotations in the type definition of *env* and in the right-hand side of the type specifications are added by hand. No nodes need be built for any of the input characters: they are all passed via the B stack. As soon as these characters are processed, they can be removed. Both execution speed and memory usage are improved by this technique.

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

Similarly, the tokens produced by the scanner can be passed efficiently to the parser. Consider, for example, the first alternative of the function *parseexp*:

```
parseexp  :: !(!token, !env) -> (!syntaxtree, !env)
parseexp(SpecialKeyword OpenSym, e)
=  (Application exp1 exp2, new_env)
where
(exp1, env1)     =  parseexp (getnexttoken e)
(exp2, env2)     =  parseexp (getnexttoken env1)
new_env          =  expectclosesymbol (getnexttoken env2)
parseexp ...
```

All calls to *getnexttoken* appear in a strict context (note that *new_env* and therefore also the other environments appear in a strict context because of the strictness annotations in the type specifications of the corresponding functions). So, the results of these calls are passed via the stacks to *parseexp*. The way this function accepts the tokens has already been discussed.

The *syntaxtree* data type is an example of a recursive data type (see the appendix for its specification). So, no strictness annotations have been added to the type. Note that there is also not much point in annotating the *syntaxtree*, since the structure will be built anyway.

The program has been executed with a lambda expression of about 14K characters as input. The timing figures are shown in Table 2. Once again, the program with strict types (without garbage collection time) is about twice as fast. The garbage collection time is highly dependent on the size of the heap. Again, the lazy types lead to extremely high heap usage. With lazy types at least 1 MByte of heap is needed, whereas for the other cases 100 KByte is sufficient.

Table 2. *Timing figures for the scanner/parser*

	Heap size	Execution time	Garbage collection time	Total time
Scanner, lazy types	3 M	0·72	0·28	1·0
Scanner, lazy types	1 M	0·70	1·13	1·83
Scanner, strict types	3 M	0·36	0·0	0·36
Scanner, strict types	100 K	0·36	0·0	0·36

5 Discussion

5.1 Related work

There is growing experience with efficient implementations of lazy functional languages. However, except for Peyton Jones and Launchbury (1991), there is no work that relates directly to partially strict data types. In current implementations of

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

lazy functional languages strictness of data types is restricted to some very specific cases. This holds both for the implementation of LML (Johnsson, 1987), and for the new implementations of Haskell (Hudak *et al.*, 1992). For example, in LML it is possible to add strictness information to algebraic data type definitions. However, it is not clear how this information is used to obtain more efficient code. Furthermore, the use of these annotations is not really encouraged.

A related but incomparable field of research is strictness analysis of non-flat data types (Burn, 1987; Wadler and Hughes, 1987). Though the results of such an analysis state how expressions can be evaluated within a certain context, it is unlikely that this can provide useful information for a general parameter passing mechanism such as we described. Such information might well be useful for implementations on parallel machines (Burn, 1987).

A similar way of adding strictness information to data types has been introduced by Peyton Jones and Launchbury (1991). They describe types, called *unboxed values*, where strictness annotations are considered as a kind of type (note that in contrast, in our approach strictness annotations determine the evaluation contexts of nodes in the graph). As a consequence, the type system has been changed in such a way that strict types are bound to special unboxed data constructors. In contrast to partially strict data types, in which coercions are generated by the compiler, conversions from unboxed to boxed values and *vice versa*, have to be done explicitly. This is also the reason that polymorphic functions cannot immediately be applied to unboxed values (though it is stated that automatic coercions can be introduced). Boxed constructors already existing in the program cannot be used for their corresponding unboxed values. Also in the case of unboxed values, as with our approach, strict recursive data types are problematic, and the authors forbid certain kinds of recursive data types. Because of these limitations, it is unclear to what extent unboxed values should be available in a language: it could be rather difficult for a programmer to explicitly manipulate (un)boxed values. It appears to us that, especially from a programmer's point of view, partially strict data types are preferable to unboxed values. This is especially true if one bears in mind that the same gain in efficiency will be achieved with the former as with the latter.

Peyton Jones and Launchbury (1991) mainly treat the semantic aspects of adding unboxed values to a language. Though the intention of unboxed values is to improve efficiency, the authors do not pay much attention to implementation issues. Because of the similarity to partially strict data types, we can also state that unboxed values will lead to more efficient code. The implementation techniques presented in this paper are equally applicable to unboxed values.

5.2 Conclusion

We have presented a method that allows more efficient functional programs to be written. Often, values that are instances of partially strict data types need not be stored in the heap. Whenever such a value appears in a strict context, it can be passed to other functions on the stacks, or even in a set of registers. Furthermore, the overall memory behaviour is better since less heap space is needed. In this paper we have only

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

discussed how partially strict non-recursive data types can be used for optimizing function calls. More optimizations and extensions are imaginable. For example, in many cases it is possible to store evaluated objects more compactly in the heap. Also, the restriction to non-recursive data type can be relaxed when the sizes of all objects belonging to a type can be determined at compile-time. These are topics for further research.

The motivation for introducing partially strict data types is to improve efficiency. However, they do not have consequences for the expressiveness of the programming language. Of course, it is difficult to estimate whether typical functional programs will benefit from partially strict data types. Experience shows that they are important when using non-recursive data types and functions that deliver multiple results (e.g. using tuples). In these cases, the gain in efficiency more than counterbalances the effort needed to supply the program with additional strictness information. For example, in the Concurrent Clean system, partially strict data types have already proven their usefulness. This system contains an extensive library for specifying general IO (including menus, dialogs and keyboard and mouse input) in a very convenient way (Achten *et al.*, 1992). Without partially strict data types, defining and using this library would have been practically impossible.

A point we want to stress is that strictness annotations can and should be added by the programmer. Often, such strictness is inherent to the types specified by the programmer. For example, the type definition of a complex number is intended to be a strict one, but without language support it is not possible to indicate this. Generally, it is very complicated or even not possible at all to derive such strictness by some kind of static analysis. Nevertheless, thanks to the additional information provided by partially strict data types the strictness analyser can derive more strictness information for other parts of the program. This also gives an important speedup.

Partially strict tuples have been implemented in the Concurrent Clean compiler (Smetsers *et al.*, 1991). The three examples of section 4 demonstrate that programs using these types can become much faster. They also show that partially strict data types can be defined and used in a rather natural way. This means that a programmer can gain efficiency simply by adding a few strictness annotations, and without losing too much of the elegance of the original language.

Acknowledgements

We would like to thank the referees for some useful comments. Special thanks are due to Kevin Hammond for helping us to correct the final version of the paper.

References

Achten, P. M., van Groningen, J. H. G. and Plasmeijer, M. J. 1992. High level specification of I/O in functional languages. In *International Workshop on Functional Languages*, Glasgow, UK, Springer-Verlag (to appear).

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

Burn, G. L. 1987. Evaluation transformers – a model for the parallel evaluation of functional languages (extended abstract). In *Conference on Functional Programming Languages and Computer Architecture*, Portland, OR, 446–470. Springer-Verlag.

Cooley, J. M. and Tukey, J. W. 1965. An algorithm for the machine calculation of complex Fourier series. *Math. Computing*, **19**: 297–301.

van Groningen, J. H. G., Nöcker, E. and Smetsers, J. E. W. 1991. Efficient heap management in the concrete ABC machine. In *Third International Workshop on Implementation of Functional Languages on Parallel Architectures*, Southampton, UK, 383–393. Technical Report Series CSTR91-07.

Heerink, G. 1990. Enkele Benchmarks voor Functionele Talen, University of Nijmegen, Stage Report, March.

Hudak, P., Peyton Jones, S. L., Wadler, P. L., Arvind, Boutel, B., Fairbairn, J., Fasel, J., Guzman, K., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R., Nikhil, R. S., Partain, W. and Peterson, J. 1992. Report on the functional programming language Haskell, version 1.2, *Special Issue of SIGPLAN Notices*, **27**, May.

Johnsson, T. 1987. *Compiling Lazy Functional Programming Languages*. PhD Thesis, Chalmers University, Göteborg, Sweden.

Koopman, P. W. M., van Eekelen, M. C. J. D., Nöcker, E., Plasmeijer, M. J. and Smetsers, J. E. W. 1990. The ABC-machine: a sequential stack-based abstract machine for graph re-writing. University of Nijmegen, Netherlands, Technical Report 90-22, December.

Nöcker, E. 1990. Strictness analysis based on abstract reduction. In *Second International Workshop on Implementation of Functional Languages on Parallel Architectures*. University of Nijmegen, Netherlands, Technical Report 90-16, 297–321.

Nöcker, E. and Smetsers, J. E. W. 1990. Partially strict data types. In *Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, University of Nijmegen, Netherlands, Technical Report 90-16, 237–255.

Nöcker, E., Smetsers, J. E. W., van Eekelen, M. C. J. D. and Plasmeijer, M. J. 1991. Concurrent clean. In *Parallel Architectures and Languages Europe* (*PARLE '91*), Eindhoven, Netherlands, 202–219, Springer-Verlag.

Peyton Jones, S. L. and Launchbury, L. 1991. Unboxed values as first class citizens. In *Conference on Functional Programming Languages and Computer Architecture* (*FPCA '91*), Cambridge, MA, 636–666, Springer-Verlag.

Peyton Jones, S. L. and Salkild, J. 1989. The spineless tagless G-machine. In *Conference on Functional Programming Languages and Computer Architecture* (*FPCA '89*), London, UK, 184–201, Addison Wesley.

Plasmeijer, M. J., van Eekelen, M. C. J. D., Nöcker, E. and Smetsers, J. E. W. 1991. The Concurrent Clean System – Functional programming on the Macintosh. In *7th International Conference of the Apple European University Consortium*, Paris.

Smetsers, J. E. W., Nöcker, E., van Groningen, J. H. G. and Plasmeijer, M. J. 1991. Generating efficient code for lazy functional languages. In *Conference on Functional Programming Languages and Computer Architecture* (*FPCA '91*), Cambridge, MA, 592–617, Springer-Verlag.

Turner, D. A. 1985. Miranda: A non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture* (*FPCA '85*), Nancy, France, 1–16, Springer-Verlag.

Wadler, P. and Hughes, R. J. M. 1987. Projections for strictness analysis. In *Conference on Functional Programming Languages and Computer Architecture* (*FPCA '87*), Portland, OR, 385–407, Springer-Verlag.

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

Appendix A Example programs

The Fast Fourier program:

```
complex    ==    (!num, !num)

plusC  ::  !complex -> !complex -> complex
plusC (r1, i1) (r2, i2)    =    (r1 + r2, i1 + i2)

minC  ::  !complex -> !complex -> complex
minC (r1, i1) (r2, i2)    =    (r1 - r2, i1 - i2)

mulC  ::  !complex -> !complex -> complex
mulC (r1, i1) (r2, i2)    =    (r1 * r2 - i1 * i2, r1 * i2 + i1 * r2)

fast  ::  ![complex] -> !num -> [complex]
fast com length    =    com, length < 2
                   =    merge res_even res_odd length
                        where
                        (even, odd)    =    split com
                        res_even       =    fast even next_length
                        res_odd        =    fast odd next_length
                        next_length    =    length/2

merge  ::  ![complex] -> ![complex] -> !num -> [complex]
merge even odd n
   =   low ++ high
   where
   (low, high)   =   merge' even odd 0
|| merge'   ::   ![complex] -> ![complex] -> !num -> (![complex], ![complex])
   merge' (e:re) (o:ro) i   =   (!ui : urest, !umi : umrest)
                                where
                                (urest, umrest)   =   ! merge' re ro (i+1)
                                ri      =   root i n
                                prod    =   mulC ri o
                                ui      =   plusC e prod
                                umi     =   minC e prod
   merge' []    []    i     =   ([], [])

root   ::   !num -> !num -> complex
root j n   =   (cos z, sin z)
               where
               z   =   (2 * j * pi)/n
```

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

```
split   ::   ![complex] -> (![complex], ![complex])
split (a:b:rest)   =   (a : even, b : odd)
                       where
                       (even, odd)   =   ! split rest
split []           =   ([], []).
```

The lambda scanner/parser:

```
token   ::=   SpecialKeyword !keyword |
              Identifier     !entry   |
              IdentName      !char    |
              Eof

keyword   ::=   OpenSym    |
                CloseSym   |
                DotSym     |
                LambdaSym

env          ==   (!file, !IdentTable)
entry        ==   num
identtable   ==   (!num, ![char])

getnexttoken   ::   !env -> (!token, !env)
getnexttoken (f, t)   =   (token, (file, table))
                          where
                          (c, file)        =   skiplayout (readchar f)
                          token'           =   convertchartotoken c
                          (token, table)   =   putinidenttable (token', t)

putinidenttable   ::   (!token, !identtable) -> (!token, !identtable)
putinidenttable (IdentName c, t)   =   (Identifier entry, table)
                                       where
                                       (entry, table)   =   insert c t
putinidenttable x                  =   x

insert   ::   !char -> !identtable -> (!entry, !identtable)
insert c t   =   (n - index, t), in_table
             =   (new_n, (new_n, c : chars)), otherwise
                 where
                 (n, chars)          =   t
                 (in_table, index)   =   findentry c chars 0
                 new_n               =   n + 1
```

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

```
findentry :: !char -> ![char] -> !num -> (!bool, !num)
findentry c []       n = (False, 0)
findentry c (d:rest) n = (True, n), c = d
                       = findentry c rest (n+1), otherwise

skiplayout :: !file -> (!char, !file)
skiplayout f = ('EofChar', f)      , if endoffile f
             = skip (readchar f)   , otherwise

skip :: (!char, !file) -> (!char, !file)

skip (' ',f)   = skiplayout (readchar f)
skip ('\t',f)  = skiplayout (readchar f)
skip ('\n',f)  = skiplayout (readchar f)
skip x         = x

convertchartotoken :: !char -> token
convertchartotoken '('       = SpecialKeyword OpenSym
convertchartotoken ')'       = SpecialKeyword CloseSym
convertchartotoken '.'       = SpecialKeyword DotSym
convertchartotoken '\\'      = SpecialKeyword LambdaSym
convertchartotoken 'EofChar' = Eof
convertchartotoken c         = IdentName c

syntaxtree ::= Variable entry                      |
               Abstraction entry syntaxtree        |
               Application syntaxtree syntaxtree   |
               Erroneous

parseexp :: !(token, env) -> (!syntaxtree, !env)
parseexp (SpecialKeyword OpenSym, e)
   = (Application exp1 exp2, new_env)
     where
     (exp1, env1) = parseexp (getnexttoken e)
     (exp2, env2) = parseexp (getnexttoken env1)
     new_env      = expectclosesymbol (getnexttoken env2)
parseexp (SpecialKeyword LambdaSym, e)
   = (Abstraction entry exp, new_env)
     where
     (env1, entry)    = expectvar (getnexttoken e)
     env2             = expectdotsymbol (getnexttoken env1)
     (exp, new_env)   = parseexp (getnexttoken env2)
```

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press

```
parseexp (Identifier entry, e)
    =  (Variable entry, e)
parseexp (Eof, e)
    =  print (Erroneous, e) "End of file encountered"
parseexp (token, e)
    =  print (parseexp (getnexttoken e)) "Error: exp expected"

expectclosesymbol :: !(!token, !env) → env
expectclosesymbol (SpecialKeyword CloseSym, e)  =  e
expectclosesymbol (t, e)                         =  print e "CloseSymbol
                                                      Expected"

expectdotsymbol :: !(!token, !env) → env
expectdotsymbol (SpecialKeyword DotSym, e)  =  e
expectdotsymbol (t, e)                       =  print e "DotSymbol Expected"

expectvar :: !(!token, !env) → (!env, !entry)
expectvar (Identifier entry, e)  =  (e, entry)
expectvar (t, e)                 =  print (e, 0) "Variable Expected"

readchar :: !file → (!char, !file)

print :: * → [char] → *
```

https://doi.org/10.1017/S0956796800000691 Published online by Cambridge University Press