Generating Efficient Code for Lazy Functional Languages

Sjaak Smetsers, Eric Nöcker, John van Groningen, Rinus Plasmeijer
University of Nijmegen*

Abstract

In this paper we will discuss how a good code generator can be built for (lazy) functional languages. Starting from Concurrent Clean, an experimental lazy functional programming language, code is generated for an intermediate abstract machine: the ABC machine. In this first pass many well-known optimisation techniques are included. However, we will also present some new ideas in this area, like the way in which strictness can be incorporated, and the implementation of higher order functions. In a second pass, the ABC code is translated to concrete target code for the Motorola MC680x0 processor. Again many optimisation methods appear to be applicable. Some of them (for example register allocation algorithms) are common for the implementation of other types of languages, but have to be adapted because of the specific properties of both source language and target machine. Other optimisations are specific for lazy functional languages, e.g. the implementation of higher order functions, efficient memory management and the optimised graph reduction. Measurements demonstrate that due to the optimisations of both passes very fast code can be generated. We have compared Concurrent Clean with two other functional languages, namely Lml and Hope, and also with the imperative language C. With respect to both functional languages this comparison clearly goes in favour of Concurrent Clean. Furthermore, we can conclude that, when using the presented compilation techniques, a lazy functional language is able to compete even with an imperative language such as C.

1. Introduction

It is very difficult to build an efficient implementation of a lazy functional language. In the recent years many compilation techniques have been presented, mostly with the aid of some abstract machine architecture. However, relatively little attention was spent on the methods to implement such an abstract machine (and therefore also the functional language) efficiently on a concrete machine.

In this paper we will discuss how a good code generator can be built for a lazy functional language. We describe a compiler for Concurrent Clean, an experimental lazy functional programming language (Brus et al. (1987),Eekelen et al. (1990),Nöcker et al. (1991)). This language contains almost no syntactical sugar which enables us to

*Faculty of Mathematics and Computer Science, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands E-mail: `clean@cs.kun.nl`

concentrate fully on efficiency topics. We note that the compiler includes a strictness analyser, based on abstract reduction (Nöcker (1990)), which plays an essential role during the compilation process.

Concurrent Clean is compiled (Smetsers (1989)) first to code for an intermediate abstract machine: the ABC machine. The ABC machine (Koopman et al. (1990)) is a stack based graph reduction machine, similar to advanced G-machine like architectures (e.g. Johnsson (1987),Peyton Jones and Salkild (1989)). In a second pass, the resulting ABC code is translated to concrete target code for the Motorola MC68020 processor (Groningen (1990)). Generating intermediate code has some advantages. The ABC machine by itself can easily be understood and implemented. Interpreters (Nöcker (1989)) as well as simple code generators (based on macro substitution of ABC instructions) are easily to build. As a consequence, it is easy to experiment with Concurrent Clean, the ABC machine or the compiler itself. A disadvantage of intermediate code is the possible loss of efficiency. The ABC code that is generated by the first pass seems to be clumsy. Furthermore, information needed by the second pass might no longer be available. However, we will show that it is very well possible to circumvent these problems and to generate efficient code.

1.1. Overview of the paper

In the rest of this introduction we will give a very short overview of the language Concurrent Clean and the ABC machine. In section 2 we will treat the Concurrent Clean to ABC code compilation. The main task of this compilation pass is to derive and process type and strictness information as adequate as possible. In the second pass, ABC code is translated to concrete machine code. Specific properties of a target machine are exploited. This is discussed in section 3. Note that we describe the generation of code for the Motorola MC68020 processor. It should be pointed out that most of the techniques are generally applicable. In section 4 we compare the implementation of Concurrent Clean with implementations other languages. Finally, we present conclusions and future work (section 5).

1.2. Concurrent Clean

Concurrent Clean is an experimental, lazy, higher-order functional programming language based on term graph rewriting (Barendregt et al. (1987),Plasmeijer and van Eekelen (1993),Barendsen and Smetsers (1992)). Concurrent Clean has many features in common with other lazy, higher-order functional languages, such as a Milner/Mycroft based polymorphic type system (including algebraic types, synonym types and abstract types) (Milner (1978),Mycroft (1984)). A key aspect of the language is that the main object that is manipulated by a program is a graph. Consequently, the programmer can explicitly indicate sharing of computations. For instance, cyclic objects can be created. The most important aspect of Concurrent Clean discussed in this paper is the way in which the order of evaluation can be controlled. Lazy evaluation can locally be changed to eager evaluation which has the advantage that it generally can be implemented considerably more efficiently than lazy evaluation. Even larger speed-up can be achieved by changing sequential evaluation into parallel evaluation. For this reason, Concurrent Clean also offers the possibility to indicate parallel execution by the use of annotations (Eekelen (1988),Eekelen et al. (1991)).

We will introduce Concurrent Clean by showing some well-known example functions (for more examples see Eekelen et al. (1990)). Consider the following definitions for the factorial function, and the function Map:

```
RULE
::  Fac !INT              →  INT                   ;
    Fac 0                 →  1                     |
    Fac n                 →  *I n (Fac (–I n))     ;

::  Map (⇒ x y) ![x]      →  [y]                   ;
    Map f [ ]             →  [ ]                   |
    Map f [a | b]         →  [f a | Map f b]       ;

::  Start                 →  [INT]                 ;
    Start                 →  Map Fac [2,3,4]       ;
```

Each function, optionally preceded by a type specification, consists of a number of alternatives. Square brackets are used for denoting lists: [ ] is an empty list, [a | b] denotes a list consisting of a list b prefixed with an element a. The example also shows that higher order functions can be used freely. There is no difference between the use of full and partial (curried) applications of functions. Types of higher order functions are specified using ⇒ (prefix notation) which corresponds to → (infix notation) as used in most other functional languages.

The sequential flow of control can be influenced by means of so called *strict annotations*. Annotations can be placed in right hand sides of function definitions, in type specifications of functions, or in type definitions.

Annotations in a type specification of a certain function are allowed to be placed before the type specification of either an argument on the left-hand-side or an argument of a tuple type appearing in a strict context. A tuple type is in a *strict context* if it has been supplied with a strict annotation itself or if it appears as the root node on the right-hand-side of the type rule. Intuitively, such a strict annotation indicates that the corresponding argument is always reduced to root normal form before the corresponding rule is applied. For example, the function Fac is made strict in its argument. It should be stated that this particular strictness property would also have been found by the strictness analyser. Strict annotations may also be used in tuple types appearing in type synonym definitions The meaning of these annotated synonym types can be explained with the aid of a simple program transformation in which all occurrences of synonym types are replaced by their right-hand-sides (of course, annotations included). These annotated type definitions are a special case of the *partially strict data types* (Nöcker and Smetsers (1990)). An example of a partially strict tuple is the following definition of a complex number:

```
TYPE
::  Complex                    →  (!REAL,!REAL)          ;

RULE
::  +C !Complex !Complex       →  Complex                ;
    +C (r1,i1) (r2,i2)         →  (+R r1 r2,+R i1 i2)    ;
```

The annotations in the type definition for Complex provide that both the real and imaginary part of a complex number are computed immediately when the complex number appears in a strict context.

1.3. The ABC Machine

Since a complete, formal description of the ABC machine goes beyond the scope of this paper, we will give only a short introduction. In the sequel, specific parts of the machine will be highlighted further if necessary.

The ABC machine is a stack based graph reduction machine. Its main parts of interest are the three stacks (A, B and C stack) and the heap. The C stack is used for storing code addresses. The other two stacks are used for evaluating or building expressions, for passing arguments to functions and for returning results from functions. The A stack contains addresses of nodes in the heap, whereas the B stack contains values of basic types, such as integers or reals. Thus, basic values can be represented in two ways: as a node in the heap or as an item on the B stack. Graphs are stored in the heap. Conceptually, the heap consists of a collection of nodes. A node in the ABC machine represents a node of a Concurrent Clean graph. Hence, nodes in the ABC machine have a variable size.

The ABC instructions that will be used in this paper are almost self-explanatory. In any case, pieces of ABC code will be commented.

2. Concurrent Clean to ABC Code Generation

The main task of the Concurrent Clean compiler is to generate efficient ABC code. The syntax of Concurrent Clean is rather simple: no complex transformations like lambda lifting or the conversion of ZF-expressions are necessary. Many standard optimisation techniques are implemented: tail recursion removal, avoiding unnecessary evaluation calls and so on. In the sequel we will emphasize those parts of the compiler that differ from other well-known implementations.

2.1. The Basic Machinery

Conceptually, graph reduction takes place in the heap: whenever a graph has to be rewritten a new graph is built. The root node of the old graph will be overwritten with the root node of the new graph. Unfortunately, this scheme will not give efficient code. The goal of the compiler is to generate code in which graph building is prevented as much as possible. For generating such efficient code type and strictness information is necessary. Strictness information is used in two ways. First, it is used for deriving the evaluation order of right hand sides. Second it is used for parameter passing. The latter is treated in the next section. First we will discus how nodes are represented and reduced to head normal form.

The structure of nodes

Generally spoken, a node of a Concurrent Clean graph consists of a symbol with a certain number of arguments. Representing a node as a variable sized object causes problems with updating: the new node doesn't need to fit in the space of old one. This problem can be solved by introducing indirection nodes, but this will slow down the access to the contents of a node. In the ABC machine a node is split in a fixed and a variable sized part. The fixed size part contains a descriptor, a code pointer and a pointer to a variable sized part.

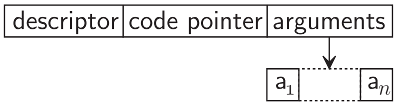

Fig 2.1 The node structure.

The *descriptor* is a representation of a Clean symbol. Normally, it is an index or pointer in a descriptor table. Descriptors are used in the following cases:

- pattern matching. Nodes contain the same symbol if they have the same descriptor.
- printing. The descriptor contains a string representing the symbol belonging to that descriptor.
- fetching arity. The arity is needed by the garbage collector, and by some ABC instructions.
- evaluating higher order functions (see further).

The code pointer refers to code with which the node can be evaluated to head normal form. This code is entered by a jsr_eval instruction. During reduction the code pointer can be changed. For example, after entering the node for evaluation a pointer to an error routine can be stored. If the node is ever entered again (indicating a non-terminating reduction) this code will be executed. If a node is updated with a head normal form value, the code pointer points to special code just containing a return statement:

```
_hnf_code:      rtn
```

In the variable sized part the arguments of the node are stored. This means that the arguments have to be fetched via an extra indirection. On the other hand, updating a node is simple: update the fixed part, and allocate space for the arguments.

For nodes containing a basic value, e.gȧn integer, the descriptor does not represent the Clean symbol (that would be the integer value itself). Instead, all integers share the same descriptor (e.g. INT). The integer value itself is stored in the argument reference part. For basic values that do not fit in the fixed part of a node (e.g. strings) a pointer to the value (for which space has to be allocated) is stored. Since basic nodes are always in normal form, they all contain the head normal form code pointer.

The evaluation of right hand sides

During the generation of code for the right hand side of an alternative, strictness information is used in order to derive what nodes are in a *strict context*. Generally, a node is in a strict context if it certainly has to be reduced. For a node in a strict context no intermediate graph needs to be built. Instead, the code that would have been stored in the node can be executed directly. Consider the following example:

```
F x y        →   G n1 n2,
                 n1: G x y,
                 n2: G y x      ;
```

Suppose that G is only strict in its first argument. Then it is easy to see that nodes n1 and x are in a strict context, and nodes n2 and y are not. The root node itself is always in a strict context. This means that only node n2 is built (node y is already available). The other two applications of G are implemented by direct calls to the code of G. If, at some time, node n2 appears to be needed, it will be unpacked, whereafter also the code for G is executed. For this reason, the code for G has two entry points: the *node_entry* for the lazy evaluation, and the *strict_entry* for the strict one. For the node entry only the node itself is needed. The code unpacks the node, whereafter the strict entry is called. For the strict entry the arguments of the function are needed. They are passed via the stacks. How parameters are passed depends on the type of the function. This is discussed in the next section. The code for the right hand side of F would look as follows:

```
F:                      | strict entry of F
    jsr_eval            | evaluate node x (top of stack) to head normal from
    create              | create node n2
    push_a 1            | push x
    push_a 3            | push y
    fill G nG 2 2       | fill node n2 with (G y x), x and y are removed
    create              | create a node for the result of G
    push_a 3            | push y
    push_a 3            | push x
    jsr G               | jsr to the strict entry of G (evaluation of node n1)
    update_a 1 3        | clean the
    update_a 0 2        | stacks: remove the old
    pop_a 2             | x and y nodes
    jmp G               | and jump to strict entry of G
```

Note that the last call to G is a direct jump instead of a subroutine call.

2.2. Passing Parameters and Returning Results

As mentioned before, the way the strict entry of a function expects its arguments and returns its result is determined by the type of this function. If a function is called (e.g. because its node is in a strict context), the calling code has to ensure that these calling conventions are obeyed. The general calling convention is straightforward. Things become complicated if the type of a function contains basic types or strict tuples.

General way of parameter passing

We will start by considering the most general situation. Suppose a function F of arity n with type:

$$:: \quad \mathsf{F}\ \mathsf{T}_1\ \ldots\ \mathsf{T}_n \quad \rightarrow \quad \mathsf{T}_r \quad ;$$

where none of the types is a basic type or a strict tuple. The above function F expects all its arguments and the node that has to be updated with the result on the A stack.

So, if the function F is to be called, first the node to be updated has to be put on the stack. Thereafter, graphs for the arguments for F have to be built (if necessary) and pushed on the A stack. Strict arguments have to be reduced before calling F: the function *expects* them to be in head normal form.

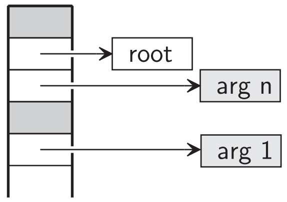

Fig 2.2: the A stack just before calling F.

The code for F takes care of the removal of the arguments. The updating of the root node is done only if it can be overwritten with a head normal form value. Consider for example the function:

```
F [ ] y      →   G y y       |
F x y        →   [ x | y ]   ;
```

In the first alternative of F another function G is called. This function will get the same root node as F. This root node will either be updated by the code for G, or by another function that is called by G. In the second alternative the root node will be updated with a Cons node:

```
alt1:                          | entry for the first alternative
      update_a 1 0             | replace topmost A stack element ([ ]) by y
      jmp sG                   | jump to the strict entry of G
alt2:                          | entry for the second alternative
      fill Cons _hnf_code 2 2  | fill the root node with the cons (x and y removed)
      rtn                      | and return
```

Notice that because of the call to G is tail recursive, the A stack does not grow: the original arguments of F are replaced by the new ones for G.

Passing strict basic arguments

This general scheme changes if either one of the arguments or the result type is a strict basic type. As is the case for ordinary strict arguments, the caller has to reduce the strict arguments first. But instead of the A stack, the B stack is now used for passing values belonging to the strict basic types. For example:

```
::   F !INT [x] !REAL      →   INT      ;
```

The integer and real values needed by F are evaluated (if they have not been evaluated before) and pushed on the B stack, whereas the second argument is pushed on the A stack:

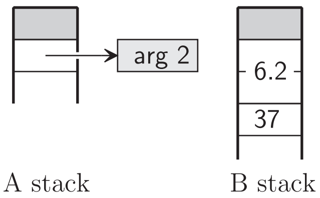

Fig 2.3: the stacks just before calling F.

Note that no root node is passed to F. The result value of F is returned via the B stack. The calling code itself decides what to do with it: it might use the value to fill a node, or pass it to another function.

Passing partially strict datatypes

For values that have a partially strict tuple type, it is exactly known at compile time how far the parts of the tuple have been reduced. So, whenever such a value is passed to another function, it is sufficient to pass only the arguments of the tuple instead of the whole tuple. Consider, for example, the following function type:

```
::    F INT ! (! INT,! [CHAR])    →    (INT,! CHAR)    ;
```

The function F requires two arguments. The first one is a non-strict integer. This value is passed via the A stack. The second argument is a strict tuple. Both elements of this tuple have to be reduced to head normal form before calling F. Then they can be passed to F via the stacks; the tuple itself needs not to be built. The integer is passed via the B stack, whereas the character list is passed via the A stack. The result value is handled in a similar way: only the elements of the tuple are returned. The first element, a non-strict integer, will be returned via the A stack, and the second, a strict character, via the B stack. In this way no intermediate tuple node is needed.

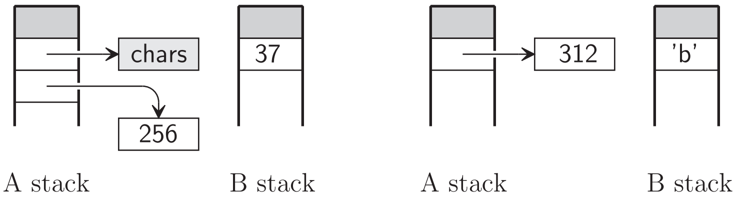

Fig 2.4: The stacks just before and after calling F.

Selector functions

If the return type of a function is a single type variable that is bound in one of the argument types, we have another special case. Consider, for example, the function Hd that return the first element of a list:

```
::    Hd [x]        →    x    ;
      Hd [x | r]    →    x    ;
```

Functions with such a type are called *selector functions*: normally, they select an already existing node from a graph. Following the previous conventions, Hd will receive a root node for storing its result. The root node will be updated with the head normal form value of x:

```
Hd_alt1:          | First alt of Hd: Cons node, elements and root on top of A
    jsr_eval      | evaluate x
    fill_a 0 3    | update the root node with the contents of node x
    pop_a 3       | clear the A stack
    rtn           | and return
```

However, since node x already exists, it would be better to return that node itself. The calling code does not need to pass a root node to such a function which will prevent the creation of new, unnecessary nodes. The code for the right hand side of Hd becomes:

```
Hd_alt1:                    | First alt: Cons node, elements (no root!) on top of A
      jsr_eval              | evaluate x
      update_a 0 2          | replace the elements of the A stack
      pop_a 2               | by x
      rtn                   | and return
```

Coercions

The type of a function together with the strictness information determines the way this function expects the layout of the stacks when it is called. When the actual layout of the stack differs from the layout that the function expects at the strict entry there is a conflict. For example, such a conflict occurs in the node entry of a function: strict arguments (that still reside in the heap) have to be reduced and, in the case of basic types, pushed on the B stack. The conversion that is necessary to solve the conflicting situation is called a *coercion*. In the node entry, a coercion is also necessary if the result has a basic type: the root node has to be filled with the basic value that is returned via the B stack. Consider for example a function with type:

```
:: F !INT ![x] → INT ;
```

The ABC code of the node entry would look as follows:

```
nF:                         | node entry of F: a node with symbol F on top of A stack
      push_args 0 2 2       | push the two arguments on the A stack
      push_a 1              | push the second argument on top
      jsr_eval              | and evaluate it
      pop_a 1               | clear the stack
      jsr_eval              | evaluate the first argument (on top of A)
      pushI_a 0 0           | and move the integer value from the node to the B stack
      pop_a 1               | remove the (integer) node form the A stack
      jsr sF                | call the strict entry of F (second arg on A, first arg on B)
      fillI 0 0             | fill root node (top of A) with the integer result (top of B)
      pop_b 1               | clear the B stack
      rtn                   | and return
```

The above kind of coercion is straightforward. However, in the case of partially strict data types coercions can become very complex pieces of code. It appears to us that the cases needing complicated coercions occur rather rarely.

2.3. Higher Order Functions

In Concurrent Clean, symbols are defined with a fixed arity (from now on called the *formal arity*). However, each symbol can be applied to a number of arguments arguments (i.e. the *actual arity*) which is less than the formal arity (currying). Such a partial application can be represented as a spine of application nodes. However, a better way is to use *partial nodes*, i.e. nodes with a partially filled argument part. Such nodes are built as standard nodes, but contain special descriptors. This implies that for each Concurrent Clean symbol of arity n, n+1 descriptors are defined. In many respects, the ABC machine treats partial nodes in the same way as standard nodes. However, a partial node may be applied to another node with the *Apply* function (for convenience just called Apply here). Apply is a special built-in function with type:

```
::    Apply !(⇒ x y) x        →    y        ;
```

It is inserted where hidden function applications appear. Consider, for example, the function Twice:

```
::    Twice (⇒ x x) x         →    x        ;
      Twice f x               →    f (f x)  ;
```

Its right hand side is internally transformed into:

```
      Twice f x               →    Apply f (Apply f x)    ;
```

For the first appearance of Apply a direct call to the Apply code is generated. The second application of Apply is not in a strict context: an Apply node will be built. The first thing the Apply code will do is an inspection of its first argument, the partial node. If that node needs precisely one more argument (being the second argument of the Apply), all arguments are available: the function of the partial node can now be called. Otherwise, a new partial node will be built. This new node is a copy of the original node, with one extra argument. The ABC code for the Apply function looks as follows:

```
apply_code:                  | the strict entry of Apply
     get_node_arity 0        | get the real arity of the partial node
     get_desc_arity 0        | get the maximum arity of the symbol of the partial node
     subI                    | subtract
     eqI_b +1 0              | test if exactly one argument is needed
     jmp_false args_needed
     push_ap_entry 0         | yes, so push the code entry of the function on C stack
     pop_b 1                 | clear the B stack
     rtn                     | jump to the code entry of the function
args_needed:
     create                  | no, so create a new node
     push_a 2                | get the argument to be added
     add_args 2 1 1          | fill the new node with one extra argument
     update_a 0 2            | clear both stacks
     pop_a 2
     pop_b 1
     rtn                     | and return
```

It will be clear that the Apply code needs to fetch the actual arity, the formal arity as well as a code address. This information is found via the descriptor which points to an entry of the descriptor table (see Fig 2.5):

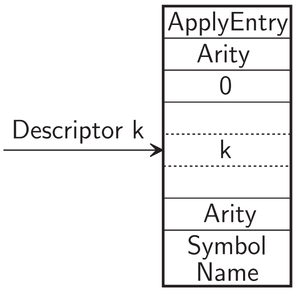

Fig 2.5: The layout of descriptors.

The method is at a disadvantage in case long spines of Apply nodes are to be built: each of these intermediate nodes has to be updated by a partial node. However, such long spines occur very rarely. In other cases the method is very fast: the efficient representation of spines saves heap space, and no 'spine searching' is needed.

2.4. General Layout of the Code, Entry Points

The actual work of a function is done by the code of the strict entry. As discussed above, the strict entry expects the arguments of the function in the right form on the stacks. If these calling conventions cannot be applied directly, additional entry points are used. The code of these entry points will perform some coercions, whereafter execution proceeds with the strict entry. Earlier, we already introduced the node_entry. There are two other ways in which a function can be called.

Firstly, a function can be called via an application of the Apply function. This happens when a partial node is supplied with a sufficient number of arguments. The arguments of the function stored in the partial node have to be pushed on the A stack, whereafter they have to be coerced as required by the calling conventions of the function. The entry that does this work is called the *apply_entry*.

Secondly, also special things have to be done for exported functions. The exported type determines the calling convention outside the module. However, inside the module another calling convention may be more efficient. This is the case if abstract types are exported (hiding the internal representation), or if the strictness analyser finds more information than has been exported. So, for strict calls from outside an additional entry point is needed. This *external_strict_entry* converts the external to the internal calling conventions and continues with the internal strict entry.

So, in general the layout of the entry part of the code of a function with arity 'ar' is as follows:

```
apply_entry:
        repl_args ar-1 ar-1        | push the first (arity-1) args on the A stack
        jmp convert_code           | do coercions and execute strict code
node_entry:
        push_args ar ar 0          | push all the arguments on the A stack
convert_code:
        ...                        | convert strict args, unpack if necessary etc
```

```
        jmp strict_entry            | jump to the strict entry
    external_strict_entry:
        ...                         | convert strict args, unpack if necessary etc
    strict_entry:                   | and continue with strict entry
```

This layout is typical for a function that uses the general calling conventions. For other functions the layout is different. For example, if a function returns an integer value, the node entry itself has to provide that the root node (which initially contains the application of that function) is overwritten the integer result. The node entry for a function returning an integer would look as follows:

```
    node_entry:
        push_args arity arity 0     | push all the arguments on the A stack
    convert_code:
        ...                         | convert strict args, unpack if necessary etc
        jsr strict_entry            | jump-subroutine to the strict entry
        fillI 0 0                   | and fill the root node with the integer result
        pop_b 1                     | clear the B stack
        rtn                         | and return
```

The strict entry depends of course on the body of the function itself. For each alternative of the function there is a piece of match code, and code for the right hand side. Pattern matching is done straightforwardly.

```
    strict_entry:
    match_code_1:
        ...                     | matching code for the first alternative
    alt_1:
        ...                     | code for the first alternative
        ...                     | other alternatives
    match_code_n:
        ...                     | matching code for the last alternative
    alt_n:
        ...                     | code for the last alternative
```

3. Generating Code towards Register Based Machines

In this section we will describe how abstract ABC code can be translated to real machine code. The ideas will be illustrated using a target machine based on the Motorola 68020 processor. A short introduction to this machine is given in the next section.

The most important aspects of implementing the ABC machine are:

- an efficient implementation of the components of the ABC machine state. As all computations take place in the heap and on the stacks, a good management of these structures is essential. This is handled in section 3.2.
- optimisation of ABC instruction sequences. It seems that the generated ABC code is not concerned with efficiency. For example, many ABC instructions require their arguments on top of the stack whereas on a real machine the arguments of corresponding instructions can be accessed more directly using one of the various addressing modes. This implies that the copy actions specified by the ABC

instructions are not really necessary. How computations specified in ABC code can be optimised when generating machine code is described in section 3.3.

- use of registers, if possible. The MC68020 processors have the property that computations are performed much faster in registers than in memory. Registers can be used, for example, for implementing the ABC stacks and heap, for storing temporary values, and for passing arguments and return values. Register allocation is discussed in section 3.4.

3.1. The MC680x0 Family

There are two reasons that we have chosen for the Motorola 680x0 family of processors (68000, 68010, 68020, 68030, ...) as target machine (Motorola (1985/6)). First of all, these kind of processors have been used in several widespread machines such as the SUN3, Apple Macintosh and Atari ST. Furthermore, the Motorola processors are very suited to serve as an actual target machine to illustrate how the ABC machine (and therefore functional languages in general) can be implemented efficiently. It should be no problem to use the presented ideas when generating actual target code for other register based processors (such as the Intel 80x86 family).

In this paper the MC68020 processor is used as example. It contains besides a program counter and a status register two kinds of general purpose registers, to wit data and address registers, eight of each kind. The data registers, often indicated by d0-d7, are mainly used in arithmetical operations whilst the address registers (indicated by a0-a7) can be used to access data structures that are kept in memory. An example of a data structure that can be implemented very efficiently with address registers is a stack. This is also due to the fact that address registers can be used in combination with many different addressing modes supported by these processors. Examples of such addressing modes are: address register indirect possible combined with post-increment, pre-decrement or displacement.

For the remainder of this paper we do not require any knowledge about the specific structure of the MC68020 processor. Some familiarity with assembly languages will suffice to understand the examples of real code given in this section.

3.2. Representing the ABC Machine

Mapping the components of the ABC machine (i.e. ABC stacks, graph store (heap) and descriptors) onto the MC68020 does not cause many difficulties. Stacks can be implemented straightforwardly using some of the address registers. Implementing the heap takes some more doing. The structure of the descriptors mainly depends on how higher order functions are implemented. Their representation can be found later on in this section where the higher order functions are treated.

The C stack

For the C stack the system stack is taken (i.e. the stack used by the processor itself when performing a subroutine call). Therefore, the jump and return instructions of the ABC machine can be mapped directly on those of the MC68020 (of course, for the jsr_eval instructions other things have to be done, see further on). This implies that address register a7 (normally called *sp*) is reserved.

The A and B stack

The A and B stack are allocated in one contiguous area of memory where they can grow in opposite direction. In this way a check on stack overflow of both A and B stack can be done with a few instructions (just compare the two stack pointers and check whether their difference is not negative). The pointers to the tops of the stack are held in registers: for the A stack register a3 is reserved, for the B stack register a4 (for convenience we will refer to those registers by *asp* and *bsp* from now on).

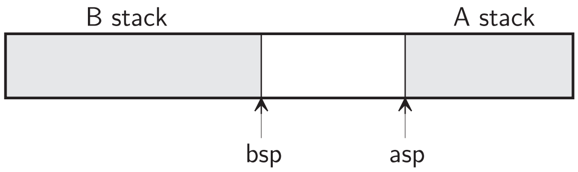

Fig 3.1: The layout of the combined A and B stack.

The heap

For the heap a contiguous area of memory is reserved. The pointer to the free area is stored in register a6 (called *hp*), whereas the number of free heap cells (1 heap cell = 1 long word = 4 bytes) is stored in register d7 (*fh* from now on). With this representation the allocation of memory becomes cheap. Also, the filling of newly created nodes in the heap can be done efficiently as illustrated by the next example.

Suppose that a Cons node should be created that has two arguments. The references to both arguments are kept in the address registers a1 and a2. At the end a pointer to the new node is returned in register a0. First, it has to be checked whether there is enough space in the heap. This is done by the following instructions:

```
subq.l   #4,fh          ; we need 4 long words for storing the new node
bcs      call_gc        ; call the garbage collector if not enough free heap
                        ; space is available
```

Now the heap pointer (held in register hp) refers to the first free long word in the heap. Filling the node by using hp is rather straightforward (for the actual representation of nodes see further):

```
return_from_gc:
    move.l   hp, d0          ; first the variable part is filled
                             ; a pointer to it is temporarily stored in d0
    move.l   a1, (hp)+       ; copy the pointer to the first argument to the new node
    move.l   a2, (hp)+       ; copy the pointer to the second argument to the new node
    move.l   hp, a0          ; now the fixed part is treated.
                             ; store a pointer to it in a0
    move.l   #Cons, (hp)+    ; fill up the descriptor field with Cons
    move.l   d0, (hp)+       ; copy the pointer to the variable part
```

Memory is recycled by a process called *garbage collection*. In our implementation a *copying garbage collector* is used which divides the memory available for the heap into two areas (*semispaces*). The nodes of the graph are stored in one semispace. When this area is filled up the garbage collector copies all the nodes still needed for the execution of the program to the other semispace leaving all the garbage behind.

Representation of nodes

As described earlier, a node in the ABC machine consists of a fixed and a variable sized part. The fixed size part consists of a pointer to a descriptor, a code pointer and a pointer to the variable sized part. A drawback of the ABC node structure is that the size of the nodes is relatively large: the fixed part would consist of 3 long words (12 bytes), one long word for each pointer. It is important that nodes are as small as possible: creating, filling and copying of nodes can be done faster and because less memory is consumed, also the garbage collector will be called less often. We can observe that if a node is in head normal form, its code field points to the head normal form code so in fact only the pointer to the descriptor is of interest. On the other hand, if a node contains an (unevaluated) expression the descriptor is not used. This observation makes it possible to combine the descriptor and code field into one, reducing the size of the fixed part by one third. However, one little problem has to be solved: the arity of a node is needed by the garbage collection. This arity, stored in the descriptor table, is not accessible when the descriptor is no longer available. The problem is solved by storing the arity not only in the descriptor table but also just before the node entry such that it can be accessed via the code pointer.

The new node structure is illustrated in the next picture. A disadvantage is that each node has to be supplied with a tag: the highest bit of the first word of the code/descr field (note that this field consists of two words) indicates whether it contains a descriptor or a code address. If this bit is set, the second word is an index in the descriptor table. Otherwise, the code/descr field contains a code pointer that is used to reduce the node to root normal form.

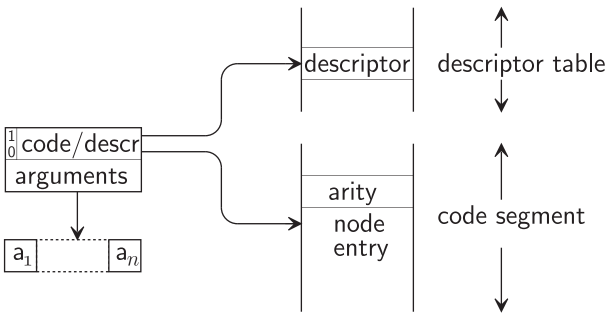

Fig 3.2: The structure of nodes.

In the ABC machine nodes can be evaluated to root normal form by means of the jsr_eval instruction. This instruction fetches the code address from the node on top of the A stack and jumps to it. Due to the combined code/desc field we first have to check whether the node is already in root normal form before the jump to the evaluation code is made. The next piece of MC68020 code shows how this can be achieved (assume that register a1 is referring to the node that is going to be reduced):

```
        move.l          (a1), d6                ; get the code/desc field
        bmi             is_in_rnf               ; check whether the highest bit is set
        save_registers                          ; save all the registers in use
        move.l          a1, a0                  ; provide that a pointer to the node that
                                                ; is going to be evaluated is in reg. a0
        move.l          d6, a1                  ; move the evaluation address in a1
        jsr             (a1)                    ; call the evaluation code
        move.l          a0,a1                   ; move the result of the evaluation back in a1
        restore_registers                       ; restore the previously saved registers
is_in_rnf:
        ...
```

Note that, in case of a node not in root normal form, this alternative representation leads to slightly less efficient code (an extra move instruction and a conditional branch are needed). But when the node is already in root normal form the code becomes much faster, for the saving and restoring of the registers is not needed anymore.

The descriptor table

In section 2.3 we described how partial (curried) function applications are implemented on the ABC machine. The ABC code shows that both the formal and actual arity of the node are needed. Furthermore, when all the arguments are available, the apply entry is also needed. The arities as well as the apply entry are found in the descriptor table that can be accessed via the descriptor stored in the node. A straightforward translation of the ABC apply code will result in rather inefficient MC68020 code. To increase efficiency, not only the actual arity of the curried application is stored in the descriptor but also a pointer to the code that should be executed when a partial application of the corresponding symbol is applied to an additional argument. This results in the following representation of symbol descriptors:

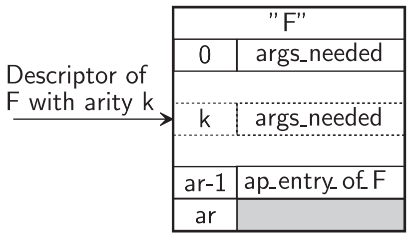

Fig 3.3: The lay-out of a descriptor in the descriptor table.

F is a function with arity ar. The descriptor contains a string representation of the name and ar+1 entries. Now, the pointer stored in the descriptor field of the node is just a reference to the entry that corresponds to the arity with which F is actually applied. Besides this actual arity that is used by the garbage collector, each entry contains the code that should be reduced when a partial application of F is applied to an additional argument. It should be clear that this code is just the args_needed code as presented in section 2.3 unless the curried application has already ar-1 arguments. In that case the extra argument provides that this particular application becomes complete. So the apply entry of F can be called.

The descriptors of all symbols defined in a Clean program are stored in a so-called *descriptor table*. As a consequence, each application of a symbol F with actual arity k can be represented by an offset in this table that corresponds to the k^{th} entry of the descriptor of F.

With the aid of the previous representation the translation of the apply code will result in the following ABC instructions (we assume that register a1 refers to the node containing the partial application and register st refers to the beginning of the symbol table):

```
move      2(a1), a2     ; get the offset of the descriptor entry
add.l     st, a2        ; add this offset to the beginning of the descriptor table
move.l    2(a2), a2     ; retrieve the reduction code
jsr       (a2)          ; call the reduction code
```

3.3. Generating MC68020 Code

A straightforward way of generating concrete machine code is by means of macro expansion: each ABC instruction is considered as a macro application that is substituted by a sequence of MC68020 instructions. The main disadvantage of this method is that when generation code for a certain instruction the context of that instruction is not taken into account. An illustrating example is given by the next piece of ABC code together with MC68020 code that might be the result of performing macro expansions. Assume the following macro definitions:

```
#macro     push_b(n)     move.l -((n+1)*4)(bsp), (bsp)+
#macro     addI          move.l -(bsp), d0
                         add.l d0, -4(bsp)
```

And the following ABC instructions:

```
push_b     0
push_b     2
addI
```

Applying the above standing macro definitions will result in:

```
move.l     -4(bsp), (bsp)+
move.l     -12(bsp), (bsp)+
move.l     -(bsp), d0
add.l      d0, -4(bsp)
```

However, if all the three instruction were considered simultaneously, one could use that fact that the MC68020 add instruction does not require that the arguments are on top of the B stack. A more efficient code generator might compile the three abc instructions into three MC68020 instructions which are about 30% faster:

```
move.l     -4(bsp), d0
add.l      -8(bsp), d0
move.l     d0, (bsp)+
```

So, before generating code it is useful to group ABC instructions into blocks. These so called *basic blocks* should have the property that they can be considered as atomic actions of the ABC machine. They specify state transitions that convert the state of the ABC machine (which is determined by the contents of the stacks and the graph store) at the beginning of these blocks into the final state at the end of the basic blocks. Now the task of a code generator becomes to implement such actions as efficient as possible. It is obvious that the largest gain will be achieved when it is tried to make these basic blocks as big as possible. In our code generator a basic block consist of the maximal sequence of ABC instructions that does not contain any label definitions or instructions that might change the flow of control (e.g. subroutine calls or conditional branches).

With the aid of basic blocks the compile-time analysis of ABC programs is simplified. A few examples:

- Flow of control: the original ABC instructions specify an evaluation order. Grouping these instructions into basic blocks allows us to deviate from this order as long as the new evaluation does not affect the final result at the end of the current basic block (i.e. the new code sequence should specify the same state transition). Changing the evaluation order makes it possible to improve the generated code as can be seen in the following example. Suppose we want to compute d0+d1 and d0+d2 in any register and d1 is used after these computations, but d0 and d2 not. First computing d0+d1 and then d0+d2:

```
move.l    d0, d3
add.l     d1, d3
add.l     d2, d0
```

  But it would be better if d0+d2 was computed first. It saves not only one instruction but also one register:

```
add.l     d0, d2
add.l     d0, d1
```

  In the ideal case the code generator will determine an evaluation order of instructions such that the execution costs of the generated code are as low as possible. The execution costs can be found by simply adding the execution times of all the individual instructions (One should notice that the execution time of an instruction may depend on the preceding instructions). But the problem of finding an evaluation order in such a way that the total time is minimal is NP-complete, which makes an algorithm based on this strategy useless. A different approach is to minimise the number of registers needed to evaluate a basic block. This approach seems reasonable: since registers of a real processor are relatively sparse, the quality of the generated code strongly depends on how well they are utilised. Also, by decreasing the number of used registers generally less instructions are needed.

- Memory allocation: a basic block may contain instructions that reserve space in the heap. Such an instruction has to check whether the heap contains enough free cells. If this is not the case, the garbage collector has to be called. As it is known compile time how many cells are actually needed in a certain basic block all these checks can be done at once instead of performing separate checks for each instruction (for an example see section 3.2).

- Optimising stack use: the generation of additional instructions for boundary checks of the stacks occurs inside a basic block can be avoided partially. For each block it can be determined how much the size of a stack will increase maximally which allows it to perform the testing in one time. The same counts for the adjustment of the stackpointer (due to the various `push` and `pop` instructions) which may involve additional `add` or `sub` instructions. Mostly these adjustments can be done on-the-fly using the post-increment or pre-decrement addressing modes. Another important improvement that might be obtained by using the post-increment or pre-decrement addressing modes is the disappearance of offsets.
- Compile time evaluation: since we are sure that a sub-sequence of instructions of a basic block can only be reached via the beginning of this block we are free to substitute such a sequence by any piece piece of code having the same result. This allows us to replace instructions inside a basic block by other instructions which are executed faster than the original ones or to evaluate certain sub-expressions compile-time.

3.4. The Code Generation Process

The translation ABC programs into MC68020 code is quite complex so it is not reasonable to consider it as occurring in a single step. For this reason it has been partitioned into a number of phases. An overview of these phases is given in the next picture:

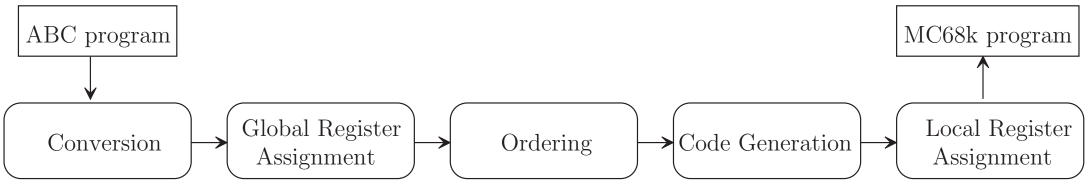

Fig 3.4: The code generation process.

During the *conversion phase* the initial ABC program is divided into so-called basic blocks and converted into an internal representation using graphs. The *global register assignment phase* determines which entries of the A and B stacks are kept in registers at the beginning and at the end of these basic blocks. The *ordering phase* determines of all the sub-expressions the order in which these expressions should be evaluated. During the *code generation phase* (pseudo) MC68020 code is generated according to the order specified by the previous phase. The only difference between real MC68020 code and the generated code is that in the latter an unlimited amount of (virtual) registers is assumed. Finally, the *local register assignment phase* replaces virtual registers by real MC68020 registers.

The conversion phase

The correspondence between the initial stack frame S_b at the beginning of a basic block b and the final stack frame F_b at the end is defined by the instructions of b. A useful data structure for representing basic blocks such that automatic analysis of these blocks can be done more conveniently is a directed acyclic graph (hereafter called a *dag*). Such a dag gives a picture of how the value computed by each statement in a basic block is used in subsequent statements in the block. The dag representation of a basic block that we use has the following properties:

- The leaves either represent constants or entries of S_b
- All the other (interior) nodes represent applications of ABC instructions. The arguments of these nodes are the representations of the arguments of the corresponding instructions.
- If a node represents an instruction whereof the result appears in F_b, this node is labelled with an identification of the corresponding entry in F_b. All the other nodes are not labelled.

The next example will serve as a guideline to illustrate the various phases of the ABC to MC68020 translation process. Consider the following Clean rewrite rule for F:

```
::   F !INT !INT    →   INT                    ;
     F a b          →   −I (*I a b) (+I a 3)   ;
```

This rule is compiled into the following ABC instructions (only the strict entry is given):

```
sF:
      pushI +3
      push_b 1          | push a on top of the stack
      addI              | add a and 3
      push_b 2          | push b on top of the stack
      push_b 2          | push a on top of the stack
      mulI              | multiply a and b
      update_b 1 3      | update the b stack
      update_b 0 2
      pop_b 2
      subI              | subtract the topmost elements
      rtn
```

The strict entry forms one basic block. The dag constructed according to this basic block is given in figure 3.5. The meaning of the additional information stored in the dag is explained later on.

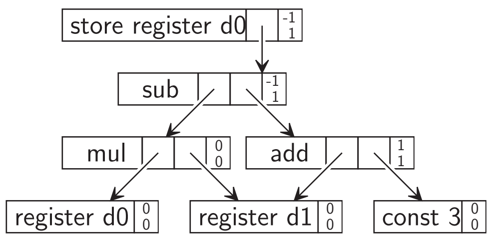

Fig 3.5: The dag of the strict entry.

Note that the (superfluous) copying actions of stack entries do not appear in the dag.

The global register assignment phase

The so-called global register assignment specifies which values of the initial and final stackframes of each basic block are kept in registers. The information that is used to determine this assignment is obtained from the original Clean program. The Clean compiler uses the type information of the Clean rules to insert special ABC directives in the corresponding ABC program. These directives are placed at the beginning and the end of the basic blocks and specify the layouts of both the A and B stacks.

In section 2.2 we already mentioned that parameters and results of functions are passed via the A and B stacks. To increase efficiency the topmost elements of both stacks are kept in registers. So with the aid of this information the code generator reserves all the data and address register the are needed for storing these elements at the beginning of a basic block if such a basic blocks starts with a label that corresponds to an entry point of a function. The directives at the end of a basic block are used to provide that when a basic block is left at run-time, the results are kept on the right places (i.e. either in one of the register or one of the final stackframes itself). The latter may require that the contents of certain registers in use have to be saved on the stack.

The result of the global register assignment of a basic block is administered in the corresponding dag with the aid of two kinds of special nodes, namely, *register* and *store_register* nodes. A register node, which refers to an entry of the initial stackframe S_b, indicates that the value of that entry is kept in a register when entering the basic block (note that register node are always leaves of the dag). A store_register node, which refers to an entry of the final stackframe F_b, indicates that the value of that entry is held in a register when leaving the basic block. So store_register nodes are always labelled. For example, consider figure 3.5: due to the global register assignment parameters a and b are kept in resp. data register d1 and d0 and the final result should be stored in register d0. These register assignments are indicated with the aid of special register and store_register nodes.

The ordering phase

After executing a basic block all the entries of the final stack frame of that block have to be defined. So, the task of the code generator becomes to generate code for all the dags that are labelled. Under the assumption[1] that none of the instructions in a basic block, except the very last instruction may produce side effects the generation of code can be done independently of the original order of ABC instructions. The only requirement that has to be met is that when generating code for a certain node of the dag, all the other nodes that are reachable from this node have already been treated. So, we are free to change the original evaluation order as long as the previous requirement is fulfilled.

The aim of the ordering phase is to produce an evaluation order for a basic block for which the number of used registers is minimal. If a basic block does not contain any common subexpression (*CSE*), so the corresponding dag is free of sharing, the problem

[1]From the (informal) definition of the notion basic block in this paper one could infer that the only instructions that have side effects are the instructions that can change the flow of control. This is not true. There are a few other ABC instructions which also have side effects and therefore have to be considered as instructions indicating the end of a basic block. An overview of these instructions can be found in Groningen (1990).

of determining the minimal number of registers is rather simple. An algorithm (which makes some assumptions about the registers and instructions of the target machine) has been given in Aho et al. (1986). The problem with common subexpressions is that the results of these expressions have to be stored somewhere until they are used the last time. This implies that after evaluating a certain sub-expression, the number of registers in use does not always increase by exactly one (due to the additional register necessary to hold the value of that sub-expression). It is possible that this increase is greater than one (if the dag contains CSE's that were not evaluated yet) or even smaller than one (if registers containing values of CSE's were used for the last time). Furthermore, the algorithm presented in Aho et al. (1986) cannot deal with values kept in registers at the beginning and at the end of a basic block. As with CSE's, such registers can be released as soon as their contents are not needed anymore.

To deal with CSE's when determining the evaluation order we first have to introduce some notions. A (rooted) connected dag (abbreviated as *cdag*) is defined as a dag that has a node r (called the root of the cdag) such that all the other nodes of this dag are reachable from r. Further, let n be the number of cdags that have to be evaluated and g_i denote the i^{th} cdag ($1 \leq i \leq n$). The evaluation order can be expressed by means of a permutation π of $1..n$ such that the cdag g_i is evaluated before a cdag g_j if $\pi^{-1}(i) < \pi^{-1}(j)$. Define $I(\pi,i)$ and $U(\pi,i)$ as:

$I(\pi,i)$ = the increase of the number of used registers due to the evaluation of $g_{\pi(i)}$ after evaluating $g_{\pi(1)}, \ldots, g_{\pi(i-1)}$.
$U(\pi,i)$ = the (additional) number of registers required to evaluate $g_{\pi(i)}$ (also after evaluating $g_{\pi(1)}, \ldots, g_{\pi(i-1)}$).

Note that $I(\pi,i)$ can be negative but $U(\pi,i)$ not and that $U(\pi,i) \geq I(\pi,i)$.

Given an evaluation order π, the maximum number of register in use during the evaluation of $g_{\pi(i)}$ is:

$$R(\pi,i) = U(\pi,i) + \sum_{k=1}^{i-1} I(\pi,k)$$

The number of registers necessary to evaluate all the graphs in an order specified by π is:

$$R_m(\pi) = \text{Maximum}\{R(\pi,i) \mid 1 \leq i \leq n\}$$

So finding an optimal evaluation order comes to the same as determining a permutation π_{min} such that for all other permutations π of $1..n$: $R_m(\pi_{min}) \leq R_m(\pi)$.

A straightforward algorithm would generate all permutations π of $1..n$ and choose the one for which $R_m(\pi)$ is minimal. Unfortunately, the complexity of such an algorithm is $O(n!)$ which is, of course, unacceptable.

The algorithm that we have implemented estimates the values of $I(\pi,i)$ and $U(\pi,i)$ on forehand (i.e. before determining the evaluation order) by resp. $I(i)$ and $U(i)$. We require that the estimations are save which implies that for all permutations π both $I(\pi,i) \leq I(i)$ and $U(\pi,i) \leq U(i)$ must be valid. After determining $I(i)$ and $U(i)$, the evaluation order of the cdags is given by the following two rules:

- First evaluate all the cdags g_i with $I(i) \leq 0$ from low to high $U(i)$;

- then evaluate all the other cdags from high to low $D(i)$, where $D(i)$ is defined as $U(i) - I(i)$.

It will be clear that the cdags g_i with $I(i) \leq 0$ have to be done first, for, after the evaluation of a graph with a non-positive I value some registers may become free. From the fact that we try to minimise the number of registers needed to evaluate the whole dag it immediately follows that these cdags should be treated in ascending $U(i)$ order. Why all the other cdags are ordered according to their D value is more difficult to see. We will to illustrate this with an example (a correctness proof can be found in Groningen (1990)): Suppose we have three graphs g_1, g_2, and g_3 with $I(1) = 1, U(1) = 2, I(2) = 1, U(2) = 2, I(3) = 1$ and $U(3) = 5$. When we start with g_3 we will need 5 registers but if we start with one of the other two graphs at least 6 registers are necessary.

In Groningen (1990) these rules have been further refined. It also has been shown that the evaluation order gives a good approximation of the optimal order.

Registers on a MC68020

The two types of registers of the MC68020 processor are not generally exchangeable. Furthermore, not all registers are freely available: some of registers are dedicated to a special task. Beause of this, the number of address registers that we may use differs from the number of data register. Both facts make it necessary to adapt the algorithm described above. This is done in two stages. First, for each node of the dag it is decided whether this node is computed in an address register or in a data register. After that we determine for each cdag g_i two values of $I(i)$: one for each kind of register. Now, the total increase $I_t(i)$ is defined as:

$$I_t(i) = a * I_d(i) + d * I_a(i)$$

where a, I_a, d, I_d are respectively the number of address registers, the increase of address registers, the number of data registers and the increase of data registers. In the same way we define $U_t(i)$ as:

$$U_t(i) = a * U_d(i) + d * U_a(i)$$

The evaluation order is obtained by applying the previous algorithm using the functions I_t and U_t instead of I and U.

Consider figure 3.5: each node is supplied with two numbers whereof the uppermost gives the I_d value of this node and the other number gives the U_d value. Since the I_a and U_a values do not matter they have been omitted. The negative I values (of the sub and the uppermost register nodes) are a consequence of the fact that the contents of the register d1 is not needed anymore after evaluating these nodes. The result of the mul node can be stored in register d0. So, in contrast with the add node, no additional registers are needed to compute the result of this node.

The code generation phase

During the code generation phase the cdags of all the basic blocks are traversed. The order wherein all the cdags are treated is specified by the ordering phase. For each cdag, code is generated in a depth-first way: First, code is generated (recursively) for all of the arguments of a node (again in an order as specified by the ordering phase). Then the operation specified by the node is translated into one or more (pseudo) MC68020 instructions, assuming an unlimited amount of (virtual) data and address register.

The local register assignment phase

Finally, during the last phase of the translation real registers are assigned to the virtual registers allocated by the code generation phase. The algorithm that determines the evaluation order tries to minimise the number of data and address registers that are needed to evaluate a basic block. However, it may be the case that one of these numbers exceeds the number of registers that are actually available (i.e. the number of virtual registers exceeds the number of real registers). In that case it will be necessary that at some point in the code the contents of a register of the required type has to be saved in memory so that it can be used again. The problem is which register has to be freed? The strategy we use is to choose the register of the required type whereof the contents will not be used for the longest time.

Again using figure 3.5: during the last two phases code is generated for this dag. Actually, the very last phase is superfluous: there are enough registers available for allocating all the virtual registers. The final result of the code generation is shown below:

```
sF:     muls.l      d1,d0
        addq.l      #3,d1
        sub.l       d1,d0
        rts
```

3.5. Other Improvements

Enlarging basic blocks

The gain in efficiency that can be achieved depends on the size of the basic blocks: large blocks offer more optimisation possibilities than small ones. In this section we describe two methods that can be used to extend basic blocks.

- Jsr_eval instructions: jsr_eval instructions are translated into sequences of MC68020 instructions which, amongst other things, investigate whether the corresponding node contains a descriptor or a pointer to the evaluation code. Since a jsr_eval instruction may change the flow of control it seems that a basic block containing such an instruction has to end at this point. However, this is not needed when the node that is going to be reduced is already in root normal form. An example of this can be found in section 3.2 in which the used registers are saved only when the evaluation code of the node is called.

- Code substitution: When calling a subroutine the current basic block is left. This, however, can be avoided if the subroutine call is substituted by the corresponding code. Note that this is only reasonable when the substituted code is relatively small. This optimisation is not yet implemented in our code generator (Though some code substitution mechanism has been implemented in the Concurrent Clean to ABC code compiler (Eekelen et al. (1990))).

Optimising booleans

In the ABC machine the conditional jump instructions (i.e. jmp_true and jmp_false) base their decision whether or not to jump on the (boolean) value that is on top of the B stack. If this boolean is the result of some comparison (which is indeed often the case) then it is not necessary to calculate this value explicitly. Instead, the conditional jump can use the condition codes of the concrete target machine that are set implicitly after performing the comparison. Take for example the following ABC instructions:

```
eql_b        +10 0
jmp_false    label
```

The following code would have been generated if the boolean is calculated explicitly (assumed the top of the B stack is kept in register d0, register d1 is used to calculate the boolean and the value -1 and 0 represent resp. TRUE and FALSE):

```
cmpi.l       #10, d0
seq          d1
extb.l       d1
tst.l        d1
beq          label
```

Much better code is generated when using the condition codes of the MC68020:

```
cmpi.l       #10, d0
bne          label
```

Using alternative instructions

On the MC68020 there are several instructions that, when applied to certain arguments, can be replaced by other instructions having the same effect but, with the advantage that their execution time is less. This replacement can easily be performed during the last phase of the MC68020 code generation or even by the assembler. Below examples of substitutions that have been implemented are given together with the gain in efficiency:

- move.l #data, Dn can be replaced by moveq #data, Dn when $-128 \leq \text{data} \leq 127$. The gain is about 65%.

- muls.l #data, Dn can be replaced by lsl #n, Dn when data=2^n. In this case the gain is 90%

The efficiency can also be increased by replacing instructions by combinations of other instructions with the same effect. Examples:

- cmpi.l #100,d0 can be substituted by moveq #100,d1 followed by cmp.l d1,d0.

- The same counts for muls.l #10,d0 and the instructions move.l d0,d1, lsl.l #2,d1, add.l d1,d0, add.l d0,d0. The gains in efficiency are resp. about 30% and 75%.

Note that those alternative combinations of instructions need an extra data register. So one has to be sure that such a register is available.

4. Performance

We have compared the implementation of Concurrent Clean with implementations of Lml, Hope and C on the SUN3 (with a MC68020, 25Mhz processor). The Lml system is considered as a standard implementation of a lazy functional language. The Hope system is an example of a fast implementation of a strict functional language. The imperative languages are represented by C. It should be stated that, if possible, C has been used in an imperative way (i.e. using iteration instead of recursion). In our tests, the following implementations of these languages were used:

Lml	The Chalmers Lazy ML compiler, version 0.99.2, (90/08/20) (Augustsson and Johnsson (1989)).
Hope	The Hope^{+} compiler, release 3.2.1, August 1989 (Burstall et al. (1980)).
C	The gnu C compiler, version 1.36 (which generally gives faster code than the standard C compiler).

Measurements were done for the following test programs (see also the Appendix):

nfib	the well known nfib program with argument 30.
tak	the Takeuchi function, called with (tak 24 16 8).
sieve	a program which generates the first 10000 primes, using quite an optimal version of the sieve of Eratosthenes (outputs only the last one).
queens	counts all solutions for the (10) queens problem.
reverse	a program which reverses a list of 3000 elements 3000 times.
twice	four times the twice on the increment function.
revtwice	four times the twice of the reverse of a list of 30 elements.
rnfib	again the nfib program, but now working on real numbers, with argument 26.
fastfourier	the fast fourier algorithm, on an array of 8K complex numbers. In the Concurrent Clean program a complex number is defined as a strict tuple of two reals.

	Clean	Lml	Clean (u!)	Hope	C	Clean (-!)
nfib	4.5	25	4.5	5.4	11	30
tak	4.9	40	4.9	7.2	11	36
sieve	8.1	25	6.8	9.1	4.5	12
queens	28	62	14	16	4.1	45
reverse	64	108	50	65	–	51
twice	1.7	SF	0.5	0.3	–	1.7
revtwice	27	OH	9	12	–	39
rnfib	11	26	11	33	19	19
fastfourier	34	–	19	–	9.0	–

Table 4.1 Performance Overview (All times in seconds cpu time).

The following notes have to be made:

- The Lml versions of twice and revtwice resulted in run-time errors for these values (SF and OH stand for 'segmentation fault' and 'out of heap' respectively).

- The reverse and twice programs have no sense in the context of C. The sieve and fast fourier programs are iterative versions. The other ones are inherently recursive.
- There is no point in computing the fast fourier with the other functional languages: they all would run out of heap space.
- The times needed to generate an executable for the example programs vary widely. On an average, the Concurrent Clean implementation consumes about 3.5 seconds cpu time, the Lml system needs 6 seconds and the Hope system even 15 seconds.

The first two columns of the table compare a standard compilation of Concurrent Clean programs with Lml. The default reduction strategy is lazy, but strictness information is added automatically by the strictness analyser. It is obvious that in all cases Concurrent Clean outruns Lml.

The next two columns present a comparison between user annotated Clean, and Hope. User annotations are inserted at some places that are not indicated by the strictness analyser (see appendix A). The result is that functions become strict in more arguments (e.g. the sieve and queens programs), or return strict tuples (e.g. the fast fourier example). Concurrent Clean produces in almost all the cases the fastest code although the difference compared with Lml is not that great anymore. The only case in which Hope is faster is the twice example. This is mainly because Hope uses a smart integer representation. This is indicated by the revtwice program, which also tests the implementation of higher order functions but avoids the use of integers.

The recursive programs written in C appear to be slower than the ones written in Concurrent Clean. However, the iterative versions of the examples written in C are faster. But, in comparison with the past, the difference between execution times of on the one hand the functional languages and on the other hand the imperative languages has significantly decreased.

The last two rows of the table are measurements for real arithmetic. In fact, they show that of the functional languages only Concurrent Clean supports reals seriously.

Finally, the last column gives execution times for Concurrent Clean programs for which no annotations were added, neither automatically by the strictness analyser, nor by the programmer herself. From these figures we can conclude that in general strictness annotations increase the efficiency. The largest gain is achieved in programs which largely manipulate objects of basic types as is the case with tak and fast fourier.

5. Conclusions and Future Work

The figures presented in the previous section show that it is possible to obtain efficient implementations of lazy functional languages on conventional architectures. The division of the complicated compilation process into several stages has been proven to be fruitful. The first stage, wherein Concurrent Clean programs are translated into ABC code allowed us to concentrate our attention fully on the problem of how to avoid graph manipulations. During the second stage, in which the generated ABC instructions are grouped into basic blocks and translated to real target code, it appeared that all information needed to do this efficiently is still available in the ABC program; no essential information has been lost during the first stage. The performance comparison shows that optimisations presented in this paper give a significant increase in speed. The differences in speed between programs on the one hand written in Concurrent Clean and, on the other hand, written in imperative languages like C are now becoming acceptable.

However, the efficiency of the generated code can still be improved. With respect to the Concurrent Clean translation for instance, a so-called "application depended strictness analysis" can be added to the system. Such an analysis tries to determine whether eager evaluation of arguments for a certain application is safe because for this specific application it is known that these arguments will have been evaluated (inspite of the fact that the applied function is not known to be strict in these arguments for the general case). To the ABC to MC68020 code generator a more comprehensive global register allocator will be added. Also, the parameter passing of reals will be altered by using the floating point registers. Finally, the evaluation of lazy expressions and the implementation of the heap will be improved further and the in-line code substitution mechanism is going to be implemented.

Besides working on a sequential implementation, we are also developing a parallel implementation of Concurrent Clean on transputers (Kesseler (1990)). At the moment a preliminary version is already available which allows interleaved multi-processing on a single transputer. This parallel system will be extended in the near future such that real speed-ups can be demonstrated. At UEA already some promising results have been obtained with a previous version of our Clean system (McBurney and Sleep (1990)).

Appendix

```
||
||   The NFib Benchmark
||

::   Nfib INT        →  INT                                          ;
     Nfib n          →  IF   (<I n 2)
                             1
                             (++I (+I (Nfib (–I n)) (Nfib (-I n 2))))   ;

::   Start           →  INT                                          ;
     Start           →  Nfib 30                                      ;

||
||   The Takeuchi Benchmark
||

::   Tak INT INT INT    →  INT                          ;
     Tak x y z          →  IF   (<=I x y)
                                z
                                (Tak (Tak (–I x) y z)
                                     (Tak (–I y) z x)
                                     (Tak (–I z) x y))  ;

::   Start              →  INT                          ;
     Start              →  Tak 24 16 8                  ;

||
```

```
||    The Sieve Benchmark
||

::    Start        →   INT                                      ;
      Start        →   Select [2 | [3 | Primes]] 10000          ;

::    Primes       →   [INT]                                    ;
      Primes       →   primes: [5 | Sieve 7 4 primes]           ;

::    Sieve INT INT [INT]    →   [INT]                                          ;
      Sieve g i prs          →   IF   (IsPrime prs g (RTOI (SQRT (ITOR g))))
                                      [g | Sieve' g i prs]
                                      Sieve (+I g i) (-I 6 i) prs)              ;

::    Sieve' INT INT [INT]   →   [INT]                                          ;
      Sieve' g i prs         →   Sieve (+I g i) (-I 6 i) prs                    ;
```

```
::  IsPrime [INT] INT INT     →  BOOL                          ;
    IsPrime [f | r] pr bd     →  IF   (>I f bd)
                                      TRUE
                                      (IF (=I (MOD pr f) 0)
                                          FALSE
                                          (IsPrime r pr bd))   ;

::  Select [x] INT            →  x                                        ;
    Select [f | r] 1          →  f                                        |
    Select [f | r] n          →  Select r (–I n)                          ;

||
||  The Queens Benchmark
||

::  Start          →  INT                           ;
    Start          →  Length (Queens 1 [ ] [ ]) 0   ;

    BoardSize      →  10                            ;

::  Queens INT [INT] [[INT]]    →  [[INT]]                                        ;
    Queens row board boards     →  IF   (>I row BoardSize)
                                        [board | boards]
                                        (TryCols BoardSize row board boards)      ;

::  TryCols INT !INT [INT] [[INT]]        →  [[INT]]      ;
    TryCols 0 row board boards            →  boards       |
    TryCols col row board boards          →
        IF   (Save col row board (–I row))
             (TryCols (–I col) row board queens)
             (TryCols (–I col) row board boards),
        queens: Queens (++I row) [col | board] boards     ;

::  Save !INT !INT [INT] INT      →  BOOL                                ;
    Save c1 r1 [ ] 0              →  TRUE                                |
    Save c1 r1 [c2 | cols] r2     →  IF   (Check r1 r2 (-I c1 c2))
                                          FALSE
                                          (Save c1 r1 cols (–I r2))      ;

::  Check !INT !INT INT           →  BOOL                                ;
    Check r1 r2 0                 →  TRUE                                |
    Check r1 r2 cdiff             →  IF   (=I cdiff (-I r1 r2))
                                          TRUE
                                          (=I cdiff (-I r2 r1))          ;

::  Length [x] INT                →  INT                                 ;
    Length [a | r] n              →  Length r (++I n)                    |
    Length [ ] n                  →  n                                   ;

||
||  The Reverse Benchmark
||
```

```
::  Rev_n INT [INT]      →  [INT]                            ;
    Rev_n 1 list         →  Rev list [ ]                     |
    Rev_n n list         →  Rev_n (–I n) (Rev list [ ])      ;

::  Rev [INT] [INT]      →  [INT]                            ;
    Rev [x | r] list     →  Rev r [x | list]                 |
    Rev [ ] list         →  list                             ;

::  Reverse INT          →  INT                              ;
    Reverse n            →  Walk (Rev_n n (FromTo 1 n))      ;

::  Walk [INT]           →  INT                              ;
    Walk [x]             →  x                                |
    Walk [x | r]         →  Walk r                           ;

::  FromTo INT INT       →  [INT]                            ;
    FromTo a b           →  IF   (=I a b)
                                 [a]
                                 [a | FromTo (++I a) b]      ;

::  Start                →  INT                              ;
    Start                →  Reverse 3000                     ;

||
||  The Twice Benchmark
||

::  Start                →  INT                                  ;
    Start                →  Twice Twice Twice Twice ++I 0        ;

::  Twice (=> x x) x     →  x                                    ;
    Twice f x            →  f !(f x)                             ;

||
||  The RevTwice Benchmark
||

::  RevList [INT]   →  [INT]                                                 ;
    RevList l       →  Rev l [ ]                                             ;

::  Start           →  [INT]                                                 ;
    Start           →  Twice Twice Twice Twice RevList (FromTo 1 30)         ;

||
||  The RNFib Benchmark
||

::  RNfib REAL      →  REAL                         ;
    RNfib n         →  IF   (<R n 1.5)
                            1.0
                            (+R (+R (RNfib (-R n 1.0))
                                 (RNfib (-R n 2.0)))
                                 1.0)               ;
```

```
::  Start                 →  REAL                          ;
    Start                 →  RNfib 26.0                    ;

||
||  The FastFourier Benchmark
||

TYPE

::  Complex                      →  (!REAL,!REAL)              ;

RULE

::  +C Complex Complex       →  Complex                    ;
    +C (a1, b1) (a2, b2)     →  (+R a1 a2, +R b1 b2)       ;

::  -C Complex Complex       →  Complex                    ;
    -C (a1, b1) (a2, b2)     →  (-R a1 a2, -R b1 b2)       ;

::  *C Complex Complex       →  Complex                    ;
    *C (a1, b1) (a2, b2)     →  (-R (*R a1 a2) (*R b1 b2),
                                 +R (*R a1 b2) (*R b1 a2))  ;

::  Root INT INT   →  Complex                                  ;
    Root j n       →  (COS z, SIN z),
                      z: /R (*R (ITOR j) 6.2831853) (ITOR n)   ;

    FastFourier com_array length           →  Fast com_array length      ;

::  Fast [Complex] INT       →  [Complex];
    Fast com 1               →  com                                  |
    Fast com length          →  Merge res_even res_odd length,
                                (even, odd): Split com,
                                res_even: Fast even next_length,
                                res_odd: Fast odd next_length,
                                next_length: /I length 2             ;

::  Merge [Complex] [Complex] INT      →  [Complex]                              ;
    Merge even odd n                   →  Append low high,
                                          (low, high): Merge2 even odd n 0       ;

::  Append [x] [x]          →  [x]                        ;
    Append [a | x] y        →  [a | Append x y]           |
    Append [ ] y            →  y                          ;

::  Merge' [Complex] [Complex] !INT !INT      → (![Complex], ![Complex])          ;
    Merge' [e | re] [o | ro] n i      →  ([!ui | urest], [!umi | umrest]),
                                         (urest, umrest): !Merge' re ro n (++I i),
                                         ri: Root i n,
                                         prod: *C ri o,
                                         ui: +C e prod,
                                         umi: -C e prod                          |
    Merge' [ ] [ ] n i                →  ([ ], [ ])                              ;
```

```
::  Split [Complex]          →  (![Complex], ![Complex])     ;
    Split [a | [b | rest]]   →  ([a | even], [b | odd]),
                                (even, odd): !Split rest     |
    Split [ ]                →  ([ ], [ ])                   ;
```

References

Aho, A.V., R. Sethi and J.D. Ullman (1986). *Compilers, Principles, Techniques and Tools*, Bell Telephone Laboratories, Incorporated, Addison-Wesley.

Augustsson, L. and T. Johnsson (1989). The chalmers lazy-ml compiler, *The Computer Journal*.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep (1987). Term graph reduction, *Proc. of Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, The Netherlands, Springer Verlag, LNCS 259 II, pp. 141–158.

Barendsen, Erik and Sjaak Smetsers (1992). Graph rewriting and copying, *Technical Report 92-20*, University of Nijmegen.

Brus, T., M.C.J.D. van Eekelen, M. van Leer, M.J. Plasmeijer and H.P. Barendregt (1987). Clean - a language for functional graph rewriting, *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, Portland, Oregon, USA, Springer Verlag, LNCS 274, pp. 364–384.

Burstall, R.M., D.B. MacQueen and D.T. Sanella (1980). Hope: An experimental applicative language, *Proc. of The 1980 LISP Conference*, pp. 136–143.

Eekelen, M.C.J.D. van (1988). *Parallel Graph Rewriting, Some Contributions to its Theory, its Implementation and its Application*, Dissertation, University of Nijmegen.

Eekelen, M.C.J.D. van, E.G.J.M.H. Nöcker, M.J. Plasmeijer and J.E.W. Smetsers (1990). Concurrent clean, version 0.6, *Technical Report 90-21*, University of Nijmegen.

Eekelen, M.C.J.D. van, M.J. Plasmeijer and J.E.W. Smetsers (1991). Parallel graph rewriting on loosely coupled machine architectures, *in:* Kaplan and Okada (eds.), *Proc. of Conditional and Typed Rewriting Systems (CTRS'90)*, Montreal, Canada, Springer Verlag, LNCS 516, pp. 354–369.

Groningen, J.H.G. van (1990). *Implementing the abc-machine on m680x0 based architectures*, Master's thesis, University of Nijmegen.

Johnsson, Th. (1987). *Compiling Lazy Functional Programming Languages*, Dissertation, Chalmers University, Göteborg, Sweden.

Kesseler, M. (1990). Concurrent clean on transputers.

Koopman, P.W.M., M.C.J.D. van Eekelen, E.G.J.M.H. Nöcker, M.J. Plasmeijer and J.E.W. Smetsers (1990). The abc-machine: A sequential stack-based abstract machine for graph rewriting, *Technical Report 90-22*, University of Nijmegen, The Netherlands.

McBurney, D. and R. Sleep (1990). Concurrent clean on zapp, *Proc. of Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, University of Nijmegen, the Netherlands, Technical Report 90-16.

Milner, R.A. (1978). Theory of type polymorphism in programming, *Journal of Computer and System Sciences*.

Motorola (1985/6). *MC68020 32-Bit Microprocessor User's Manual*, Prentice Hall.

Mycroft, A. (1984). Polymorphic type schemes and recursive definitions, *Proc. of 6th Int. Conf. on Programming*, Eindhoven, The Netherlands, Springer Verlag, LNCS 167, pp. 217–239.

Nöcker, E.G.J.M.H. (1989). The pabc simulator, v0.5. implementation manual, *Technical Report 89-19*, University of Nijmegen.

Nöcker, E.G.J.M.H. (1990). Strictness analysis based on abstract reduction, *Proc. of Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, University of Nijmegen, the Netherlands, Technical Report 90-16, pp. 297–321.

Nöcker, E.G.J.M.H. and J.E.W. Smetsers (1990). Partially strict data types, *Proc. of Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, University of Nijmegen, the Netherlands, Technical Report 90-16, pp. 237–255.

Nöcker, E.G.J.M.H., J.E.W. Smetsers, M.C.J.D. van Eekelen and M.J. Plasmeijer (1991). Concurrent clean, *Proc. of Parallel Architectures and Languages Europe (PARLE'91)*, Eindhoven, The Netherlands, Springer Verlag, LNCS 505, pp. 202–219.

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

Plasmeijer, M.J. and M.C.J.D. van Eekelen (1993). *Functional Programming and Parallel Graph Rewriting*, Addison Wesley, University of Nijmegen, The Netherlands.

Smetsers, J.E.W. (1989). Compiling clean to abstract abc-machine code, *Technical Report 89-20*, University of Nijmegen.