FCG: a Code Generator for Lazy Functional Languages

Koen Langendoen Pieter H. Hartel

University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
e-mail: *koen@fwi.uva.nl*

Abstract. The FCG code generator produces portable code that supports efficient two-space copying garbage collection. The code generator transforms the output of the FAST compiler front end into an abstract machine code. This code explicitly uses a call stack, which is accessible to the garbage collector. In contrast to other functional language compilers that generate assembly directly, FCG uses the C compiler for code generation, providing high-quality code optimisations and portability. To make full use of the C compiler's capabilities, FCG includes an optimisation scheme that transforms the naively generated stack-based code into a register-based equivalent form. The results for a benchmark of functional programs show that code generated by FCG performs well in comparison with the LML compiler.

1 Introduction

Functional languages offer the programmer referential transparency, and increased expressiveness in comparison with ordinary imperative languages: higher order functions, lazy evaluation (streams), and built-in storage management (automatic garbage collection). These extras, however, place an extra burden on the implementation which results in more complex compilers and runtime support systems, while the compiled programs usually execute slower than their imperative counterparts.

One popular method for compiling functional languages is to make maximal use of existing imperative compiler technology by constructing a front end that translates a functional program into imperative code (e.g., C) [11, 10]. Higher order functions and lazy evaluation are typically handled by calling support functions which manipulate heap allocated closures that hold a function identifier and some arguments. The quality of the generated code heavily depends on the front end's strictness analysis that minimises the inefficient usage of these closures.

Although the "generate-C" method requires minimal implementation effort, the C-compiler hampers performance of the resulting object code because it prohibits control over (heap) pointers that are stored in the processor's registers and stack. This control is crucial for two reasons:

- Efficient garbage collection algorithms like two-space copying and generation scavenging require all pointers into the heap to be known to the garbage collector because objects are moved and pointers have to be adjusted accordingly.
- Frequently accessed pointers like the pointer to the start of free heap space should be stored in global registers.

For maximal performance we need intimate knowledge of the location of pointers on the calling stack and in registers. Therefore several functional language compilers have adopted the do-it-yourself method of generating assembly code directly [8, 12, 9]. This alternative approach gives total control over all pointers and the processor, but it goes against the grain of the *lazy* implementor because now we have to deal with important low-level issues like register allocation and code scheduling, while this can be done perfectly well, and probably better, by (part of) an existing C compiler. Besides implying extra work, this "generate-assembly" method looses on portability as well, since C compilers are available for almost any type of computer.

This paper describes the code generator in a functional language compiler that combines the advantages of both previous approaches: it compiles down to a level where it has control over the location of pointers and then uses part of the C compiler to generate object code. We have made use of the existing FAST front end [5], which includes an advanced strictness analyser. The front end translates a functional program into a severely restricted subset of C, which is called Functional C, with standard call-by-value semantics. Since pointers are passed as ordinary parameters, direct compilation of Functional C results in code that can not be used in combination with moving garbage collectors. Therefore the Functional C Code Generator (FCG) compiles the FAST output further to code (KOALA) that uses an explicit call stack, which brings all pointers under control of the garbage collector. The complete compiler is organised as a pipe-line of three programs:

1. The Fast front end translates functional programs to Functional C, thereby making the functional expressiveness explicit by inserting calls to support functions as with the "generate-C" approach.
2. The FCG code generator compiles the FAST output to the KOALA assembly frame work that supports features like register allocation, code optimisations, and code scheduling.
3. The final phase consists of assembling KOALA into machine code, using the GNU gcc compiler to do the hard work.

The source and target languages of the FCG code generator, Functional C and KOALA respectively, are discussed in sections 2 and 3. Section 4 then presents the kernel compilation schemes of the FCG code generator, while Section 5 contains some mandatory optimisations to achieve quality code. The performance effects of those optimisations are presented in Section 6, which also includes a comparison with other compilers to assess the absolute performance of code generated by FCG.

2 Functional C

The FAST compiler, which has been developed at Southampton University in the UK, forms the first stage in our compiler pipe-line. It accepts programs with lazy semantics, which are written in a small lazy functional language called *intermediate*. This language is intended to serve as an intermediate between a general purpose lazy functional language, such as Haskell, and an optimising backend.

Function definitions in *intermediate* have the form of a set of recursive equations with the possibility to express list pattern matching on function arguments. **[]** represents the empty list and the cons operator is denoted by the colon (**:**). The language is higher order, curried, and lazy.

The function *append* in Figure 1 gives an example of an *intermediate* program. It uses list pattern matching on its first argument to decide whether to recurse on the tail of the list, or to return the second argument.

```
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Figure 1: Append in *intermediate*

The FAST compiler produces as output equivalent Functional C programs with call by value semantics. Functional C serves as the source language for the FCG code generator and is essentially a subset of C, see the syntax in Figure 2. The main restrictions are directly related to the functional style of programming: single assignment of local variables, no global variables, and if-then-else as the only control structure. Functional C supports only one type, namely **ptr**, which may be a basic value or a pointer into the heap, and relies on the primitive functions to correctly interpret their operands. For example, *add_i* operates on integers, while *and_b* uses booleans. As a consequence all types in Functional C have the same size; therefore data structures are represented as a pointer to a sequence of fields in the heap.

program	::=	function$_1$ $\cdots$ function$_p$
function	::=	**ptr** id**(**id$_1$**,** $\cdots$ **,** id$_f$**)** decl {decl body}
decl	::=	**ptr** id$_1$**,** $\cdots$ **,** id$_d$**;**
body	::=	assignment **;** body
		| **return** expr **;**
		| **if (**expr**)** {body$_t$ } **else** {body$_e$ }
assignment	::=	id = expr
expr	::=	id**(**expr$_1$**,** $\cdots$**,** expr$_e$**)**
		| id**[**num**]**
		| id

Figure 2: Functional C

The Functional C code for *append* as shown in Figure 3 has essentially the same structure as the program in Figure 1. The important difference is that the implicit laziness of the *intermediate* version is now explicit in the form of the calls to the library functions *reduce* and *vap* (for vector application). The latter builds a suspension of a function (*append* in this case) in the heap, and the former evaluates a previously built suspension. Thus all

```
ptr append( x_xs, ys)
ptr x_xs, ys;
{
  ptr x, xs;

  if( null(x_xs)) {
    return reduce(ys);
  }
  else {
    x  = x_xs[0];    /* head */
    xs = x_xs[1];    /* tail */
    return cons(x,vap(prel_append,xs,ys));
  }
}
```

Figure 3: Append in Functional C

functions present in Figure 3 can safely and efficiently be called with call by value semantics, as is the usual case in C programs.

The FAST compiler generates Functional C code based on the principle that the callee decides what form its arguments should have. Some functions require arguments that are evaluated, while other arguments may be passed as is. The null test for example is strict in its argument, because *null* needs to inspect the argument to see whether or not it represents the end of a list. The principle that the callee decides on the form of its arguments has the interesting consequence that functions are not embedded directly in a suspension, but via another "prelude" function. When a suspension is evaluated, the prelude function first calls reduce for every strict argument before calling the function proper. In case of *append* the prelude function ensures that the strict first argument is indeed evaluated before entering *append*. The non-strict second argument is merely passed on.

The presence of calls to *reduce* requires runtime interpretation of the graph that resides in the heap. This is a much less efficient way of evaluating an expression than obeying straight sections of C code. The FAST compiler makes strenuous attempts to avoid interpreting the graph whenever possible, so there are far fewer occurrences of *reduce* than a naive front end would generate. The compiler employs a host of other analyses to further improve the quality of the Functional C code that is the input to the FCG code generator.

3 KOALA

KOALA is a high-level "assembly" frame work that serves as the target language for the FCG code generator. It provides a simple abstract machine suitable for graph reduction, similar to the well known G-machine [8], and consists of a cpu, an *unlimited* number of registers, a stack, and (heap) memory. KOALA's instructions are listed in Figure 4.

The KOALA stack is used to implement the function call mechanism: parameters are passed via the stack and the local state of a function is saved on the stack when calling a (recursive) function. The return address is passed as an extra parameter that will be used as branch destination on function return. This simple calling sequence does not include a frame pointer, so each function has to **squeeze** the call stack into a proper state before returning to its caller. If the result value is returned via a register then this amounts to just popping some items off the stack; this is cheaper than maintaining a frame pointer, which has to be saved and restored on function calls. To support easy integration into a parallel implementation, we have restricted KOALA to one single stack that combines the multiple stacks found in other abstract reduction machines.

For simplicity the heap can only be accessed via basic **load** and **store** instructions that transfer one word between a register and memory. In particular there are no high level instructions to allocate heap cells because the FCG code generator will perform certain optimisations on heap bound checks like inlining and clustering that would make those tests redundant inside the allocate instruction.

The minimal set of control instructions provides enough functionality to implement the function call/return sequence and the if-then-else construct present in Functional C. The remaining instructions that do the actual arithmetic computations, logical operations, etc. are provided all at once with the parameterised **alu** instruction, which takes the specific operation as it first argument.

The description of the KOALA instruction set contains no notion of data types; every item is regarded as

fetch $n\ reg$	fetch the n-th stacked value into register reg $(n \geq 1)$.
push id	push id on the stack; id refers to a register or constant.
pop reg	pop the value on top of the stack into register reg.
dup n	duplicate the n-th stacked value on top of the stack.
squeeze $n\ m$	slide down the top n elements of the stack, squeezing out the m below.
store $reg_{src}\ reg_{addr}$	store the contents of reg_{src} in memory at the location specified in reg_{addr}.
load $reg_{addr}\ reg_{dest}$	load one word at location reg_{addr} from memory into register reg_{dest}.
label lbl	instruction label (pseudo instruction).
branch id_{dst}	unconditional branch; id_{dst} is (the contents of) a register or a label.
bfalse lbl	pop boolean from the stack and branch to lbl if it is false.
jfalse $reg\ lbl$	jump if the boolean value in register reg is false.
fun $name$	function entry point (pseudo instruction).
call fun	branch to function fun.
return	return to caller; pop return address from the stack.
move $id_{src}\ reg$	move id_{src} to register reg; id_{src} refers to a register or constant.
alu $op\ id_1 \cdots id_n\ reg$	parameterised n-operand instruction: reg = $op\ id_1 \cdots id_n$ op is a basic alu operation: add, mul, etc. id_i is either a constant or a register.

Figure 4: KOALA's instruction set

some value with the wordsize of the underlying machine, which corresponds both to Functional C's uniform usage of the **ptr** type, and the internal working of modern RISC style processors. The unlimited number of (virtual) registers makes KOALA different from traditional assemblers.

Implementation

All KOALA instructions that manipulate the stack, like **push** and **pop**, can be expanded straightforwardly into a few kernel instructions (**alu** + **load**/**store**), which operate with a fixed top-of-stack register. The subset of KOALA that then remains, matches with the traditional intermediate code of imperative compilers like the three-address code described in [1]. Unfortunately not many compilers are capable of reading-in some intermediate code file, let alone that there exists a universally agreed upon format. Therefore we have taken the detour of translating KOALA back to C, which will then (again) be translated into some intermediate code by the C compiler itself.

A nuisance with using C as a sophisticated assembly language is that standard C does not support code labels properly: it forbids the usage of labels in expressions. This makes it impossible to directly push a (C) label as return address on the stack (i.e. store it in memory). The work-around is to use one level of indirection: KOALA labels are encoded as integers, and branches are translated to indirect gotos:

KOALA	C
label *lbl*	`lbl:`
branch *lbl*	`goto lbl;`
branch *reg*	`dest = reg; goto jump;`

	`jump: switch (dest) {`
	`  case 0: goto lbl_0;`
	`  case 1: goto lbl_1;`
	`  ⋮`
	`  }`

Note that only branches with a register target suffer this indirection. Examination of SPARC assembly code showed that such an indirect branch expands into 9 sparc instructions. Hand patching of indirect to direct jumps in the assembly code of the function-call intensive nfib benchmark program, reduces the runtime to 70%. For "real" programs that contain large basic blocks, however, the difference will be considerably less.

Our KOALA-in-C implementation translates a complete KOALA program into one single C function. This stresses most C compilers since they usually generate code for a procedure at once, but in return our method produces "globally" optimised code. It is, of course, possible to disassemble KOALA into C style functions,

but this would introduce inefficiencies like maintaining an explicit pointer stack for the garbage collector. Experience with the SUN and GNU C-compilers has shown that the SUN compiler with optimisations enabled gives up on large programs due to swapping problems, while the GNU compiler on the contrary executes faster with optimisations asserted than without.

The generated object code for the SPARC matches well with the KOALA source; in particular the GNU C compiler manages to assign KOALA's virtual registers to the sparc processor's physical registers without spilling values to the C stack.

4 FCG: Compiling Functional C

The task of FCG is to compile Functional C into KOALA assembly such that frequently accessed pointers are allocated in registers and all heap pointers can be accessed during garbage collection. The former requirement amounts to allocating the start-of-free-space and end-of-heap pointers into fixed KOALA registers, while the latter is accomplished by saving all local state on the KOALA stack on function calls. The stack is scanned by the (two-space copying) garbage collector to find all root pointers into the heap. To enable the discrimination between pointers and other data values (e.g., integers), all values contain tag information in the least significant bits as will be discussed in a following section.

In figure 5 the following three compilation schemes are used to show how FCG implements the function-call mechanism in KOALA:

$\mathcal{K}[\![\ function\ ]\!]$ The top-level scheme generates code for a function definition.

$\mathcal{R}[\![\ body\ ]\!]\ \rho\ d$ The $\mathcal{R}$eturn scheme generates code to return the value produced by $body$, where d is the current depth of the stack frame, and ρ is an association list (symbol table) that maps variables to their location in the frame.

$\mathcal{E}[\![\ expr\ ]\!]\ \rho\ d$ The $\mathcal{E}$xpression scheme generates code to compute the head normal form of $expr$. It puts (a pointer to) the value on top of the stack.

(0)	$\mathcal{K}[\![$ fun(id$_1$, $\cdots$, id$_n$) decl {decl body} $]\!]$ =	**fun** fun$_{n+1}$; $\mathcal{R}[\![$ body $]\!]$ [<cont,n>,<id$_1$,n−1>,$\cdots$,<id$_n$,0>] (n+1)	
(1)	$\mathcal{R}[\![$ id = expr; body $]\!]$ ρ d =	$\mathcal{E}[\![$ expr $]\!]$ ρ d $\mathcal{R}[\![$ body $]\!]$ (<id,d>:ρ) (d+1)	
(2)	$\mathcal{R}[\![$ return expr; $]\!]$ ρ d =	$\mathcal{E}[\![$ expr $]\!]$ ρ d $\mathcal{E}[\![$ cont $]\!]$ ρ (d+1) **squeeze** 2 d; **return**;	(stack return address)
(3)	$\mathcal{R}[\![$ if (expr) {body$_t$} else {body$_e$} $]\!]$ ρ d =	$\mathcal{E}[\![$ expr $]\!]$ ρ d **bfalse** lbl; $\mathcal{R}[\![$ body$_t$ $]\!]$ ρ d **label** lbl; $\mathcal{R}[\![$ body$_e$ $]\!]$ ρ d	(fresh label)
(4)	$\mathcal{E}[\![$ fun(expr$_1$, $\cdots$, expr$_n$) $]\!]$ ρ d =	$\mathcal{E}[\![$ expr$_n$ $]\!]$ ρ d ⋮ $\mathcal{E}[\![$ expr$_1$ $]\!]$ ρ (d+n−1) $\mathcal{E}[\![$ lbl $]\!]$ ρ (d+n); **call** fun$_{n+1}$; **label** lbl;	(stack fresh label)
(5)	$\mathcal{E}[\![$ id[n] $]\!]$ ρ d =	$\mathcal{E}[\![$ field(id,'WORDSIZE*n') $]\!]$ ρ d	
(6a)	$\mathcal{E}[\![$ id $]\!]$ [$\cdots$, <id,p>, $\cdots$] d =	**dup** (d−p);	(variable)
(6b)	$\mathcal{E}[\![$ id $]\!]$ ρ d =	**push** id;	(global name)

Figure 5: FCG's compilation rules to KOALA instructions

When calling a function the caller constructs a call-frame by evaluating the parameter expressions one by one on top of the stack and pushing a return address. Then a jump is made to the function entry point, see rule (4) in Figure 5. When the callee has computed the result, it fetches the return address from the stack and removes its call frame from the stack with the **squeeze** instruction, while leaving the result on top of the stack,

see rule (2). The presence of the return address as an extra parameter is made explicit in rule (0) where the identifier cont(inuation) is inserted in the symbol table.

According to the syntax of Functional C (Figure 2), assignments to local variables only occur at the beginning of basic blocks. This restriction guarantees that whenever an assignment is encountered, the call stack contains only parameters and variables, but no anonymous temporary expressions. Rule (1), which compiles the assignment statement, therefore simply extends the call frame by calling the $\mathcal{E}$ scheme and records the location in the association list (ρ) for compilation of the remainder. This contrasts with the common technique of allocating space for all local variables at once at the function entry. Our method of not allocating complete call frames at the function entry has several advantages:

- There is no need to initialise variables on function entry to keep the garbage collector from chasing arbitrary pointers placed on the stack sometime earlier. Omitting the initialisation might (and will!) lead to a crash of garbage collectors that move objects since it is possible to find an old pointer (to a deallocated object) on the stack that now points in the middle of a new object.
- The size of the stack frames is usually smaller because variables are allocated on demand. If a function calls another function then only variables that have already been assigned are saved on the stack and no space is wasted for variables that will be assigned when control returns. Furthermore, if the then and else branch of an if-statement use a different number of variables then the actual number of variables is saved in each branch when calling a function instead of the maximum number.
- It is easier to optimise a stack without "holes" as will become clear in Section 5.

Rule (3) in the compilation schemes handles the if-then-else control flow construct of Functional C; note that both branches use the $\mathcal{R}$eturn scheme to compute the function result, hence no additional trailing code is necessary. The array subscript is syntactic sugar for the *field* primitive, which loads a word from the heap at the location specified by the base and offset arguments, rule (5). Finally, rule (6) handles the evaluation of identifiers. It distinguishes between two types: variables, which are to be found on the call stack, and global names that refer to (constant) functions.

The $\mathcal{K}$, $\mathcal{R}$, and $\mathcal{E}$ compilation schemes in Figure 5 generate a subset of the KOALA instruction set: only the pure stack instructions (i.e. the ones that do not have register operands) are being used. This is a consequence of using Functional C, which contains no built-in operators, but calls primitive functions instead. These primitives are directly coded in KOALA and exercise the remainder of the KOALA instructions (e.g., alu, load, and store). The optimising compilation schemes, that will be presented in Section 5, make full use of the KOALA instruction set.

Data representation

Besides the ordinary primitives for arithmetic, logic, etc. the FAST compiler uses several primitives that operate on the heap: *cons*, *head*, *tail*, *vap*, *reduce*, etc. Those primitives that allocate objects (vectors) in the heap must check whether the garbage collector has to be invoked. *Reduce* executes delayed computations that are stored in the heap, but first test whether its argument already refers to a data object or not. The efficiency of the garbage collector and of *reduce* depends on the exact data encoding of (pointers to) heap objects.

Our data representation is targeted towards a simple two-space copying garbage collection algorithm [3]. The collector moves all live data to the empty semi-space by scanning the call stack for root pointers into the heap. To facilitate the scanning operation, all values (words) on the stack, except return addresses, are tagged so that the collector can distinguish between heap pointers and basic data like integers. For the same reason, all heap objects are encoded as a sequence of tagged words. Since objects are copied, closures and other variable length objects include their size in the header field.

The data representation scheme in Figure 6 is designed for a 32-bit word machine, which allows us to encode tag information in the two least significant pointer bits for "free". These two bits are insufficient to encode all different data types, so both constructor and vector-apply nodes start with a header field that provides additional information. Note that both the list constructor and function application node do not contain explicit tag fields, but are just recognised by the two least significant pointer bits in their first field: the tail (!) and function pointer respectively. For the other node types (e.g., suspensions) the tag info is combined with useful attributes like node size and function arity into one header field for space efficiency. Furthermore these headers are encoded as basic values, i.e. the least significant bit is set to 1, so that the garbage collector will automatically skip them when scanning moved nodes for pointers into live data.

With this data representation scheme, the *reduce* primitive only has to check the two least significant bits of its argument to determine whether it is in head normal form or not. In the latter case, inspection of the header field shows whether the argument refers to a suspension that can be invoked immediately, or to a function

Pointer	Type	Description
$xxxx1$	Basic	Basic data types like integers and characters are encoded in the pointer itself; $xxxx$ represents the 31-bit value.
$xxx10$	Cons	$xxx00$ points to a data constructor, which is a vector of consecutive fields. Additional type information is encoded in the least significant bits of the first field: header type $yyy0$ list $yyy1$ curried function (i.e a function and some of its arguments)
$xxx00$	VAP	$xxx00$ points to a vector apply node; the first field is used to distinguish between two types: header type $yyy0$ function application $yyy1$ suspension (i.e a function and all its arguments)

Figure 6: Data representation

application chain that has to be unwound first. *Reduce* uses a third VAP type, namely the indirection node; to preserve the sharing of lazy computations, *reduce* must update a VAP node with the result. Since the tag is encoded in the pointer to the VAP node, *reduce* can not change the type to, for example, integer, but overwrites the header field with the identity function and stores the result in the argument field. By recognising indirection nodes, *reduce* can simply fetch the result instead of making a function call.

In comparison with data representation schemes that do not use pointer bits to encode tags, but always include a tag in an object that may or may not be reduced, our method has two advantages: First, the *reduce* function saves a memory reference since it does not have to fetch the tag from memory to decide whether its argument has already been evaluated or not. Secondly, the nodes are encoded as space efficiently as possible, which reduces the number of garbage collections, and improves the cache locality as well. The disadvantages of our scheme are the usage of indirection nodes and the tagging/masking of basic data values. The latter can largely be avoided by using a garbage collector that has knowledge about the layout of stack frames and vector nodes.

5 Optimisations

The usage of the FCG compilation schemes of Figure 5 in combination with the KOALA-to-C translator results in poor runtime performance of the generated object code. The C compiler, which is the final stage in our compiler pipeline, does not "understand" the meaning of KOALA's stack instructions and faithfully compiles every **push** and **dup** instruction to loads and stores. The C compiler can not properly optimise basic blocks by keeping temporary stacked values in registers. To make full use of the C compiler's optimisation capabilities, we therefore present some optimisation schemes that transform the FCG's stack code into a form that is amenable to optimisations by the C compiler; of particular importance are those optimisations that replace KOALA stack instructions by register moves.

To illustrate the effects of the various optimisations, we will use the append function (Figure 3) as an example throughout the remainder of this section. A quantitative analysis of the runtime performance of a benchmark with various optimisations is provided in Section 6. The unoptimised compiler schemes of Figure 5 produce the following code for *append*:

```
fun append_3;  dup 2;                label L3;  push #4;
               push L0;                        dup 4;
               call null_2;                    push L4;
label L0;      bfalse L1;                      call field_3;
               dup 3;                label L4;  dup 5;
               push L2;                        dup 2;
               call reduce_2;                  push prel_append;
label L2;      dup 2;                          push L5;
               squeeze 2 3;                    call vap_4;
               return;               label L5;  dup 3;
label L1;      push #0;                        push L6;
               dup 3;                          call cons_3;
               push L3;              label L6;  dup 4;
               call field_3;                   squeeze 2 5;
                                               return;
```

Tail call optimisation

Both branches of the if-then-else construct in *append* end by returning the value of a function call (*reduce*, and *cons* respectively). The corresponding KOALA code evaluates the function call, reorders the stack frame, and returns the (unmodified) result. At runtime this results in *reduce*/*cons* jumping back to *append*, squeezing the call stack, and jumping back to *append*'s caller. The sequence of jump-to-caller instructions can be collapsed into one by reordering the stack frame before the (tail) call, and passing *append*'s return-address on to the primitive call. This can easily be accomplished by extending the $\mathcal{R}$ scheme to include a special case as shown in Figure 7.

(2a) $\mathcal{R}[\![\ \text{return fun}(\text{expr}_1, \cdots, \text{expr}_n);\ ]\!]\ \rho\ \text{d} \quad = \quad$

$\mathcal{E}[\![\ \text{expr}_n\ ]\!]\ \rho\ \text{d}$
$\vdots$
$\mathcal{E}[\![\ \text{expr}_1\ ]\!]\ \rho\ (\text{d+n}-1)$
$\mathcal{E}[\![\ \text{cont}\ ]\!]\ \rho\ (\text{d+n})$ (stack return address)
squeeze $(n+1)$ d;
branch fun_{n+1};

Figure 7: FCG compilation rule for tail calls, to be inserted before rule 2 in Figure 5.

The effect of rule (2a) can be seen in the following code; revision bars indicate the difference with the naive code:

```
fun append_3;  dup 2;                label L3;  push #4;
               push L0;                        dup 4;
               branch null_2;                  push L4;
label L0;      bfalse L1;                      call field_3;
               dup 3;                label L4;  dup 5;
               dup 2;                          dup 2;
               squeeze 2 3;                    push prel_append;
               branch reduce_2;                push L5;
label L1;      push #0;                        call vap_4;
               dup 3;                label L5;  dup 3;
               push L3;                        dup 5;
               call field_3;                   squeeze 3 5;
                                               branch cons_3;
```

Compile-time stack simulation

To improve the KOALA stack based code by introducing registers we will need to simulate the stack at compile time. Then it will be possible to replace matching **push** and **pop**/**fetch**/**dup** instructions by (register) **move**s inside basic blocks. Basic blocks are delimited by **fun**, **call**, **branch**, and **return** instructions.

In contrast with the tail-call optimisation, we do not enhance the basic FCG compilation schemes, but provide an $\mathcal{A}$ssembly scheme that will be used as an optimising filter on KOALA code. This approach has the advantage that it is much easier to pass the stack on to the following instruction than in the $\mathcal{E}$ scheme where

we would need an attribute grammar to do so. Besides the instruction stream and the (simulated) stack the $\mathcal{A}$ scheme in Figure 8 takes an environment argument to record the lexical scope.

$\mathcal{A}$[| **fun** $name_n$; Code |] S E	= **label** $name$; $\mathcal{A}$[| Code |] ε (S:E)
$\mathcal{A}$[| **push** id; Code |] S E	= $\mathcal{A}$[| Code |] (id:S) E
$\mathcal{A}$[| **pop** reg; Code |] (v_1:S) E	= **move** v_1 reg; $\mathcal{A}$[| Code |] S E
$\mathcal{A}$[| **pop** reg; Code |] ε E	= **pop** reg; $\mathcal{A}$[| Code |] ε E
$\mathcal{A}$[| **dup** n; Code |] (v_1:$\cdots$:v_n:S) E	= **move** v_n reg_{new}; $\mathcal{A}$[| Code |] (reg_{new}:v_1:$\cdots$:v_n:S) E
$\mathcal{A}$[| **dup** n; Code |] S E	= **fetch** (n−#S) reg_{new}; $\mathcal{A}$[| Code |] (reg_{new}:S) E
$\mathcal{A}$[| **squeeze** n m; Code |] (v_1:$\cdots$:v_{n+m}:S) E	= $\mathcal{A}$[| Code |] (v_1:$\cdots$:v_n:S) E
$\mathcal{A}$[| **squeeze** n m; Code |] (v_1:$\cdots$:v_n:S) E	= **squeeze** 0 (m−#S); $\mathcal{A}$[| Code |] (v_1:$\cdots$:v_n:ε) E
$\mathcal{A}$[| **squeeze** n m; Code |] S E	= **squeeze** (n−#S) m; $\mathcal{A}$[| Code |] S E
$\mathcal{A}$[| **bfalse** *lbl*; Code |] (v_1:S) E	= **jfalse** v_1 *lbl*; $\mathcal{A}$[| Code |] S (S:E)
$\mathcal{A}$[| **bfalse** *lbl*; Code |] ε E	= **bfalse** *lbl*; $\mathcal{A}$[| Code |] ε (ε:E)
$\mathcal{A}$[| **branch** fun_n; Code |] (v_1:$\cdots$:v_s:ε) (S′:E)	= **push** v_s; $\cdots$; **push** v_1; **branch** fun; $\mathcal{A}$[| Code |] S′ E
$\mathcal{A}$[| **call** fun_n; Code |] (v_1:$\cdots$:v_s:ε) E	= **push** v_s; $\cdots$; **push** v_1; **branch** fun; $\mathcal{A}$[| Code |] ε E
$\mathcal{A}$[| **return**; Code |] (v_1:v_2:ε) (S′:E)	= **push** v_2; **branch** v_1; $\mathcal{A}$[| Code |] S′ E
$\mathcal{A}$[| **return**; Code |] (v_1:ε) (S′:E)	= **branch** v_1; $\mathcal{A}$[| Code |] S′ E
$\mathcal{A}$[| Instr; Code |] S E	= Instr; $\mathcal{A}$[| Code |] S E
$\mathcal{A}$[| ε |] S E	= ε

Figure 8: Compile-time stack optimisation

When translating the **dup** instruction the $\mathcal{A}$ scheme first checks whether the referenced stack item is present in the simulated stack or not. In the latter case a **fetch** instruction is issued to load the value from the physical stack into a fresh (virtual) KOALA register (reg_{new}). Otherwise the value is copied into a fresh register to avoid aliasing problems (see next section). In general the $\mathcal{A}$ scheme contains multiple rules for one KOALA instruction depending on whether the instructions arguments are present in the simulated stack or not.

An important invariant of the calling sequence in the $\mathcal{A}$ scheme is that parameters are passed on the physical KOALA stack. Therefore the $\mathcal{A}$ scheme flushes the simulated stack to the KOALA stack with a sequence of **push** instructions when calling a function (see the rules for **branch** and **call**). The same holds for **return**ing a result.

The Environment argument is used to handle the if-then-else construct of Functional C. Since the syntax of Functional C guarantees that each conditional branch terminates with a return statement, the lexical scope structure is a simple tree. The FCG compilation schemes traverse this tree in a fixed order (i.e. then before else), hence, we can record the lexical scopes with a stack. When the $\mathcal{A}$ scheme enters the then branch, it stacks the current (simulated) stack for the else branch; the beginning of the then branch is marked by the **bfalse** instruction. When the $\mathcal{A}$ scheme enters the else branch it pops the saved stack from the environment argument; the end of the then part is marked by either a **return** or **branch** instruction.

The $\mathcal{A}$ scheme uses the $\mathcal{K}$ scheme from Figure 5 (augmented with Figure 7) as follows: $\mathcal{A}$ [\| $\mathcal{K}$ [\| prog \|] \|] ε ε. Compiling the *append* example results in registers being used, but the net effect is zero since the basic blocks do not contain any real work; just setting up stack frames to call functions does not benefit from register optimisations when parameters are passed on the stack. The new append code does not contain revision bars since it has changed too much:

```
label append;   fetch 2 R0;
                push R0;
                push L0;
                branch null 2;
label L0;       bfalse L1;
                fetch 3 R1;
                fetch 1 R2;
                squeeze 0 3;
                push R1;
                push R2;
                branch reduce;
label L1;       fetch 2 R3;
                push #0;
                push R3;
                push L3;
                branch field 3;
label L3;       fetch 3 R4;
                push #4;
                push R4;
                push L4;
                branch field 3;
label L4;       fetch 5 R5;
                fetch 1 R6;
                push R5;
                push R6;
                push prel_append;
                push L5;
                branch vap;
label L5;       fetch 3 R7;
                fetch 4 R8;
                squeeze 1 5;
                push R7;
                push R8;
                branch cons;
```

Inlining of primitive functions

A first solution to make the simulated stack of the $\mathcal{A}$ scheme more effective, is to enlarge the basic blocks by inlining some of the primitive functions. Many of these primitives map to a single **alu** instruction, if the operands are available in registers. The following example shows the typical coding style of such primitives for the null and field primitives:

fun null_2;	**fun** field_3;
pop reg_{ret};	**pop** reg_{ret};
pop reg_{list};	**pop** reg_{base};
alu eq reg_{list} NIL reg_{test};	**pop** reg_{off};
push reg_{test};	**alu** add reg_{base} reg_{off} reg_{addr};
branch reg_{ret};	**load** reg_{addr} reg_{val};
	push reg_{val};
	branch reg_{ret};

Figure 9: *Null* and *field* primitives in KOALA

Inlining the primitive code in the KOALA instruction stream directly does not work for two reasons. Firstly, registers used in the primitives have to be renamed to avoid name clashes (α conversion). Secondly, the $\mathcal{A}$ scheme interprets the primitive's trailing **branch** instruction as a basic block marker, and flushes the simulated stack to memory, which reduces the benefits of the compile-time stack simulation. Therefore the additional inlining rules in Figure 10 use the α-converted-body() function that renames registers and strips the **fun** pseudo, the first **pop** and the last **branch** instruction of the primitive code. The second rule handles the tail call of a primitive function.

$\mathcal{A}[\![$ **push** *lbl*; **call** $inline\_prim$; **label** *lbl*; Code $]\!]$ S E
$\quad = \quad \mathcal{A}[\![$ α-converted-body($inline\_prim$) Code $]\!]$ S E

$\mathcal{A}[\![$ **dup** c; **squeeze** n m; **branch** $inline\_prim$; Code $]\!]$ S E
$\quad = \quad \mathcal{A}[\![$ α-converted-body($inline\_prim$)
$\quad\quad$ **dup** $(c+1)$; **squeeze** 2 $(m+n-1)$; **return**; Code $]\!]$ S E

Figure 10: Rules for inlining primitive functions, to be added to those in Figure 8.

The *append* function greatly benefits from inlining the *null* and *field* primitives. In reality all of the simple primitives are inlined, but this is not shown here to save space.

```
label append; fetch 2 R0;                    alu add R30 R31 R32;
              move R0 R10;                   load R32 R33;
              alu eq R10 NIL R11;            fetch 3 R5;
              jfalse R11 L1;                 move R33 R6;
              fetch 3 R1;                    push R23;
              fetch 1 R2;                    push R33;
              squeeze 0 3;                   push R5;
              push R1;                       push R6;
              push R2;                       push prel_append;
              branch reduce;                 push L5;
label L1;     fetch 2 R3;                    branch vap;
              move R3 R20;           label L5; fetch 3 R7;
              move #0 R21;                   fetch 4 R8;
              alu add R20 R21 R22;           squeeze 1 5;
              load R22 R23;                  push R7;
              fetch 2 R4;                    push R8;
              move R4 R30;                   branch cons;
              move #4 R31;
```

The apparent redundant data movement between registers will be handled by the KOALA assembler (i.e. the C compiler), and is of no concern for the $\mathcal{A}$ scheme. The basic blocks can be even further enlarged by inlining user functions. In principle this could be done by the $\mathcal{A}$ scheme (two passes), but the FAST front end is already capable of inlining user functions.

Parameters passed in registers

Now that we have used the simulated $\mathcal{A}$ stack to optimise stack instructions inside basic blocks, we would like to extend the scheme to optimise parameter passing between functions as well. This is attractive since the callee can use its arguments directly from registers instead of loading them from the (physical) stack first. Such register parameters still have to be saved on the stack if the callee itself calls another function, except when it makes a tail call ($\approx$ 25% of all calls). Quite often functions terminate by replying a value directly, in which case the parameters do not have to be saved at all. In general passing parameters in registers extends the basic blocks across function calls until the first sub function call and thereby provides more opportunity to optimise stack instructions.

The calling sequence will be changed as follows: parameters are passed in "global" registers, while the caller will save its internal state (arguments + locals) on the stack. When the caller resumes execution it will not restore the internal state in registers immediately, but rather fetch values from the stack on demand. This lazy scheme is advantageous after function calls if not all of the internal state is used. A function result is also passed in a global register instead of on the stack.

$\mathcal{A}[\![$ **fun** $name_n$; Code $]\!]$ S E $=$ **label** $name$; $\mathcal{A}[\![$ Code $]\!]$ (param$_1$:$\cdots$:param$_n$:ε) (S:E)

$\mathcal{A}[\![$ **branch** fun_n; Code $]\!]$ (v_1:$\cdots$:v_n:ε) (S$'$:E) $=$ **move** v_1 param$_1$; $\cdots$; **move** v_n param$_n$;
branch fun;
$\mathcal{A}[\![$ Code $]\!]$ S$'$ E

$\mathcal{A}[\![$ **branch** fun_n; Code $]\!]$ (v_1:$\cdots$:v_s:ε) (S$'$:E) $=$ **move** v_1 param$_1$; $\cdots$; **move** v_s param$_s$;
pop param$_{s+1}$; $\cdots$; **pop** param$_n$;
branch fun;
$\mathcal{A}[\![$ Code $]\!]$ S$'$ E

$\mathcal{A}[\![$ **call** fun_n ; Code $]\!]$ (v_1:$\cdots$:v_n:S) E $=$ **push** $v_{n+\#\mathrm{S}}$; $\cdots$; **push** v_{n+1};
move v_1 param$_1$; $\cdots$; **move** v_n param$_n$;
branch fun;
$\mathcal{A}[\![$ Code $]\!]$ ($reply$:ε) E

$\mathcal{A}[\![$ **call** fun_n ; Code $]\!]$ (v_1:$\cdots$:v_s:ε) E $=$ **move** v_1 param$_1$; $\cdots$; **move** v_s param$_s$;
pop param$_{s+1}$; $\cdots$; **pop** param$_n$;
branch fun;
$\mathcal{A}[\![$ Code $]\!]$ ($reply$:ε) E

$\mathcal{A}[\![$ **return**; Code $]\!]$ (v_1:v_2:ε) (S$'$:E) $=$ **move** v_2 $reply$; **branch** v_1; $\mathcal{A}[\![$ Code $]\!]$ S$'$ E

$\mathcal{A}[\![$ **return**; Code $]\!]$ (v_1:ε) (S$'$:E) $=$ **pop** $reply$; **branch** v_1; $\mathcal{A}[\![$ Code $]\!]$ S$'$ E

Figure 11: Calling sequence with parameter registers, replaces corresponding rules in figure 8

Figure 11 implements the new calling sequence and replaces the **fun**, **branch**, **call**, and **return** rules in the previous $\mathcal{A}$ schemes. On function entry the simulated stack is no longer empty, but is loaded with the global parameters. On function exit, the result is moved to the *reply* register and control is passed back to the caller. A tail call (i.e. a **branch** instruction) is translated to a sequence of instructions that moves the call parameters (from the simulated stack) into the global $param_i$ registers; if not all parameters reside on the simulated stack then the remainder has to be fetched from the physical stack with **pop** instructions. Making a function call is slightly more complicated than the tail call case: if all parameters reside on the simulated stack then the additional stacked values (i.e. locals) have to be saved on the physical stack before transferring the parameters to their global registers, else the lacking parameters have to be fetched from the physical stack as with the tail call. The code after the call proceeds with a simulated stack that contains just the result value. If a reference is then made to the saved state, the $\mathcal{A}$ scheme will automatically fetch it from the physical stack. Now the code for *append* looks much better:

```
label append; move param2 R0;               move param3 R5;
              move R0 R10;                  move R33 R6;
              alu eq R10 NIL R11;           push param3;
              jfalse R11 L1;                push param2;
              move param3 R1;               push param1;
              move param1 R2;               push R23;
              move R2 param1;               push R33;
              move R1 param2;               move L5 param1;
              branch reduce;                move prel_append param2;
label L1;     move param2 R3;               move R6 param3;
              move R3 R20;                  move R5 param4;
              move #0 R21;                  branch vap;
              alu add R20 R21 R22; label L5; fetch 2 R7;
              load R22 R23;                 fetch 3 R8;
              move param2 R4;               squeeze 0 5;
              move R4 R30;                  move R8 param1;
              move #4 R31;                  move R7 param2;
              alu add R30 R31 R32;          move reply param3;
              load R32 R33;                 branch cons;
```

Life-time analysis

The above calling sequence can be improved on two major points:

1. If after a function call the KOALA code makes multiple references to the same stack location, the $\mathcal{A}$ scheme will generate the same number of **fetch** instructions since the simulated stack is empty. One **fetch** instruction and some register moves provide the same functionality.
2. When calling a function, the $\mathcal{A}$ scheme blindly saves all local state on the physical stack, but often some stack locations will not be referenced in the remainder of the code. See for example the previous *append* code where after the *vap* call only two of the five saved values are being used.

Redundant loads from the stack can be avoided by adding another argument to the $\mathcal{A}$ scheme that records the status of the physical stack: if an item is fetched from the stack then its register is remembered.
The avoidance of saving dead variables is more difficult. Fortunately, the FAST front end has the ability to output pseudo function calls for a reference counting garbage collector, where they are used to increment or decrement the reference count of objects. The $\mathcal{A}$ scheme can take advantage of the increment/decrement pseudo functions by maintaining a life count with each item on the stack. When calling a function, only those stack items with a positive count have to be saved on the physical stack. A slight complication with this scheme is that the one-to-one correspondence between FCG's simulated stack locations and the actual physical location can no longer be maintained since we must not create holes in the stack, but in return we can reuse the stack locations of variables that were saved before and have died since the last function call.

We have constructed a new $\mathcal{O}$ptimal scheme that incorporates both improvements mentioned above. Since this $\mathcal{O}$ scheme is a straightforward extension of the previous $\mathcal{A}$ scheme, we have not provided a listing.

6 Performance

To assess the runtime performance effects of the optimisations described in the previous section, we have run a set of benchmark programs several times on a SUN 4/690. The benchmark is written in *intermediate* and

not only consists of well known toy applications like nfib and queens, but also includes 5 larger functional programs (up to 700 lines):

fft 2048 Fast Fourier transform on a vector of 2048 points, arrays are represented as lists [7].

wave 5 A mathematical model of the tides in a rectangular area of the North Sea. It consists of a sequence of 5 iterations that update a state matrix. Matrices are represented as lists of lists [14].

sched A program to find the optimal schedule of a set of tasks on a number of processors. Implemented as a parallel tree search algorithm [14]

comp_lab An image processing program that labels all four connected pixels into objects [13].

15-puzzle A branch and bound program to solve the 15-puzzle. The iterative deepening search strategy is used [4].

The execution times (user + system time) are presented in Table 1. The column marked naive contains the results for code that was produced by FCG with the straightforward $\mathcal{K}$, $\mathcal{R}$, and $\mathcal{E}$ schemes from Figure 5. The following columns list the results for adding the optimisations of the previous section one by one to the naive version: *t*ail call optimisation, *s*tack simulation, *i*nlining of primitives, *p*arameter registers, and *l*ife-time analysis. For example, the *+i* column presents the results for FCG with the basic schemes, the additional rule for tail calls (Figure 7), the stack simulation of the $\mathcal{A}$ scheme (Figure 8), and the rule to inline primitives (Figure 10).

	naive	*+t*	*+s*	*+i*	*+p*	*+l*
nfib 30	15.1	15.0	10.2	3.0	2.1	2.0
coins 279	6.7	6.7	4.7	1.4	1.1	1.1
queens 10	30.1	29.4	21.5	7.6	6.2	5.9
sieve 1000	11.3	10.9	8.1	4.6	4.1	4.1
quicksort 1000	1.5	1.5	1.1	0.5	0.5	0.4
hamming 8000	1.1	0.7	0.5	0.4	0.3	0.3
fft 2048	7.8	7.5	5.9	4.7	4.3	4.2
wave 5	1.2	1.2	0.8	0.6	0.4	0.4
15-puzzle	10.4	9.2	7.1	4.9	4.2	4.1
sched	21.6	21.2	15.0	9.4	8.7	7.9
comp_lab	4.3	4.1	3.0	1.5	1.3	1.2

Table 1: Execution time [sec] of benchmark programs under various compiler optimisations

As can be seen from the results, the optimisations improve the performance of the compiled code. The largest difference is reached for the nfib program: the optimal (*+l*) version runs 7.5 times as fast as the naive version. The fast Fourier transform shows the smallest improvement under the various optimisations: only a factor 1.9. As is apparent from the results, the inlining of the primitive functions is of vital importance for generating quality code.

It is impossible to assess the effects of individual optimisations by comparing two columns in Table 1 since the different optimisations influence each other's effects. For example, the benefits of the tail call optimisation seem minimal (usually less than 10%), but it is an important optimisation in combination with the other optimisations; the surprising performance gain of the basic $\mathcal{A}$ scheme before primitives have been inlined to enlarge the basic block sizes, see column *+s*, is facilitated by the tail call optimisation. When constructing the call frame of a tail call to another function, the parameter values are no longer naively duplicated and squeezed on the stack as with plain function calls, but they are loaded into registers and then stored directly at their final stack locations instead.

To judge the absolute performance of the code generated by FCG, we have made a comparison with two other systems: the LML compiler and the naive code generator for the FAST compiler [6]. To compare FCG with the LML_0.99 compiler from Chalmers University [2], we have translated most of the benchmark programs to LML. It is not difficult to faithfully translate *intermediate* into LML since *intermediate* is essentially a subset of LML. Table 2 lists the execution times of the benchmark programs for the three compilers.

The benchmark results show that FCG always generates better code than FAST, except for the nfib program. This is a rather surprising result since FCG generates a lot of extra code to manipulate tag bits that are present in each data value to support garbage collection. Apparently this overhead is of no great importance. The bad performance of FCG on the function call intensive nfib program is mainly caused by the C compiler that does not efficiently support FCG's calling sequence, see section 3; the execution time can be improved to 1.4 sec by sacrificing portability and generating assembly directly.

	FCG	FAST	LML
nfib 30	2.0	1.6	8.2
coins 279	1.1	4.6	3.5
queens 10	5.9	15.0	14.9
sieve 1000	4.1	18.1	5.5
quicksort 1000	0.4	1.8	1.3
hamming 8000	0.3	0.9	0.5
fft 2048	4.2	11.7	
wave 5	0.4	0.6	1.5
15-puzzle	4.1	14.9	
sched	7.9	25.0	15.7
comp_lab	1.2	4.3	

Table 2: Execution time [sec] of benchmark programs under various implementations

In contrast to FCG and FAST, the LML compiler generates native sparc assembly code. The execution times in Table 2 show that the code generated by FCG performs better than the LML counterpart. The LML compiler does not include such an interprocedural strictness analyser as the FAST front end, which explains why the FCG version of nfib performs 4 times better than its LML counterpart, and the inherently lazy sieve program only shows a 1.3 factor difference. The performance difference is also caused by the large difference in heap usage of the programs compiled by the two compilers; preliminary measurements have shown that the wave program compiled by FCG uses a factor 4,000 less bytes of heap space than the LML compiled version.

7 Conclusions

We have built an efficient, portable, functional language compiler that supports moving garbage collectors with minimal effort. This has been accomplished by reusing large parts of existing compiler technology: the FAST front end for making lazy evaluation explicit, and the C compiler for generating optimised code.

The FCG code generator consists of two passes that are described with simple transformation schemes. The first basic scheme describes a recursive descent parser that generates code for a pure stack machine, which is optimised by the second scheme that makes a linear scan over the code to combine matching stack instructions into register based equivalents. Both schemes can be combined into one attribute grammar, but that would make the optimisation scheme far more difficult to understand since the inherently sequential state information flow has to be propagated indirectly through the parse tree.

The performance results of a benchmark of functional programs, show that the optimisations have a large effect; the difference between naive code and the optimised version ranges between a factor 1.9 and 7.5. The comparison between the FAST compiler, which does not perform garbage collection, and FCG, which includes a copying collector, shows that FCG outperforms FAST on most benchmark programs. The benchmark results of the LML compiler, which generates native assembly code, show that it can not match the performance of FCG's code.

Acknowledgements

We thank Marcel Beemster, Rutger Hofman and Henk Muller for their comments on a draft version of the paper. The anonymous referees provided useful suggestions for improving the paper. The FAST compiler represents joint work with Hugh Glaser and John Wild, which is supported by the Science and Engineering Research Council, UK, under grant No. GR/F 35081, FAST: Functional programming for arrays of Transputers.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachusetts, 1986.

[2] L. Augustsson and T. Johnsson. The Chalmers lazy-ML compiler. *The Computer Journal*, 32(2):127–141, Apr. 1989.

[3] C. J. Cheney. A non-recursive list compacting algorithm. *CACM*, 13(11):677–678, Nov. 1970.
[4] J. Glas. The parallelization of branch and bound algorithms in a functional programming language. Masters thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Apr. 1992.
[5] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. Technical report CSTR 91-03, Dept. of Comp. Sci, Univ. of Southampton, UK, Jan. 1991.
[6] P. H. Hartel, H. W. Glaser, and J. M. Wild. On the benefits of different analyses in the compilation of functional languages. In H. W. Glaser and P. H. Hartel, editors, *Implementation of functional languages on parallel architectures*, pages 123–145. CSTR 91-07, Dept. of Electronics and Comp. Sci, Univ. of Southampton, UK, June 1991.
[7] P. H. Hartel and W. G. Vree. Arrays in a lazy functional language – a case study: the fast Fourier transform. Technical report CS-92-02, Dept. of Comp. Sys, Univ. of Amsterdam, May 1992.
[8] T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN notices*, 19(6):58–69, June 1984.
[9] R. Loogen, H. Kuchen, K. Indermark, and W. Damm. Distributed implementation of programmed graph reduction. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Parallel languages and architectures (PARLE), LNCS 366*, pages 136–157, Eindhoven, Netherlands, June 1989. Springer Verlag.
[10] S. L. Peyton Jones. The spineless tagless G-machine: a second attempt. In H. W. Glaser and P. H. Hartel, editors, *Implementation of functional languages on parallel architectures*, pages 147–191. CSTR 91-07, Dept. of Electronics and Comp. Sci, Univ. of Southampton, UK, June 1991.
[11] W. Schulte and W. Grieskamp. Generating efficient portable code for a strict applicative language. In J. Darlington and R. Dietrich, editors, *Declarative programming*, pages 239–252, Sasbachwalden, West Germany, Nov. 1991. Springer Verlag.
[12] S. Smetsers, E. G. J. M. H. Nöcker, J. van Groningen, and M. J. Plasmeijer. Generating efficient code for lazy functional languages. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 592–617, Cambridge, Massachusetts, Sept. 1991. Springer Verlag.
[13] Q. F. Stout. Supporting divide-and-conquer algorithms for image processing. *J. Parallel and Distributed Computing*, 4(1):95–115, Feb. 1987.
[14] W. G. Vree. *Design considerations for a parallel reduction machine*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Dec. 1989.