An Optimizing Backend for the ML Kit Using a Stack of Regions

Martin Elsman Niels Hallenberg

Department of Computer Science, University of Copenhagen

July 5, 1995

Abstract

Recent research has shown that static memory management is a promising alternative to runtime garbage collection in higher order functional programming languages. Region inference determines where in a program memory can be allocated and deallocated in a stack like manner. We present an optimizing backend for the ML Kit compiler building on region inference. As intermediate representation we use nested basic blocks. This representation allows optimizing transformations such as copy propagation, dead code elimination and register allocation to be more precise than if conventional flat basic blocks were used. We have implemented a code generator for the HPPA RISC architecture and experiments show that we in some cases generate faster code than the Standard ML of New Jersey compiler. In general, experiments show that the region based ML Kit compiler, when using our optimizing backend is comparable to the Standard ML of New Jersey compiler, not only regarding memory usage but also regarding execution time.

1 Introduction

Strongly typed languages as ML offer many possibilities of program optimization. Recent techniques have made it possible to avoid garbage collection in ML like languages by use of region inference [TT94]. Region inference is a technique that by use of type information in an intermediate lambda language infers when memory can be allocated and deallocated. Experiments show that memory usage in such implementations are modest.

Different techniques have been used for implementing backends for ML like languages. Caml Light [Ler90] generates highly compact portable byte code from an untyped extended lambda language whereas Standard ML of New Jersey uses continuation passing style to generate efficient native code for a number of architectures [AM91,App92,SA94]. The semantics of Standard ML is described formally in The Definition of Standard ML [MTH90] and further commented in the Commentary on Standard ML [MT91].

There are many issues concerned with compilation of Standard ML. In this report we only address issues regarding the backend of a compiler for the Core language of Standard ML. We describe how a set of nested basic blocks is used as intermediate language and how optimization on such an intermediate representation is performed. Also, we describe how register allocation is done using a simple coloring algorithm.

The ML Kit [BRTT93] is a highly modular Standard ML compiler written in Standard ML. Birkedal and Tofte have developed a backend for the ML Kit using region inference that generates ANSI C code [Bir94]. There are some practical drawbacks however, when compiling into C code. First, the compilation time used by all available C compilers on the generated code is very long and second, some C compilers on some architectures are not capable of compiling the generated code[1]. Third, the necessary freedom in flow of control of the generated code is not easily implemented in ANSI C[2]. When generating native machine code all these problems disappear though the compiler is now confronted with a series of new tasks.

In the following two sections we describe how the new backend is connected with the ML Kit. In the succeeding three sections we describe the low level intermediate language of the compiler and how optimizations and register allocation are performed on this language. The runtime system is described in section 7 and in

[1]This is for example the case for the Gnu C compiler on the HPPA RISC architecture, when compiling large programs.

[2]The current version of the compiler for ANSI C uses a *dispatch* loop to implement the flow of control. A function then returns a pointer to a new function which can then be called by the dispatch loop.

section 8 we describe how efficient native code for the HPPA RISC architecture [Pac94,Pac91b] is generated. Finally, in section 9 we show some experimental results.

2 The ML Kit and Region Inference

The ML Kit is a very modular Standard ML compiler written in Standard ML [BRTT93]. The ML Kit includes a compiler from an abstract syntax tree (a.s.t.) of the Core language of Standard ML into a typed lambda language[3] (t.l.l.). See figure 1 for an overview of the region based ML Kit compiler. Notice that dashed boxes denote optional passes.

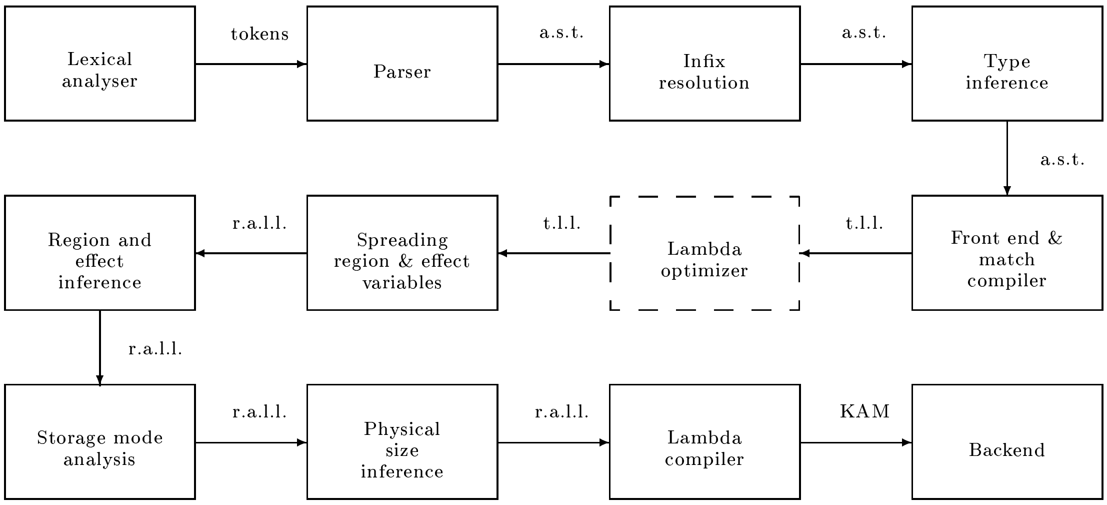


Figure 1: *Overview of the region based ML Kit compiler.*

After compilation into the typed lambda language various optimizing transformations are performed (optional). Region inference now works by first spreading fresh region and effect variables throughout the typed lambda program and then in a separate step unifying region and effect variables. A closer description of the algorithm is beyond the scope of this paper. See [TT93, section 5] for details. The storage mode analysis determines if values must be stored at top in a region or may be stored at bottom in a region (by first resetting the region) [Tof94b]. Physical size inference determines the physical sizes of regions with logical size one. Such regions are stack allocated. The backend provides an interface to the lambda compiler called the *Kit Abstract Machine* (KAM). The lambda compiler compiles the region annotated lambda language (r.a.l.l.) into sequential code including various primitives for region handling.

3 Structure of the Optimizing Backend

A backend for the ML Kit compiler must match the interface of the *Kit Abstract Machine*[4]. This abstract machine was designed to be simple and close to conventional von Neumann machines [Bir94]. It resembles the continuation machine of the Standard ML of New Jersey compiler [App92] with extra instructions for region management. The *Kit Abstract Machine* has an unlimited number of registers, hence register allocation for the target machine must be performed at a succeeding pass.

As mentioned, Birkedal and Tofte have implemented a backend for the ML Kit compiler that generates ANSI C code [Bir94]. In this implementation it is left for the C compiler to perform all optimizing transformations including register allocation. Due to the modularity and use of Standard ML Modules it is possible to construct a new backend that generates efficient machine code for a given architecture by matching the interface of the *Kit Abstract Machine.*

[3]A compiler for the entire language including Modules are under construction.
[4]More specifically, a backend for the ML Kit must match the signature *KAM_BACKEND*.

As intermediate representation in the new backend we introduce Kit backend programs (Kbp). Such *programs* are mappings from labels to what we call Kit Basic Blocks. A Kit basic block is a sequence of instructions that again may contain nested Kit basic blocks making it possible for different analyses to perform better than if only simple basic blocks were allowed. The instructions of a Kit basic block is chosen to be very close to the instructions of the *Kit Abstract Machine*. See figure 2 for an overview of the backend of the ML Kit compiler. Notice that dashed boxes denote optional passes. The Sun SPARC code generator has not been implemented.

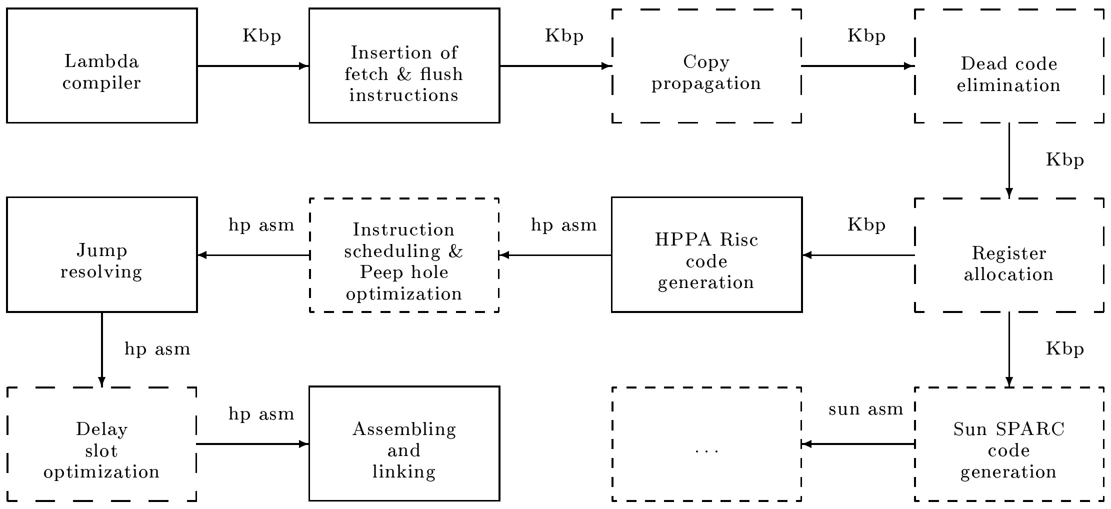


Figure 2: *Overview of the optimizing backend for the region based ML Kit compiler.*

Different optimizing transformations may be performed on Kit backend programs including register allocation. Further, different optimizing transformations depending on the target architecture may be applied after target code generation prior to the final assembly code is emitted. For the HPPA RISC architecture jump resolving and delay slot optimization is performed prior to emitting assembly code[5]. Finally, the file containing assembly code may be assembled and linked to the runtime system using conventional system commands (`as` and `ld`). Incremental compilation is not yet possible.

4 Intermediate Language

Most optimizing compilers only use one intermediate language for optimization. This is not the case for the region based ML Kit compiler we describe here. As pointed out by Appel different intermediate languages have different advantages [App92, chapter 1]. The region based ML Kit compiler perform optimizing transformations both at a lambda language level and at a Kit backend program level as described in the subsequent section. One disadvantage to this approach is that if the optimizations where performed at the same level then every optimization would be guaranteed to lead to new optimizations in the iterating process if possible.

Standard ML of New Jersey performs optimizations at a *continuation passing style* language (CPS), which is basically a restricted form of the untyped lambda language. Many optimizations are possible at this level, especially when the language is untyped. Since region inference uses type information obtained during elaboration the lambda language in the region based ML Kit compiler needs to be a typed language. This complicates the lambda optimizer somewhat, since reductions and manipulations of type expressions may be necessary when transforming a typed lambda expression. Also, not all transformations possible in an untyped lambda language are possible in a typed lambda language. The Standard ML of New Jersey compiler performs optimizations that are neither performed by the typed lambda language optimizer or the Kit backend optimizer of the region based ML Kit compiler. These optimizations include *argument flattening*, and *common subexpression elimination* [App92, chapter 6].

[5]Instruction scheduling and Peep hole optimization have not been implemented.

4.1 Kit backend programs

Kit backend programs are finite mappings from labels to what we call Kit basic blocks. A Kit basic block is a sequence of Kit backend instructions. Notice that we allow conditional instructions, containing Kit basic blocks inside a Kit basic block. This is usually not the case for basic blocks [ASU86, page 528], [Bra95].

Kit backend programs are architecture independent and serve as a low level intermediate representation of a program. Since Standard ML is a higher order language the flow of control between Kit basic blocks is implicit in the language. To make the flow of control explicit requires a global analysis as the closure analysis described in [Ses91]. Exceptions however, complicate the analyses and it is not clear how the possibility of raising exceptions may influence the analysis.

We perform various optimizing transformations on Kit backend programs and also register allocation is performed at this level. For a given architecture a few registers are reserved for the translation into the target machine language. Though optimizing transformations and register allocation are not performed across top level Kit basic blocks the analyses are allowed to extend sequences of simple instructions, allowing for better results.

4.2 Semantic objects

In this section we present semantic objects used in the presentation of the semantics of the intermediate language. Though we will not be formal in our presentation of the intermediate language (we could have implemented and presented an interpreter for the intermediate language), we will give a formal presentation of the semantic objects used in the following sections.

When A is a set then $\mathrm{Seq}(A)$ denotes the set of all sequences of elements of A and when A and B are sets then $A \xrightarrow{\text{fin}} B$ denotes the set of finite maps from A to B.

We define the following semantic objects:

$$\begin{array}{rcl}
v & \in & \mathrm{Var} \\
r & \in & \mathrm{Reg} \\
l & \in & \mathrm{Label} \\
i, \mathit{off} & \in & \mathrm{Immed} \\
\mathit{str} & \in & \mathrm{String} \\
f & \in & \mathrm{Float} \\
\mathit{is} & \in & \mathrm{InStream} \\
\mathit{os} & \in & \mathrm{OutStream} \\
k, d, r & \in & \mathrm{VarReg} = \mathrm{Var} \cup \mathrm{Reg} \\
x, y, e, s, t & \in & \mathrm{EffAddr} = \mathrm{Label} \cup \mathrm{Immed} \cup \mathrm{String} \cup \mathrm{Float} \cup \mathrm{InStream} \cup \mathrm{OutStream} \cup \mathrm{VarReg} \\
C & \in & \mathrm{KitBasicBlock} = \mathrm{Seq}(\mathrm{KitBackendInst}) \\
\mathit{KBP} & \in & \mathrm{KitBackendProg} = \mathrm{Label} \xrightarrow{\text{fin}} \mathrm{KitBasicBlock}
\end{array}$$

Instructions in the set of Kit backend instructions KitBackendInst are described in section 4.3. The set Var contains the Kit abstract machine variables, and Reg is the set of physical registers.

There is a set of registers denoted StdRegs containing registers that are considered live across Kit basic blocks. In the implementation the set includes the following registers:

exnPtr: Register holding the current exception pointer.

stdArg: Register holding standard argument.

stdArg1: Register holding pointer to vector of actual region parameters (used for region polymorphism).

stdRes: Register holding standard result.

sp: Register holding current stack pointer (points to first free word on the stack).

The registers holding the current exception pointer and the stack pointer may be used implicitly by an instruction. As an example, a push instruction manipulates the stack pointer implicitly.

The integer *representation* function $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are defined as follows:

$$\mathcal{R}\ [\![i]\!] \equiv 2 * i + 1$$

$$\mathcal{R}^{-1}\ [\![i]\!] \equiv (i - 1)/2$$

4.3 Kit backend instructions

The set of Kit backend instructions KitBackendInst may be divided into nine groups. In the following sections we descibe the semantics of each instruction in details. Notice that strings and floating point values are boxed. We represent the store as a map M from addresses to values. The store is fragmented in bytes.

Basic memory and register operations

In this group of instructions are operations for simple data flow of registers, variables and constant values.

Move(s, d): Moves s into d: $d \leftarrow_{32} s$.

Offset(s, off, d): Calculates the address of the offset off from s (off is specified in words): $d \leftarrow_{32} s + 4 * \mathit{off}$.

FetchIndexL(x, off, d): Fetches a word indexed off bytes from x: $d \leftarrow_{32} M[x + \mathit{off}]$.

StoreIndexL(s, x, off): Stores s (one word) into address indexed off bytes from x: $M[x + \mathit{off}] \leftarrow_{32} s$.

Push(s): Pushes s onto the stack: $M[sp] \leftarrow_{32} s$; $sp \leftarrow_{32} sp + 4$.

Pop(d): Pops the top word of the stack into d: $sp \leftarrow_{32} sp - 4$; $d \leftarrow_{32} M[sp]$.

AllocDouble(r): Allocates space on the stack for a floating point value.

DeallocDouble$()$: Pops a floating point value from the stack.

FetchVar(v): If v is assigned a register r then move v into r.

FlushVar(v): If v is assigned a register r then move r into v.

Having explicit instructions for allocation and deallocation of stack allocateable floating point values makes it possible to decide for each architecture if such floating point values should be allocated on a separate stack or if the machine stack should be used.

The instructions FetchVar and FlushVar are pseudo instructions used for liveness information necessary for register allocation and various optimizations.

Control flow operations

There are a very limited number of control flow operations. The target of a jump instruction may either be an immediate label or the address may be located in a register or a variable.

Jmp(t): Jumps to the target t.

Switch(x, sel, C_{def}): sel is a sequence of pairs of integer values and sequences of instructions. Control flows to the first sequence of instructions for which the corresponding integer value matches x. If no integer value matches x then control flows to the instruction sequence C_{def}.

Cond(c, s_1, s_2, C_1, C_2): If the conditional test $s_1\ c\ s_2$, where $c \in \{=, \neq, <, \leq, >, \geq\}$, is true then control flows to C_1. Otherwise, control flows to C_2.

Cond_infiniteRegion(r, C_1, C_2): If the *infinite* bit of r is set then control flows to C_1. Otherwise, control flows to C_2.

Cond_atbotRegion(r, C_1, C_2): If the *atbot* bit of r is set then control flows to C_1. Otherwise, control flows to C_2.

Notice that a switch instruction always includes a default branch. The *infinite* and *atbot* bit is explained in section 7.1.

Integer instructions

Integer instructions that may result in an unrepresentable result may raise an exception.

- $\mathsf{Muli}(x, y, d)$**:** Integer multiplication: $d \leftarrow_{32} \mathcal{R}\ [\![\mathcal{R}^{-1}\ [\![x]\!]\ * \mathcal{R}^{-1}\ [\![y]\!]\ ]\!]$.
- $\mathsf{Addi}(x, y, d)$**:** Integer addition: $d \leftarrow_{32} \mathcal{R}\ [\![\mathcal{R}^{-1}\ [\![x]\!]\ + \mathcal{R}^{-1}\ [\![y]\!]\ ]\!]$.
- $\mathsf{Subi}(x, y, d)$**:** Integer subtraction: $d \leftarrow_{32} \mathcal{R}\ [\![\mathcal{R}^{-1}\ [\![x]\!]\ - \mathcal{R}^{-1}\ [\![y]\!]\ ]\!]$.
- $\mathsf{Divi}(x, y, e, d)$**:** Integer division: $d \leftarrow_{32} \mathcal{R}\ [\![\mathcal{R}^{-1}\ [\![x]\!]\ \mathsf{div}\ \mathcal{R}^{-1}\ [\![y]\!]\ ]\!]$. Raises exception e when divisor is one or when result is not representable.
- $\mathsf{Modi}(x, y, e, d)$**:** Integer modulo: $d \leftarrow_{32} \mathcal{R}\ [\![\mathcal{R}^{-1}\ [\![x]\!]\ \mathsf{mod}\ \mathcal{R}^{-1}\ [\![y]\!]\ ]\!]$. Raises exception e when divisor is one or when result is not representable.
- $\mathsf{Noti}(x, d)$**:** Boolean complement: $d \leftarrow_{32} \mathcal{R}\ [\![\neg(\mathcal{R}^{-1}\ [\![x]\!]\ )]\!]$.
- $\mathsf{Negi}(x, e, d)$**:** Integer negation: $d \leftarrow_{32} \mathcal{R}\ [\![\sim (\mathcal{R}^{-1}\ [\![x]\!]\ )]\!]$. Raises exception e if result is not representable.
- $\mathsf{Absi}(x, e, d)$**:** Integer absolute value: $d \leftarrow_{32} \mathcal{R}\ [\![\mathsf{abs}(\mathcal{R}^{-1}\ [\![x]\!]\ )]\!]$. Raises exception e if result is not representable.
- $\mathsf{Reali}(x, d)$**:** Converts the integer $\mathcal{R}^{-1}\ [\![x]\!]$ to a boxed floating point value d.

The instructions Muli, Addi, Subi and Noti are not passed the appropriate exception value to raise when the result is not representable (on overflow). The operations div and mod are defined as in [MTH90, page 79].

Floating point instructions

Also floating point instructions may raise exceptions when the result is not representable.

- $\mathsf{Mulf}(x, y, d)$**:** Floating point multiplication: $d \leftarrow x * y$.
- $\mathsf{Addf}(x, y, d)$**:** Floating point addition: $d \leftarrow x + y$.
- $\mathsf{Subf}(x, y, d)$**:** Floating point subtraction: $d \leftarrow x - y$.
- $\mathsf{Divf}(x, y, e, d)$**:** Floating point division: $d \leftarrow x/y$. Raises exception e when divisor is zero or when result is not representable.
- $\mathsf{Negf}(x, e, d)$**:** Floating point negation: $d \leftarrow -x$. Raises exception e when result is not representable.
- $\mathsf{Absf}(x, e, d)$**:** Floating point absolute value: $d \leftarrow \mathsf{abs}\ x$. Raises exception e when result is not representable.
- $\mathsf{Floorf}(x, e, d)$**:** Floating point to integer conversion rounding towards zero. Raises exception e when result is not representable.
- $\mathsf{Sqrtf}(x, e, d)$**:** Floating point square root. Raises exception e when result is not representable.
- $\mathsf{Sinf}(x, d)$**:** Floating point sine.
- $\mathsf{Cosf}(x, d)$**:** Floating point cosine.
- $\mathsf{Arctanf}(x, d)$**:** Floating point arc. tangent.
- $\mathsf{Expf}(x, e, d)$**:** Floating point exponential. Raises exception e when result is not representable.
- $\mathsf{Lnf}(x, e, d)$**:** Floating point natural logarithm. Raises exception e when result is not representable.

The instructions Mulf, Addf and Subf are not passed the appropriate exception value to raise when the result is not representable[6]. Operations on floating point values are operations performed on boxed floating point values.

[6]According to the definition the primitive operations *sin* and *cos* must be defined for every argument. However, for very large arguments loss of precision makes *sin* and *cos* undefinable. For this reason the instructions Sinf and Cosf could be passed appropriate exceptions to raise when the result is not defined.

String instructions

The string instructions generating strings are passed the actual region variables (r) in which the result(s) are to be placed. See section 7 for an explanation of regions.

Sizes(x, d): Determines the size (number of characters) of the string pointed to by x: $d \leftarrow_{32} \mathcal{R} [\![|x|]\!]$.

Implodes(x, r, d): Concatenates all strings in the list pointed to by x. The result (pointer to string header) is put into d and the resulting string itself is put into region r.

Explodes(x, r_1, r_2, r_3, d): A string pointed to by x is exploded into a list of strings of size one and returned in d. Cons cells are put into region r_1, pairs into region r_2 and strings into region r_3.

Concats(x, y, r, d): Concatenates two strings pointed to by x and y, respectively. The result (pointer to string header) is put into d and the resulting string itself is put into region r.

Equals(x, y, d): Tests equality on the strings pointed to by x and y, respectively. The boolean result is put into d.

Ords(s, e, d): Computes the ordinal value of the first character in the string s. The result is put into d and if the string is empty the exception e is raised.

Chrs(x, e, r, d): Generates a one-character string in region r with pointer to string header returned in d and with ordinal value $\mathcal{R}^{-1} [\![x]\!]$. If $\mathcal{R}^{-1} [\![x]\!] \notin [0; 255]$ then the exception e is raised.

Input/output instructions

Some input/output instructions also raise exceptions when an error is reported from the operating system.

Open_inIO(s, e, d): Opens a file named s for reading. The instream value is put in d. Raises exception e if an error occurs.

Open_outIO(s, e, d): Opens a file named s for writing. The instream value is put in d. Raises exception e if an error occurs.

InputIO(is, x, r, d): Reads and removes the first $\mathcal{R}^{-1} [\![x]\!]$ characters from is. The resulting string is put into region r and the pointer to the string is returned in d. If is is terminated after $k < n$ characters then a string containing the k characters is returned and the characters are removed from is.

LookaheadIO(is, r, d): Reads the first character from is, puts the string in region r and the resulting string header in d.

Close_inIO(is): Closes the instream is.

End_of_streamIO(is, d): If lookahead(is) returns the empty string then the boolean value true is returned. Otherwise false is returned.

OutputIO(os, s, e): Writes the characters of the string s to the outstream os. If an error occurs the exception e is raised.

Close_outIO(os): Closes the outstream os.

Flush_outIO(os): Flushes the outstream os.

Notice, that the flush instruction is not a Standard ML operation [MTH90, page 80]. The instruction Open_outIO may raise an exception which is also an extension to the definition.

Comparison instructions

We allow special (efficient) comparison operations for integers and floating point values (and for string values, see above).

Cmpi(c, s_1, s_2, d): Returns in d the conditional result of $s_1 \ c \ s_2$, where $c \in \{=, \neq, <, \leq, >, \geq\}$.

Cmpf(c, s_1, s_2, d): Returns in d the conditional result of $s_1 \ c \ s_2$, where $c \in \{=_f, \neq_f, <_f, \leq_f, >_f, \geq_f\}$.

EqualPoly(x, y, d): Performs structural equality test of the values x and y. The boolean result is returned in d.

The polymorphic equality function uses tag information when traversing two equality values (of the same type) for equality.

Region and region variable instructions

These operations are the most important operations in our region based backend. Especially, it is important that allocation of memory in a region is implemented efficiently.

AllocRegion(r): Allocates an empty region with desciptor located at address r. Sets *infinite* bit of r.

DeallocRegion(): Pops a region descriptor from the stack and releases used memory pages.

DeallocRegionsUntil(r): Releases all memory pages in regions above r on the stack.

ResetRegion(r): Resets a region to be empty.

AllocMemL(r, i, d): Allocates i consecutive words in region r. The address of the first word is returned in d.

SetAtBotBit(r): Sets the *atbot* bit of region r.

ClearAtBotBit(r): Clears the *atbot* bit of region r.

Miscellaneous instructions

There are only three instructions in this group. The comment instructions are only used for debugging purposes.

Fresh_exname(d): Returns a fresh exception name in d.

Comment(s): Puts s as a comment in the target code.

Die(s): Prints "Uncaught exception s." and terminates the program. This instruction is used by the topmost exception handler.

The operation for returning a fresh exception name may be implemented by use of either a variable or a register as an internal counter.

4.4 Generation of a Kit backend program

In the present implementation a Kit Backend program is not generated immediately by the lambda compiler. That is, there are no fetch and flush instructions in the generated code. Two separate passes are necessary at the moment, for insertion of fetch and flush instructions. Fetch and flush instructions are necessary for optimizations and register allocation to obtain sufficient liveness information about variables.

The first pass inserts fetch instructions of the form FetchVar(v) in the Kit backend program. The traversal is a forward traversal and fetch instructions are inserted prior to the first use of a variable if the variable is not defined earlier in the Kit basic block. Also, if a variable v is defined in a branch of a conditional then a fetch instruction FetchVar(v) is inserted at the end of every other branch not defining v. Notice that unnecessary fetch instructions may be introduced during this pass since a variable defined in a branch of a conditional may be local to this branch and hence is not live at the end of the conditional. These unnecessary fetch instructions are removed by dead code elimination.

Insertion of flush instructions is done in a backward traversal of the Kit basic blocks in the backend program. During the traversal we keep track of the set of variables flushed so far. This is to ensure that a flush instruction for variable v is only introduced after the last definition of v. If control leaves the block at the end of a branch of a conditional we reset the set of variables flushed at this point during the backward traversal. Flush sets are intersected at branch points making it possible for a variable to be flushed more than once in a Kit basic block. Not all variables defined in a Kit basic block must be flushed. Only variables that are actually fetched in some block in the program need to be flushed. For this reason we keep track of the set of variables in fetch instructions in the entire Kit backend program.

In the future fetch and flush instructions could be generated by the lambda compiler making it explicit whether a variable is local to a block or shared with other blocks. This is also preferable for incremental compilation.

5 Intermediate Language Optimization

In this section we describe various optimizing transformations on Kit basic blocks, including boolean idiom simplification, copy propagation and dead code elimination. Register allocation is explained in section 6.

5.1 Boolean idiom simplification

In Standard ML boolean values *true* and *false* are constructors of a simple datatype *bool*. For simplicity reasons the lambda compiler treats boolean values as all other values. In particular, when the lambda compiler generates code for a conditional expression of two values it generates code that first compares the two values and then branch on the result of the comparison. In this way an unnecessary test is performed. We could avoid introductions of such superfluous tests by use of appropriate compilation techniques in the lambda compiler [ASU86, page 493]. Instead we follow a simpler approach similar to the boolean idiom simplification approach described in [App92, page 75]. To eliminate superfluous tests we translate every instruction sequences of the form

$$\begin{array}{ll} & \ldots \\ (1) & \mathsf{Cmpi}(c, x, y, d) \\ (2) & \mathsf{Switch}(d, [(true, C_1)], C_2) \\ & \ldots \end{array}$$

into instruction sequences of the form

$$\begin{array}{ll} & \ldots \\ (1') & \mathsf{Cmpi}(c, x, y, d) \\ (2') & \mathsf{Cond}(c, x, y, C_1, C_2) \\ & \ldots \end{array}$$

Notice that we must not remove instruction (1') since d may be live after instruction (1'). If this is not the case instruction (1') is removed by dead code elimination. However, this simple approach does not remove all superfluous tests as if appropriate compilation techniques were applied. For example, compiling conditional expressions including **andalso** and **orelse** does result in code containing superfluous tests.

5.2 Copy propagation

We present in this section a simple translation scheme for copy propagation. Copy propagation is a simple technique that translates many unnecessary move instructions into dead code that can be eliminated in a succeeding pass [Bra95, page 73], [ASU86, page 594].

Copy propagation works as a forward traversal of the instructions in a Kit basic block. In the instruction sequence

$$\begin{array}{ll} & \ldots \\ (1) & \mathsf{Addi}(x, y, z) \\ (2) & \mathsf{Move}(z, w) \\ (3) & \mathsf{Subi}(b, w, a) \\ & \ldots \end{array}$$

instruction (3) is translated into $\mathsf{Subi}(b, z, a)$ leaving instruction (2) dead if there are no definitions of z before uses of w after instruction (3).

During traversal we collect variables defined by move instructions in an environment mapping these variables to the source variables of the move instructions ($\mathsf{Move}(z, w)$ gives the environment $\{w \to z\}$). Since we do not require single static assignment form[7] we must at every program point remove mappings from the environment containing defined variables at this program point.

[7]Single static assignment form (SSA) requires that each variable in the program is only assigned once. Special Φ functions then connect variables at meeting points.

For the purpose of this translation we define the following semantic object:

$$M \quad \in \quad \text{PropEnv} = \text{VarReg} \xrightarrow{\text{fin}} \text{VarReg}$$

Also, we use k to range over VarReg. Dom M and Rng M denotes the domain and the range of the environment M. As usual, we define $M+M'$, called M *modified* by M', to be the environment with domain Dom $M \cup$Dom M' and values

$$M + M' = \text{if } k \in \text{Dom } M' \text{ then } M'(k) \text{ else } M(k)$$

Define $M \backslash\backslash K$ to be the restriction of M to the domain

$$\text{Dom } M \setminus (\{k \mid M(k) \in K\} \ \cup \ K)$$

Further, define $M \sqcap M'$ to be the environment M restricted to the domain

$$\{k \mid k \in \text{Dom } M \cap \text{Dom } M' \ \wedge \ M(k) = M'(k)\}$$

We take the liberty to consider M a substitution without explicitly extending the finite map to a total map. Notice that only if an effective address is a variable or a register it may be affected by a substitution.

The copy propagation translation scheme $\mathcal{P}$ can now be stated. An instruction sequence is translated in an environment M into a pair of an instruction sequence and an environment. The Kit basic blocks of a Kit backend program are translated in the empty environment. Variables x, y and t range over effective addresses. Variables C and c range over instruction sequences and conditions, respectively.

$$\mathcal{P} \ [\![[]]\!] \ M \ = \ ([], \ M)$$

$$\mathcal{P} \ [\![\mathsf{Jmp} \ t :: C_{dead}]\!] \ M \ = \ ([\mathsf{Jmp}(M \ \ t)], \ M)$$

Unnecessary dead code C_{dead} is removed from the instruction sequence.

```
$\mathcal{P}$ $[\![\mathsf{FetchVar}\ v :: C']\!]$ $M$ =
    let
       val $M' = M \backslash\backslash \{v\}$
       val $(C'', M'') = \mathcal{P}\ [\![C']\!]\ M'$
    in
       $(\mathsf{FetchVar}\ v :: C'', M'')$
    end
```

Notice that we do not apply a substitution to v in the scheme for FetchVar though mappings involving v is removed from the environment.

```
$\mathcal{P}$ $[\![\mathsf{FlushVar}\ v :: C']\!]$ $M$ =
    let
       val $(C'', M'') = \mathcal{P}\ [\![C']\!]\ M'$
    in
       $(\mathsf{FlushVar}\ v :: C'', M'')$
    end
```

Here we neither apply a substitution to v or remove mappings involving v from the environment.

```
$\mathcal{P}$ $[\![\mathsf{Move}(k, k') :: C']\!]$ $M$ =          $k \in$ VarReg
    let
       val $k_1 = M\ k$
       val $M' = M + \{k' \mapsto k_1\}$
       val $(C'', M'') = \mathcal{P}\ [\![C']\!]\ M'$
    in
       $(\mathsf{Move}(k_1, k') :: C'', M'')$
    end
```

```
$\mathcal{P}$ $[\![\textsf{Move}(x, k') :: C']\!]$ $M$ =        $x \notin$ VarReg
    let
        val $M' = M \setminus\setminus \{k'\}$
        val $(C'', M'') = \mathcal{P}\ [\![C']\!]\ M'$
    in
        $(\textsf{Move}(x, k') :: C'', M'')$
    end
```

There are two different schemes for Move. In the first scheme we allow k (actually k_1) to be propagated to later uses of k' whereas in the second scheme, since x is not a variable or a register no propagation is allowed for x. Notice that we use environment *modification* instead of environment (substitution) *composition* in the first scheme for Move above. Composing environments would not necessarily be a good choice since more associations could then be removed from the environment by \\.

```
$\mathcal{P}$ $[\![\textsf{Cond}(c, x, y, C_1, C_2) :: C']\!]$ $M$ =
    let
        val $(C_1', M_1) = \mathcal{P}\ [\![C_1]\!]\ M$
        val $(C_2', M_2) = \mathcal{P}\ [\![C_2]\!]\ M$
        val $M' = M_1 \sqcap M_2$
        val $(C'', M'') = \mathcal{P}\ [\![C']\!]\ M'$
    in
        $(\textsf{Cond}(c,\ M\ x,\ M\ y,\ C_1',\ C_2') :: C'', M'')$
    end
```

Here environments are intersected at code intersection points. The translation scheme for the switch construct and for other conditional constructs follow directly from the conditional construct above.

```
$\mathcal{P}$ $[\![\mathit{inst} :: C']\!]$ $M$ =
    let
        val $M' = M \setminus\setminus \mathit{defs}(\mathit{inst})$
        val $(C'', M'') = \mathcal{P}\ [\![C']\!]\ M'$
        val $\mathit{inst}' = \mathit{substUses}(M, \mathit{inst})$
    in
        $(\mathit{inst}' :: C'', M'')$
    end
```

The above compilation scheme is applied when no other scheme may be applied. The notation *defs*(*inst*) denotes the set of variables/registers defined by the instruction *inst*. The function *substUses* applies a substitution to every uses of a variable in an instruction. In the implementation copy propagation is extended to also propagate small sized integers.

5.3 Dead code elimination

The purpose of dead code elimination is to remove instructions that have no influence on the result of the computation [Bra95, page 84], [ASU86, page 595]. It is a requirement that fetch and flush instructions have been inserted in the Kit backend program prior to dead code elimination. As mentioned earlier the set StdRegs is the set of registers that are live across blocks.

Dead code elimination works as separate backward traversals of the Kit basic blocks. We collect the set of live variables/registers during the backward traversal starting with the set of registers live across blocks, StdRegs. We union live sets at branch points.

The dead code elimination translation function $\mathcal{D}$ can now be stated. An instruction sequence is translated, given a set of live variables/registers L, into a pair of an instruction sequence and a new set of live variables/registers.

$$\mathcal{D}\ [\![[]]\!]\ L = ([],\ L)$$

$\mathcal{D}$ ⟦Jmp $t :: C_{dead}$⟧ $L = ([Jmp\ t],\ L \cup$ (VarReg of $\{t\}$))

Unnecessary dead code is removed from the instruction sequence.

```
D ⟦Cond(c, x, y, C₁, C₂) :: C′⟧ L =
    let
      val (C″, L′) = D ⟦C′⟧ L
      val (C′₁, L₁) = D ⟦C₁⟧ L′
      val (C′₂, L₂) = D ⟦C₂⟧ L′
      val L″ = L₁ ∪ L₂ ∪ (VarReg of {x,y})
    in
      (Cond(c, x, y, C′₁, C′₂) :: C″, L″)
    end
```

We union live sets at branch points. As for copy propagation the translation scheme for the switch construct and for other conditional constructs follows directly from the scheme for the conditional construct above.

```
D ⟦inst :: C′⟧ L =
    let
      val (C″, L′) = D ⟦C′⟧ L
    in
      if ∀x.(x ∈ defs(inst) ⇒ x ∉ L′) andalso notDangerous(inst) then
          (C″, L′)
      else
          (inst :: C″, (L′ \ defs(inst)) ∪ uses(inst))
    end
```

Here *uses*(*inst*) are the set of variables/registers used by instruction *inst*. An instruction is said to be *dangerous* if it may cause side effects. For instance the instruction Divi(x, y, z) cannot be considered dead even if z is not live at the given program point, since the instruction may raise the exception Div. Only instructions that are not dangerous may be removed by dead code elimination. The function *notDangerous* is currently defined as follows[8]:

```
fun notDangerous inst =
   case inst
     of Move _ ⇒ true
      | Compi _ ⇒ true
      | _ ⇒ false
```

Notice that *defs*(FetchVar v) $= \{v\}$, *uses*(FlushVar v) $= \{v\}$ and *defs*(FlushVar v) $=$ *uses*(FetchVar v) $= \{\}$.

In the implementation we also remove instructions of the form

- Move(x, y) where $x = y$
- Offset($x, 0, y$) where $x = y$

5.4 Further optimization

The above optimization strategies decreases the number of instructions in the final Kit backend program drastically. By inspection of the generated code it seems that no other optimizing transformations are appropriate at the level of the Kit backend program. To implement any interprocedural optimization strategy requires the flow of control in the intermediate representation to be explicit. In general, as explained above this is not the case for Kit backend programs.

[8]Other instructions than Move and Compi could be regarded as not dangerous.

6 Register Allocation Using Graph Coloring

Register allocation is performed on the level of Kit backend programs. There are several reasons to perform register allocation at this level. First, Kit backend programs are independent of the target machine language making it possible to use the same register allocation scheme for all target machines. Second, most instructions are simple at this level compared to machine code instructions and finally it is possible to run some of the optimizing transformations after register allocation to improve the quality of the code further.

At first it seems expensive to perform register allocation at this stage since three or four registers must be reserved for target code generation. On architectures such as HPPA RISC however, registers are split up into a set of callee-saves registers and a set of caller-saves registers. Only callee-saves registers may be live across runtime system calls (procedure calls in general) and since calls to the runtime system appear fairly often register allocation only uses the set of callee-saves registers. This allows us to use all caller-saves registers as temporaries when generating target machine code as long as there are no need for temporaries to be live across calls to the runtime system.

The scheme for register allocation that we present here is close to the approach given in [Cha82] extended to work on Kit basic blocks. Callahan and Koblenz describe in [CK91] how inter procedural register allocation may be performed via hierarchical graph coloring. Unfortunately, this attempt requires precise explicit control flow information in the intermediate representation. As we have seen earlier this is not in general the case for our intermediate language.

Register allocation is done in two passes over the Kit basic blocks. The first traversal of a Kit basic block is a backward traversal in which an interference graph is calculated by use of liveness information. Given an interference graph and a set of colors (available registers on the target machine) a coloring can be found for the block. Finally, a second traversal substitutes registers for variables and transforms flush and fetch instructions into move instructions[9]. Variables that are not assigned registers are left untouched by this traversal. It is the job of the target code generator to generate code for fetching and flushing variables into temporary registers when needed.

Floating point register allocation has not been implemented at this point since floating point values are represented boxed in the implementation. Floating point register allocation naturally relies on a box/unbox analysis though a simpler approach is possible [App92, section 13.11].

6.1 Computing the interference graph

The interference graph (IG) of a Kit basic block is computed in a backward traversal by use of collected liveness information. An edge is introduced in the interference graph between node v and v' if v is live when v' is defined. We do not at the moment coalesce or combine nodes which are the source and targets of copy operations as suggested in [Cha82] (subsumption).

An interference graph is defined as a set of pairs of variables where each pair $(v, v') \in \mathrm{Var} \times \mathrm{Var}$ represents an edge between the nodes v and v'[10]. For a given node v its neighbours written $\mathrm{IG}(v)$ are given by the set $\{v' \mid (v', v) \in \mathrm{IG} \vee (v, v') \in \mathrm{IG}\}$. We define $\mathrm{Dom}(\mathrm{IG})$ to be the set $\{v \mid \exists v' : (v, v') \in \mathrm{IG} \vee (v', v) \in \mathrm{IG}\}$. When *defs* and *live* are sets of variables we define $\mathrm{IGgen}(\mathit{defs}, \mathit{live})$ to be the set $\{(v, v') \mid v \neq v' \wedge ((v \in \mathit{defs} \wedge v' \in \mathit{live}) \vee (v' \in \mathit{defs} \wedge v \in \mathit{live}))\}$.

The interference graph generation function $\mathcal{IG}$ to be applied to each Kit basic block in the Kit backend program can now be stated. The set L contains all live variables and is initially empty.

$$\mathcal{IG}\ [\![[]]\!]\ L = (L, \mathrm{IGgen}(\{\}, \{\}))$$

[9]Flush instructions are not removed from the code during this pass. Rather a move instruction is inserted prior to the flush instruction. In this way the move instruction is not removed by a succeeding dead code elimination pass.

[10]For reasons of efficiency we do not represent a graph in the implementation simply as a set of pairs.

```
$\mathcal{IG}$ $[\![\mathsf{Cond}(c,x,y,C_1,C_2) :: C']\!]$ $L$ =
    let
        val $(L', \mathrm{IG}') = \mathcal{IG}$ $[\![C']\!]$ $L$
        val $(L_1, \mathrm{IG}_1) = \mathcal{IG}$ $[\![C_1]\!]$ $L'$
        val $(L_2, \mathrm{IG}_2) = \mathcal{IG}$ $[\![C_2]\!]$ $L'$
    in
        $(\{x,y\} \cup L_1 \cup L_2,\ \mathrm{IG}' \cup \mathrm{IG}_1 \cup \mathrm{IG}_2)$
    end
```

The translation scheme for the switch construct and for other conditional constructs follow directly from the conditional construct above.

```
$\mathcal{IG}$ $[\![\mathsf{FetchVar}(v) :: C']\!]$ $L$ =
    let
        val $(L', \mathrm{IG}') = \mathcal{IG}$ $[\![C']\!]$ $L$
    in
        if $v \in L'$ then
            $(L' \setminus \{v\}, \mathrm{IG}' \cup \mathrm{IGgen}(\{v\}, L'))$
        else
            (Remove $\mathsf{FetchVar}(v)$ from the code as a side effect;
             $(L', \mathrm{IG}')$)
    end
```

If v is not live when the instruction $\mathsf{FetchVar}(v)$ is met, then it is not necessary to fetch v[11].

```
$\mathcal{IG}$ $[\![\mathit{inst}(\mathit{defs}, \mathit{uses}) :: C']\!]$ $L$ =
    let
        val $(L', \mathrm{IG}') = \mathcal{IG}$ $[\![C']\!]$ $L$
    in
        $((L' \setminus \mathit{defs}) \cup \mathit{uses}, \mathrm{IG}' \cup \mathrm{IGgen}(\mathit{defs}, L'))$
    end
```

6.2 Graph coloring

To find an N-coloring from the interference graph we follow the algorithm described in [Cha82]. As long as we can find a node with less than N edges this node is removed from the graph and pushed onto a stack of colored nodes. If at some point no nodes can be removed from the graph, we explicitly remove the node with most interferences (spilling), hoping that the rest of the graph will be colorable. When the graph is empty we generate a color function $F_{col} : \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{Reg}$ by popping nodes from the stack and putting them back into the graph giving them a color different from their neighbours. The set *colors* $\subseteq$ Reg contains the N different colors. Variables not colored during this insertion result in spill code being introduced during code generation. To generate the stack of colored nodes we use the function $\mathcal{CO}$. When A is a set, $\|A\|$ is the number of elements in A. Further, we define $\mathrm{IG}\backslash\backslash n$ to be the set $\mathrm{IG} \setminus \{(v,v') \mid \exists v'' : (v = n \wedge v' = v'') \vee (v = v'' \wedge v' = n)\}$.

```
$\mathcal{CO}(\mathrm{IG}, \mathit{colorStack}, \mathit{spillStack})$ =
    if $\mathrm{Dom}(\mathrm{IG}) = \{\}$ then
        $(\mathit{colorStack}, \mathit{spillStack})$
    else if $\exists x \in \mathrm{Dom}(\mathrm{IG}) : (\|\mathrm{IG}(x)\| < N)$ then
        $\mathcal{CO}(\mathrm{IG}\backslash\backslash x, x :: \mathit{colorStack}, \mathit{spillStack})$
    else
        let
            val $x = n : n \in \mathrm{Dom}(\mathrm{IG}) \wedge \forall m \in \mathrm{Dom}(\mathrm{IG}\backslash\backslash n) : \|\mathrm{IG}(n)\| \geq \|\mathrm{IG}(m)\|$;
        in
            $\mathcal{CO}(\mathrm{IG}\backslash\backslash x, \mathit{colorStack}, x :: \mathit{spillStack})$
        end
```

Given the set *colorStack* we generate the color function F_{col} by use of the function $\mathcal{C}$ defined as follows.

[11] The dead code elimination process could have removed instructions that used the variable v.

```
$\mathcal{C}(\mathrm{IG}, colorStack) = \mathcal{C}'(\mathrm{IG}, colorStack, \{\})$

$\mathcal{C}'(\mathrm{IG}, [], F_{col}) = F_{col}$

$\mathcal{C}'(\mathrm{IG}, v :: colorStack, F_{col}) =$
        let
            val $allowableColorSet = colors \setminus \{F_{col}(v') \mid v' \in \mathrm{IG}(v)\}$
            val $color = c : c \in allowableColorSet$
        in
          $\mathcal{C}'(\mathrm{IG}, colorStack, F_{col} + \{v \mapsto color\})$
        end
```

6.3 Substituting variables

Given the color function F_{col}, a simple forward traversal substitutes registers for variables in the Kit basic block. Variables not colored by the graph coloring algorithm are not touched by the substitution. Also, during this traversal fetch instructions are translated into move instructions of the form $\mathsf{Move}(v, r)$, where v is a variable and r a physical register in the target machine. Similarly, appearance of a flush instruction results in introduction of a move instruction of the form $\mathsf{Move}(r, v)$ prior to the flush instruction. Notice that the flush instruction is not removed from the resulting code, since this would make a succeeding dead code elimination pass remove the inserted move instruction.

7 Runtime System and Data Representation

Standard ML includes a set of predefined functions for input/output, manipulations on strings and mathematical operations. Some of these functions are very time consuming to implement in assembly language and therefore these functions are implemented in a runtime system written in C. Since C is fairly portable it is possible to reuse most parts of the runtime system in backends for other architectures.

For implementing region inference the backend must provide a set of functions to manipulate a stack of regions, where each region behaves like a heap [TT93]. An algorithm for region inference inserts operations in a program for allocation and deallocation of regions and annotates each constructed value with the region in which the value should be stored. Memory allocations in regions are not necessarily allocations in the top region, since overall memory usage does not in general follow a stack discipline. The region inference algorithm is not always able to predict the size of a region at compile time, hence we need the flexibility of being able to allocate an unknown number of words in each region at runtime. Regions which size is known at compile time are allocated on the runtime stack for reasons of efficiency.

Various functions for operations on the stack of regions are included in the runtime system, since development and modifications of C code rather than assembler code is preferable. It turns out however, that region operations are frequent and that in-lining of some of the procedures speeds up execution time for many example programs significantly[12].

The runtime system also includes functions for collecting statistics on region usage. More advanced region profiling is described in [Tof94a].

Generally, the runtime system is compiled and linked to an assembled generated target program prior to execution of the program. The target program may in this way call functions in the runtime system. One may change or extend the runtime system with other high level routines and then allow use of these in target programs without being an expert on assembly programming. In overview the runtime system includes:

- Functions for region manipulations and statistics on these. Some small operations like allocation and deallocation of regions may optionally be in-lined directly in the HPPA assembler code.
- String representation and operations on these.
- Integer and floating point operations that are not implemented directly in HPPA assembler code.

[12]In some of our tests (see section 9) profiling showed that region manipulations were performed more than seven million times. Flags are provided to enable/disable in-lining of some region operations (see section A.2).

- I/O operations.
- Exception handling for exceptions raised by primitives in the runtime system.
- A function for testing polymorphic equality.

We also discuss data representation in this section because it is closely connected to the runtime system.

7.1 A stack of regions

Regions may be determined statically to be either *finite* or *infinite*. Finite regions may be allocated on the runtime stack whereas infinite regions may not[13]. Infinite regions (from now on just regions) need special treatment, since all data in such regions may not be allocated at the same time. Also, it is not known statically how much data a region will contain at runtime. Regions are allocated and deallocated in a stack like manner and memory are allocated on top of each region, but not always in the top region. Therefore we need a two dimensional memory space in order for any region to grow arbitrarily (see figure 3).

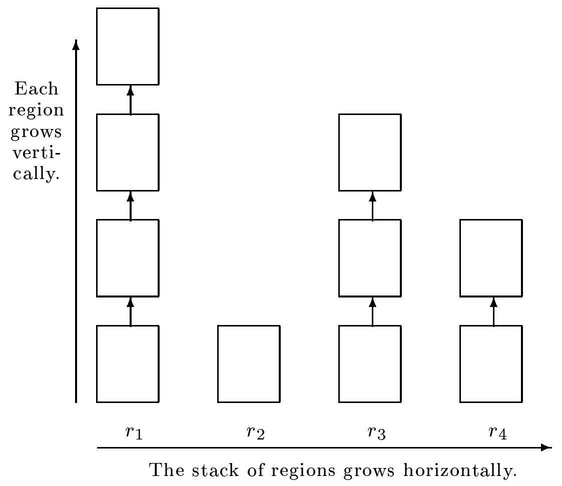

Each region grows verti- cally.
r_1
r_2
r_3
r_4
The stack of regions grows horizontally.

Figure 3: *A stack of regions. Region r_4 is the top region.*

Every region consist of one or more region pages of fixed size. These pages are connected by pointers and are taken from a pool of free pages that the runtime system maintains. If no pages are available the system call sbrk is used to get a set of new pages from the operating system. A program can at most allocate a continuous chunk of memory in a region comparable to the size of a page, hence using the right page size is important. The size of each page and number of pages allocated by sbrk are easily changed (see section A.2). When a region is deallocated the runtime system pops it from the region stack and the region pages are inserted into the pool of free pages.

A storage mode analysis [Tof94b] determines that some values may be stored at the bottom of a region when all values stored in the region already are considered dead. For region polymorphic functions storage mode information is given to the function along with actual regions at runtime. Each region has an *atbot* bit which is set when the region may be reset. This information then determines whether at some program point the actual region may be reset or not before allocation in the region. When resetting a region all region pages that are freed are inserted in the free-list.

A region has one more tag, called the *infinite* bit which is necessary to have region polymorphic functions distinguish between finite regions allocated on the runtime stack and infinite regions allocated on the region stack. An *infinite* region is identified by a pointer to a region descriptor (see next section) on the runtime stack and since addresses on the runtime stack are word aligned we may use the two least significant bits to hold the *atbot* and *infinite* bits.

Because a compiled program uses the machine stack as the program stack in co-operation with the runtime system, the runtime system does not *simulate* an ordinary stack. If at some time one is necessary, it can easily

[13]Only regions which are determined to be of logical size one (contains only one element) are considered finite. In practice regions have either logical size one or logical size infinite.

be implemented by use of a single region. Indeed, a region was used to simulate a stack in an early stage of the project.

7.1.1 Region manipulation

Each region has a *region descriptor* which contains a pointer *fp* to the linked list of pages, a pointer *a* to the next word in the region to be allocated, a pointer *b* which marks the end of the region and a pointer *p* which points to the previous region descriptor (see figure 4(c)).

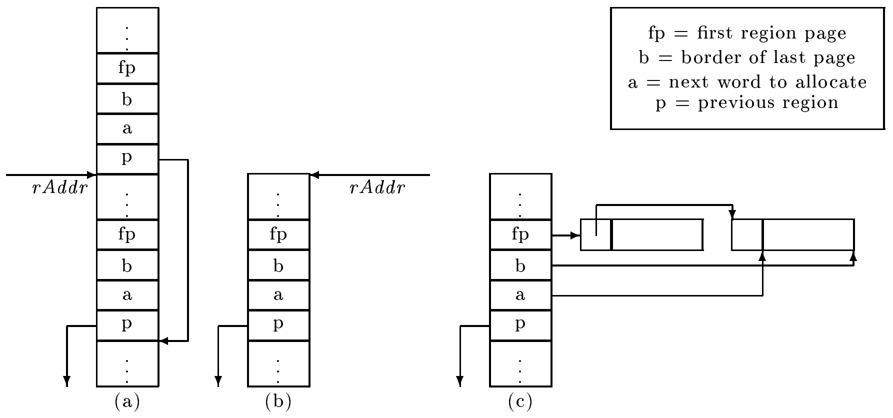


Figure 4: (a) *The stack after allocation of two regions by calls to* allocateRegion(*rAddr*). (b) *The stack when the top region has been deallocated by a call to* deallocateRegion(), *returning rAddr.* (c) *A region with two region pages where the last page is empty.*

The runtime system includes the following operations for region manipulation (see also figure 4).

alloc($rAddr, n$)**:** This function allocates n words in the region pointed at by *rAddr*. To see that all n words can be held in the current region page the expression $a+n$ must be less-than or equal to b. If this is not the case then a new region page is allocated from the free-list. If there are no free pages in the free-list a system call sbrk is executed to get a new chunk of memory, which is then fragmented into region pages and put into the free-list. The function returns a pointer to the first allocated word.

allocateRegion(*rAddr*)**:** When a region is allocated (with allocateRegion(*rAddr*)) the region descriptor is put onto the stack (pointed at by *rAddr*). A region page is taken from the free-list and the pointers in the region descriptor are set appropriately[14]. The function returns the address *rAddr* with the infinite bit set.

deallocateRegion()**:** A global variable *topRegion* always points at the descriptor for the topmost region. When a region is deallocated the freed region pages are inserted in the free-list and *topRegion* is updated to point at the previous region. The function returns the address of the deallocated region descriptor (the new stack pointer).

resetRegion(*rAddr*)**:** A region is reset by moving all region pages except one into the free-list and the allocation pointer *a* is set to point at the first word in the remaining region page. The other pointers are set as in figure 4(c) except that there is only one region page.

deallocateRegionsUntil(*stackAddr*)**:** This function is part of the exception mechanism (see section 7.4).

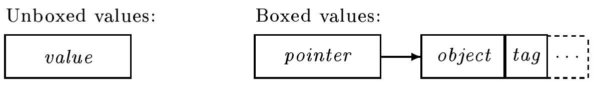
7.2 Data representation

Values are represented either boxed or unboxed at runtime. Values of type int, bool, unit, out_stream and in_stream are unboxed since these values can be held in one word. Values of other types are boxed at runtime,

[14]It is not necessary to have allocateRegion allocate the first region page, because it can be done by alloc which in all circumstances have to contain the code that fetches region pages from the free-list.

meaning that each of these values are represented as a pointer to the *real* object that may be larger than one word.

Polymorphic equality requires values of some types to be tagged to test equality of two values of (the same) arbitrary type. Among values with unboxed representation these are values of type int, bool and unit. These values should be distinguishable from boxed values for the polymorphic equality function to work.



Since pointers are always aligned on even addresses we use the lowest bit of values of type int, bool and unit as a tag denoting that the value is unboxed. For boxed values we reserve the first three bit of the object to hold a tag denoting the *kind* of value. There are six different value tags:

000 valueTagReal	**010** valueTagCon0	**100** valueTagRecord
001 valueTagString	**011** valueTagCon1	**101** valueTagRef

In the following we describe the representation of each kind of value and operations on these values.

7.2.1 Integers, booleans and units

An integer i is represented in the store as $2 * i + 1$ written $\mathcal{R}\ [\![i]\!] \equiv 2 *_c i +_c 1$, where the subscription c indicate C language operations. We will use the subscription $_{\text{ML}}$ to indicate ML operations. As we shall see, this representation makes various arithmetic operations more expensive. It is cheaper though, than to use boxed integers. The boolean value *true* is represented as the integer 3, *false* as 1 and the unit value () as 1.

The integer operations $\sim_{\text{ML}}$, $*_{\text{ML}}$, $+_{\text{ML}}$ and $-_{\text{ML}}$ are defined as follows:

$$\begin{aligned}
\mathcal{R}\ [\![\sim_{\text{ML}} i]\!] &\equiv 2 -_c \mathcal{R}\ [\![i]\!] \\
\mathcal{R}\ [\![i *_{\text{ML}} j]\!] &\equiv (\mathcal{R}\ [\![i]\!] -_c 1) *_c (\mathcal{R}\ [\![j]\!] \gg_c 1) +_c 1 \\
\mathcal{R}\ [\![i +_{\text{ML}} j]\!] &\equiv \mathcal{R}\ [\![i]\!] +_c \mathcal{R}\ [\![j]\!] -_c 1 \\
\mathcal{R}\ [\![i -_{\text{ML}} j]\!] &\equiv \mathcal{R}\ [\![i]\!] -_c \mathcal{R}\ [\![j]\!] +_c 1
\end{aligned}$$

The operation $\mathcal{R}\ [\![j]\!] \gg_c 1$ means shifting of $\mathcal{R}\ [\![j]\!]$ one bit to the right. All four operations are in-lined in the generated assembler code. The four functions $<_{\text{ML}}, >_{\text{ML}}, \leq_{\text{ML}}, \geq_{\text{ML}}$ and abs_{ML} on integers are also in-lined and are implemented as usual.

The two operations $q = i\ \mathsf{div}_{\text{ML}}\ j$ and $r = i\ \mathsf{mod}_{\text{ML}}\ j$ are defined by the formula $i = qj + r$ where $0 \leq r < j$ or $d < r \leq 0$ and d have the same sign as r [MTH90, page 79]. These functions are implemented in the runtime system by use of div and mod from the C language. $\mathcal{R}\ [\![i\ \mathsf{div}_{\text{ML}}\ j]\!]$ is defined as follows:

$$\mathcal{R}\ [\![i\ \mathsf{div}_{\text{ML}}\ j]\!] \equiv \begin{cases}
\text{raise } \mathsf{Div} & \text{if } \mathcal{R}\ [\![j]\!] = 1 \\
\mathcal{R}\ [\![0]\!] & \text{if } \mathcal{R}\ [\![i]\!] = 1 \\
2 *_c ((\mathcal{R}\ [\![i]\!] +_c 1)\ \mathsf{div}_c\ (\mathcal{R}\ [\![j]\!] -_c 1)) -_c 1 & \text{if } \mathcal{R}\ [\![i]\!] < 1 \text{ and } \mathcal{R}\ [\![j]\!] > 1 \\
2 *_c ((\mathcal{R}\ [\![i]\!] -_c 3)\ \mathsf{div}_c\ (\mathcal{R}\ [\![j]\!] -_c 1)) -_c 1 & \text{if } \mathcal{R}\ [\![i]\!] > 1 \text{ and } \mathcal{R}\ [\![j]\!] < 1 \\
2 *_c ((\mathcal{R}\ [\![i]\!] -_c 1)\ \mathsf{div}_c\ (\mathcal{R}\ [\![j]\!] -_c 1)) +_c 1 & \text{otherwise}
\end{cases}$$

The last three cases are easily verified by use of the equations:

$$\begin{aligned}
&x\ \mathsf{div}_{\text{ML}}\ y = ((x+1)\ \mathsf{div}_c\ y) - 1, && \text{for } x < 0 \text{ and } y > 0 \\
&x\ \mathsf{div}_{\text{ML}}\ y = ((x-1)\ \mathsf{div}_c\ y) - 1, && \text{for } x > 0 \text{ and } y < 0 \\
&x\ \mathsf{div}_{\text{ML}}\ y = (x\ \mathsf{div}_c y), && \text{for } (x > 0 \text{ and } y > 0) \text{ or } (x < 0 \text{ and } y < 0) \\
&x\ \mathsf{div}_c\ y = 2x\ \mathsf{div}_c\ 2y
\end{aligned}$$

For $\mathsf{mod}_{\mathrm{ML}}$ we have the equations

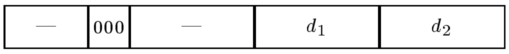
$$\begin{array}{ll} x\ \mathsf{mod}_{\mathrm{ML}}\ y = (x\ \mathsf{mod}_c\ y) + y, & \text{for } x<0 \text{ and } y>0 \text{ and } x\ \mathsf{mod}_c\ y \neq 0 \\ & \text{or } x>0 \text{ and } y<0 \text{ and } x\ \mathsf{mod}_c\ y \neq 0 \\ x\ \mathsf{mod}_{\mathrm{ML}}\ y = (x\ \mathsf{mod}_c\ y), & \text{for } (x>0 \text{ and } y>0) \text{ or } (x<0 \text{ and } y<0) \\ & \text{or } x\ \mathsf{mod}_c\ y = 0, \end{array}$$

and since $x\ \mathsf{mod}_c\ y = (2x\ \mathsf{mod}_c\ 2y)/2$ we have

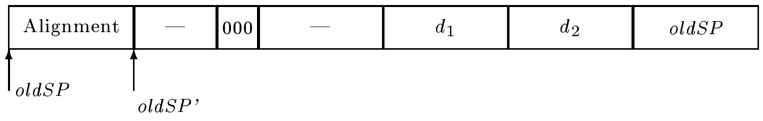
$$\mathcal{R}\ [\![i\ \mathsf{mod}_{\mathrm{ML}}\ j]\!]\ \equiv \begin{cases} \text{raise } \mathsf{Mod} & \text{if } \mathcal{R}\ [\![j]\!]\ = 1 \\ (\mathcal{R}\ [\![i]\!]\ -_c 1)\ \mathsf{mod}_c\ (\mathcal{R}\ [\![j]\!]\ -_c 1) +_c 1 & \text{if } \mathcal{R}\ [\![i]\!]\ < 1 \text{ and } \mathcal{R}\ [\![j]\!]\ < 1 \text{ or} \\ & \ \mathcal{R}\ [\![i]\!]\ > 1 \text{ and } \mathcal{R}\ [\![j]\!]\ > 1 \text{ or} \\ & \ (\mathcal{R}\ [\![i]\!]\ -_c 1)\ \mathsf{mod}_c\ (\mathcal{R}\ [\![j]\!]\ -_c 1) = 0 \\ (\mathcal{R}\ [\![i]\!]\ -_c 1)\ \mathsf{mod}_c\ (\mathcal{R}\ [\![j]\!]\ -_c 1) +_c \mathcal{R}\ [\![j]\!] & \text{otherwise} \end{cases}$$

7.2.2 Reals

On many architectures floating points (doubles) must be double-aligned. This is indeed the case for HP's PARISC. Since real values need to be tagged (polymorphic equality) we have no other choice than to represent real values as four words:



This representation requires/allows real values to be stored on double-aligned addresses. This may again cause some overhead; zero or one word for constants, one or two words when put onto the stack, and zero when put in a region. When a real value is pushed onto the stack we have:



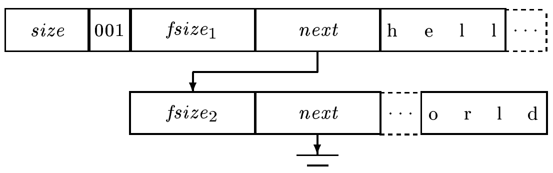
where the alignment word is only used when alignment is necessary (when *oldSP* does not point to a double-aligned address). To pop a real from the stack we pop the top word (*oldSP*) into the stack pointer.

No regions containing real values may contain values of other types. This ensures, since every region page is double-aligned that every real value located in a region is double-aligned. Thus explicit alignment of real values when put in regions is avoided.

The primitive ML functions real, /, floor, sqrt, sin, cos, arctan, exp and ln are implemented in the runtime system whereas the primitives $*$, $+$, $-$, $\sim$, $<$, $>$, $\leq$, $\geq$ and abs are in-lined.

7.2.3 Strings

A string is divided into a set of fragments. This is to allow strings to be larger than the region page size. The total size of a string is stored in the first word of the string value together with a string tag.



Constant strings are allocated statically in only one fragment. Because a single character is allocated as a string it occupies four words. The ML functions ^, size, chr, ord, explode and implode are implemented in the runtime system. There are also functions for allocating a string in a region, comparing two strings and concatenating two strings (see also [MTH90, pages 77-79]) as described below:

allocString(*rAddr*, *size*): A string of size *size* is allocated in region *rAddr*. Since *size* is in bytes and allocation in regions are in words there may be some overhead on the last word in the string.

makeChar(*rAddr*, *ch*): A string of size one containing character *ch* is allocated.

ordString(s): Returns the ordinal value of the first character of s as an integer or if the string s is empty *exnORD* is returned, with the result that exception Ord is raised (see section 7.4).

chrString(*rAddr*, *charNo*): Returns makeChar(*rAddr*, *charNo*) if *charNo* $\in [0;255]$ and otherwise *exnCHR* is returned with the result that exception Chr is raised.

sizeString(s): An integer denoting the size of the string s is returned.

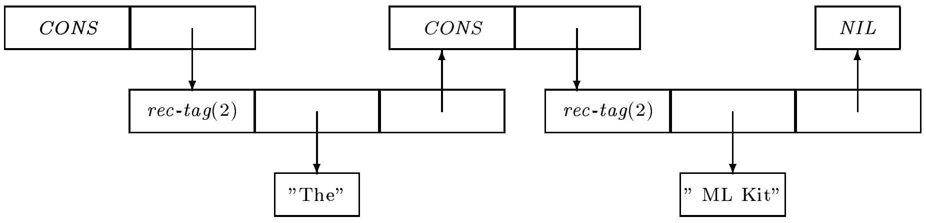
equalString(s_1, s_2): Returns *true* if the two strings are equal. Otherwise *false* is returned.

concatString(*rAddr*, s_1, s_2): String s_1 is concatenated with string s_2 and the result string is put in region *rAddr*. Since string concatenation is defined by

$$\textbf{fun } s_1 \hat{}\ s_2 = \textit{implode}((\textit{explode } s_1) \mathbin{@} (\textit{explode } s_2))$$

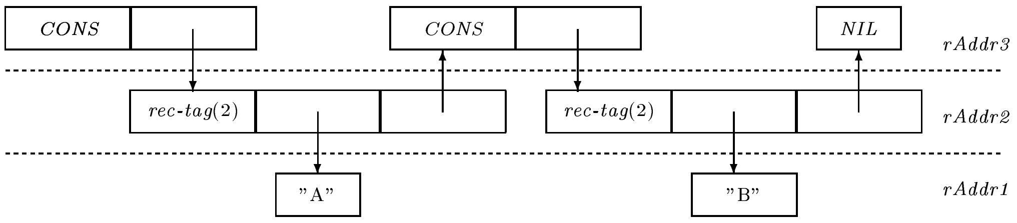
in the *prelude* for the ML Kit concatString is not used. For reasons of efficiency concatString could be used instead.

implodeString(*rAddr*, *xs*): Given a list of strings (["The", " ML Kit"]) represented as



the string "The ML Kit" allocated in region *rAddr* is returned.

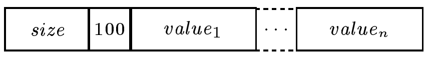
explodeString(*rAddr1*, *rAddr2*, *rAddr3*, s): Given a string s ("AB"), a list ["A", "B"] represented as



is returned. The *CONS* cells are put in region *rAddr3*, the value cells (represented as records of size two) are put in region *rAddr2* and the characters (strings) are put in region *rAddr1*.

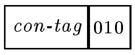
7.2.4 Records

Records are tagged. The polymorphic equality function needs (besides the tag) the size of the record to traverse all components, if necessary. The tag and the size (number of components) of the record is stored in the first word of the record.

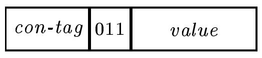


7.2.5 Constructed values

Constructed values are either nullary (carrying no argument) or unary (carrying one argument). A nullary constructed value is of size one word containing a nullary-constructor-tag (010) and a constructor tag (denoting the constructor in the datatype binding).

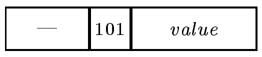


A unary constructed value is of size two words. The first word holds a unary-constructor-tag (011) and a constructor tag, and the second word holds the carried value.



7.2.6 References

References are represented as unary value constructors. A special tag is used to uniquely identify references at runtime.



7.2.7 Closures

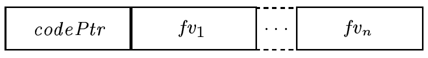
For ordinary function calls (no region polymorphism) there is a standard calling convention [Bir94, pages 14–17] using the following three registers:

stdArg: Register for containing the argument value.

stdClos: Register for holding the address of the closure for the function.

stdRes: Register to hold the result of a function call.

A closure contains the code pointer to the function and the free variables used by the function:



For region polymorphic functions [Bir94, pages 17–20] another register *stdArg1* is used pointing at a record holding the region parameters. The register *stdClos* then points at a closure containing free variables used by all the mutual recursive functions defined. No code pointers are needed for these shared closures since code pointers for the functions are known statically.

7.3 I/O operations

For implementing input/output operations we use the C language I/O facilities, such that a stream in ML [MTH90, page 80] is equivalent to a C file descriptor. The runtime system includes the following operations for input/output:

openInStream(s): Given a file name s (an ML string) the file is opened for reading and if the operation succeeds a C file descriptor is returned. Otherwise *exnIO* is returned and the exception [Io, "Cannot open s"] is raised (see section 7.4).

openOutStream(s): Given a file name s a file is created for writing. If the file exists the content is erased. If the operation succeeds a C file descriptor is returned. Otherwise *exnIO* is returned with the result that exception [Io, "Cannot open s"] is raised. This is a slight modification to the definition.

inputStream(is, n): Given a C file descriptor *is*, n characters are read from *is* and returned in an ML string. If end-of-file (EOF) is reached, the read characters up to that point are returned and the length of the returned string is calculated. We allocate the string before reading the characters. If only $k < n$ characters are read when end-of-file (EOF) is reached then unused fragments in the string are not deallocated (see section 7.2.3). To minimize the use of memory in such situations code for explicit deallocation of unused fragments could be inserted.

lookaheadStream($is, rAddr$): Given a C file descriptor *is* and a region *rAddr*, the next character to be read is returned in an ML string allocated in the region *rAddr*. If an error occurs the empty ML string is returned.

closeStream(st): Given a C file descriptor st (either an instream or an outstream) the associated file is closed.

endOfStream(is): Given a C file descriptor is, *true* is returned if end-of-file (EOF) is reached and otherwise *false* is returned. If an error occurs *false* is returned.

outputStream(os, s): The ML string s is written on the file associated to the outstream os. If any errors occur *exnIO* is returned with the result that exception [Io, "Output stream is closed"] is raised.

flushStream(st): Flushes the instream or outstream st given a C file descriptor.

7.4 Exception handling

Some runtime operations, such as the operation of finding the ordinal value of the first character in a string (ord), may raise exceptions. When this happen the current exception handler must be called with the exception value as argument. This can be done by calling the exception handler directly from the runtime system or by returning to the caller with a special value as the result denoting that an exception must be raised. It is then the responsibility of the caller to call the exception handler. For all primitives that may raise an exception the latter idea have been applied. When an exception should be raised by a primitive function in the runtime system the function returns the value zero to the caller[15]. The caller must then check if zero is returned by the primitive function and if this is the case the exception handler is called with the appropriate exception value as argument.

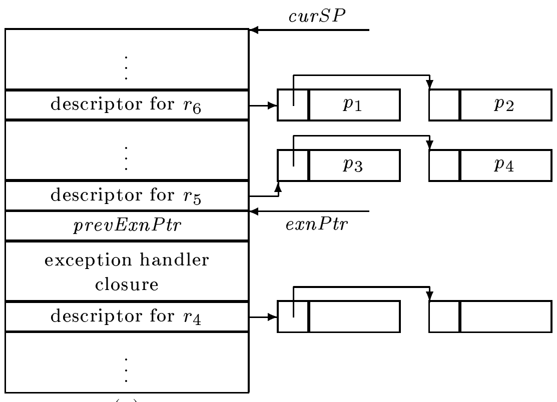


Figure 5: deallocateRegionsUntil *is called with exnPtr as argument. Starting with the top region r_6 it deallocates all regions over exnPtr (r_6 and r_5) and makes r_4 the top region. The allocated region pages ($p_1 \ldots p_4$) are inserted in the free-list. After calling* deallocateRegionsUntil *exnPtr is set to prevExnPtr and the handler closure is called with the exception value as argument.*

When the current exception handler is called (by a raise construct in the source program or a primitive operation) it resets the stack pointer from the current stack pointer (*curSP*) down to a given new address stored in a global exception pointer register (*exnPtr*). Unfortunately, adjusting the stack pointer is not sufficient. It is also necessary to deallocate regions which have been allocated in the time the stack has grown from the bottom address *exnPtr* to the current address *curSP*. For this reason region descriptors are put onto the runtime stack instead of on a separate stack. A special function deallocateRegionsUntil() is used to deallocate all regions that have their descriptors in the address space between *curSP* and *exnPtr*. The previous exception pointer and the handler closure are now located on the stack where *exnPtr* points. Before calling the handler with the exception value as argument *exnPtr* is set to be the previous exception handler. Figure 7.4 shows what happens when the exception handler is called.

[15]All runtime operations that may raise an exception will never have zero as a legal result.

7.5 Polymorphic equality

For testing polymorphic equality a function equalPoly(x, y) is implemented that traverses the two structures x and y recursively and compares each component.[16] Note that two references are considered equal if and only if the pointers are equal. As mentioned earlier it has been necessary to tag values that may be tested for equality for polymorphic equality to be implemented.

7.6 Statistics

To see how memory behaves at runtime and particularly to find space leaks in a program the runtime system allows for collection of statistics on regions. For example one can see the result of changing the region page size or the number of allocated bytes by sbrk (see section A.2).

If a program uses a lot of nested regions with no more than a few words in each, then it is probable that a small page size is better than a large page size. On the other hand, regions with many memory allocations will spend a lot of time allocating pages. In this way for example, statistics may help us choosing a suitable page size. Statistics will for a given program give the following information:

- Number of calls to deallocateRegion.
- Number of calls to alloc.
- Number of calls to resetRegion.
- Number of calls to deallocateRegionsUntil.
- Number of calls to allocateRegion.
- Number of calls to sbrk.
- Maximum number of used pages.
- Maximum number of allocated regions.
- Number of allocated words by sbrk.
- Maximum number of words used in regions.

The runtime system includes the following functions for statistics:

allocatedSpaceInRegion: Calculate the allocated and used space in a particular region.

printRegion: Print the region descriptor and linked list of regions in a particular region.

printRegionStack: Print all region descriptors in the region stack.

printStat: Print collected statistics.

To obtain correct statistical information one must use the region functions in the runtime system and not the in-lined versions of the functions (see appendix A.2).

8 Generating Code for the HPPA RISC Architecture

Because the Kit backend instruction set are close to the instruction set for a RISC architecture the compilation from a backend program to HPPA code is fairly straight forward. It is necessary however, to understand the procedure calling conventions of the HPPA RISC family of compilers to be able to interact with the runtime system. Also, it is necessary to understand many subtleties of the HPPA RISC architecture in order to generate efficient code. Among other things this involves decisions regarding register and instruction usage. Johnson and Miller describe in [JM86] an optimized compiler for the HP's PARISC architecture.

In the following we describe the parts of the HPPA RISC architecture (PARISC) [Pac94] and the standard HPPA RISC procedure calling conventions[17] [Pac91b] that are used by our backend. In particularly we make attention to:

- The registers, and how they are partitioned into caller and callee save partitions. The idea is that registers which are saved by caller, can freely be used by callee without being restored and vice versa.

[16] It cannot be implemented as a state machine because of records.

[17] In this text, a procedure call is any call to some code, that we eventually will return from.

- Result passing, and where caller can expect to find the result of a function call.
- Rules for allocating frames on the stack containing information for callee and caller to actually perform the call and return. For cache optimization there are also rules for frame alignment.
- The distinction between leaf and non leaf procedures, where leaf procedures are those that do not perform additional calls.
- Classification of procedure calls into intra-module (local) and inter-module (external) calls and an optimized type of call called *millicode* call. The last one is used to implement low-level functions like integer multiplication and division where saving can be omitted when the calls are invoked.

We also show how the different calls are coded with use of HP's assembler (as, see [Pac91a, chapter 6]).

Execution of a program generated by our backend starts in the runtime system (runtimeHPPA.c). The runtime system code calls a procedure code in the module hpcode.s, which contains the generated target code. From code we typically make a huge number of calls to the runtime system. When the procedure code has finished execution we call a procedure (terminateProg) in the runtime system, which optionally outputs some statistics, and then exits to the operating system. We use terminateProg instead of just returning to main in runtimeHPPA.c because we do not know how many global regions that are left after execution, and hence the stack pointer is not preserved.

In figure 6 we show the flow and physical layout of hpcode.s.

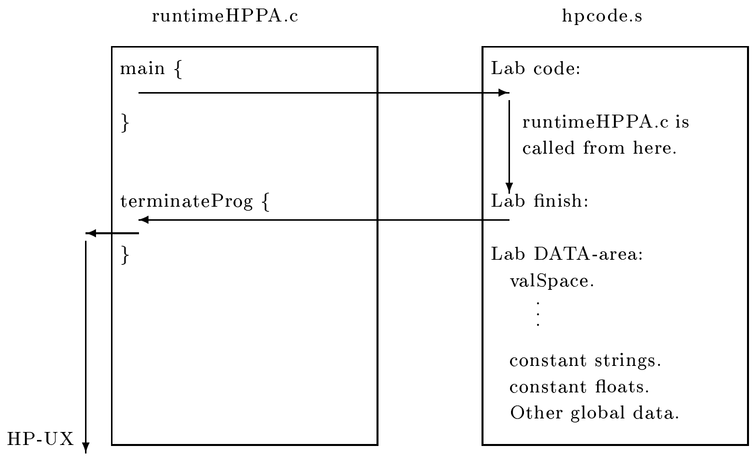


Figure 6: *Layout of* hpcode. *All global data (*valSpace *and constants) are in the* DATA*-area.*

All spilled registers are stored in the valSpace area which are statically allocated in the global data area (DATA-area). In DATA-area we also put all constants.

We only use the procedure calling conventions when branching between runtimeHPPA.c and hpcode.s. When branching inside hpcode.s we use the ML Kit calling convention. That is mainly the use of the registers *stdArg*, *stdClos*, *stdRes*, *stdArg1* and *exnPtr* (see sections 4.2, 7.2.7, 7.4 and [Bir94, pages 14–17]).

8.1 Register partitioning

The PARISC has 32 general registers and 32 floating point registers[18]. The 32 general registers are by the procedure calling conventions partitioned into 11 caller-save registers, 16 callee-save registers, and five specific registers.

In our backend caller-save registers are typical those saved in the procedure code before a call to the runtime system. The callee-save registers are typical those saved in the runtime system when a procedure is called.

Of the caller-save registers the convention dedicates four registers as argument registers and two as result registers, such that a 64 bit result can be returned. The caller also has the responsibility of storing four non

[18]We assume version 1.1 of the HPPA RISC architecture. Version 1.0 only has 16 floating point registers.

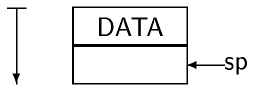
dedicated general purpose registers and a scratch[19] register Gr1. In general the dedicated registers are also used as scratch registers. Table 1 shows the register usage convention.

Registers	Synonyms	Savee	Usage convention
Gr0			Writing to this register does not effect its contents.
Gr1		Caller	Scratch register. May be changed in a call.
Gr2, Gr31	rp, mrp		Ordinary and millicode return pointer.
Gr3 - Gr18		Callee	General purpose registers.
Gr19 - Gr22	tempReg0 - tempReg3	Caller	In our backend only used as temporaries.
Gr23 - Gr26	arg3 - arg0	Caller	Argument and scratch registers.
Gr27	dp		Global data pointer. Does not hold any other value.
Gr28, Gr29	ret0, ret1	Caller	Integer return registers and scratch registers.
Gr30	sp		Stack pointer. Does not hold any other value.

Table 1: *General register usage. The synonym names* tempRegx *are only used in our backend. The other synonyms are inherited from the Procedure Calling Convention manual.*

The called procedure (callee) has the responsibility of storing 16 general purpose registers, but it is only necessary to store the registers which are actually changed by callee. This storing convention makes it possible to minimize the number of registers to store over procedure boundaries, because callee knows which are used in the called procedure. Another convention which makes caller responsible of storing all registers would not be preferable because then, caller should know which registers callee uses, if not all registers were to be saved.

The five registers Gr0, Gr2, Gr27, Gr30 and Gr31 are not in one of the two partitions. The register Gr0 has the value zero, and can not be updated to anything else. The global data pointer (Gr27) always points at the global data area with modifiable data. For compatibility with the HP-UX operating system all data must be accessed relative to Gr27 [Pac91a, page 2-7]. The stack pointer (Gr30) points at the next available address on the stack.



There are two return pointer registers (linkage registers); one for ordinary procedure calls (Gr2) and one for millicode calls (Gr31) (see section 8.2.1).

The 32 floating point registers are also partitioned into caller and callee save registers (table 2). Because all floats are boxed in our backend we only use a few floating point registers for temporaries, arguments and return values when calling floating point operations like sinus and cosinus.

Registers	Synonyms	Savee	Usage convention
Fr0 - Fr3			Status registers not used in the backend.
Fr4	retFloat	Caller	Floating point return register.
Fr5 - Fr7	argFloat0 - argFloat3	Caller	Floating point argument registers.
Fr8 - Fr11	tempFloat0 - tempFloat3	Caller	Only used as temporaries in our backend.
Fr12 - Fr21		Callee	General registers not used in our backend.
Fr22 - Fr31		Caller	General registers not used in our backend.

Table 2: *General floating point register usage. The synonym names are only used in our backend.*

The Kit backend instructions Absf, Negf, Subf, Mulf and Addf are implemented in assembler code by use of tempFloat0 - tempFloat3. The argument (argFloatx) and return (retFloat) registers are used by the Kit backend instructions Sinf, Cosf and Arctanf, because here we use the same math library functions that cc uses. The

[19]By scratch registers we mean registers that are used to hold temporary values for a very short period (few instructions).

remaining Kit backend instructions Lnf, Expf, Sqrtf and Floorf are implemented in the runtime system. Here we use the general argument (argx) and return (retx) registers. Floating point arguments to procedures in the runtime system are boxed.

8.2 Procedure calls

We translate the Kit backend program into one huge procedure called code, which is called from the runtime system. From code we have to call the runtime system, and to do this we follow the standard procedure calling conventions. Code for calls are optimized by reducing the number of registers that are stored onto the stack, and by using appropriate jump instructions corresponding to the size (length) of the jump. Insertion of the appropriate jump instructions is handled by the jump resolver (section 8.3).

For reducing the storing onto the stack we, in the next section, look at a topic called *frame allocation* (see [Pac91b]).

8.2.1 Leaf and non leaf procedures

A leaf procedure is one that does not invoke additional calls[20], and a non leaf procedure is one that does invoke additional calls. The distinction is simple and very important, because a call to a leaf procedure is much cheaper than a call to a non leaf procedure.

When a non leaf procedure is called, a *frame* is put onto the stack. A frame contain some static information, and the size is fixed (eight words) such that the called procedure can find the bottom of the frame (sp points at the first address above the frame when callee is called). If caller has to store some caller save registers, or pass some arguments in memory, they are saved before the frame is allocated. By default four words are allocated before the frame, which can be used to hold the four arguments passed in registers. In figure 7 we show the stack when callee is called. The layout is found in [Pac91b, page 2-3].

	Callers area for caller save registers.
$SP - 4(N + 9)$	Last argument in callers argument area. If more than four arguments are needed they are
$\cdots$	stored from $SP - 4(4 + 9)$ to $SP - 4(N + 9)$, where N is the last argument number.
$SP - 4(4 + 9)$	This is the first argument which is not passed in registers.
$SP - 4(3 + 9)$	The area $SP - 4(0 + 9)$ (arg0) to $SP - 4(3 + 9)$ (arg3) are always allocated and are used by
$\cdots$	callee to store the four arguments passed in registers if need be.
$SP - 4(0 + 9)$	Place to store the first argument passed in register arg0.
$SP - 32$	External Data/LT pointer (LPT).
$SP - 28$	External sr4/LT pointer (LPT).
$SP - 24$	External/stub RP.
$SP - 20$	Current rp. (set after entry).
$SP - 16$	Static link (set before call).
$SP - 12$	Clean up (set before call).
$SP - 8$	Relocation stub RP (set after call)
$SP - 4$	Previous sp (set before call).
SP	Next available address on the stack. From this address the callee save registers are
$\cdots$	saved if need be. This area are allocated by callee.

Figure 7: *Layout of stack and frame necessary for a single non leaf procedure call. The frame area are from $SP - 32$ to $SP - 4$, and the four argument area $SP - 52$ to $SP - 36$ are always allocated.*

For an explanation of the eight words $SP - 4$ to $SP - 32$ see [Pac91b, page 2-4].

If the called procedure uses the register rp (for example for another procedure call) it can be stored in $SP - 20$. A non leaf procedure will store the register rp at this place in the frame. Because a millicode call uses the register mrp as link register and not the register rp, procedures using millicode calls only are leaf procedures, hence the frame allocation is skipped [Pac91b, page 6-4]. We have not implemented any millicode procedures

[20] By calls we mean ordinary procedure calls and not millicode calls. A procedure using only millicode calls is a leaf procedure.

in our backend, though we use one which is also used by the cc compiler. It is for multiplication of two integers where we use the millicode procedure $$mull contained in the math library.[**Comment: ref appendix**].

8.2.2 Control flow of a standard procedure call

When invoking a procedure call, the procedure calling conventions specify which elements of the call are to be performed by caller and callee respectively. The flow of a standard procedure call is summarized in figure 8.

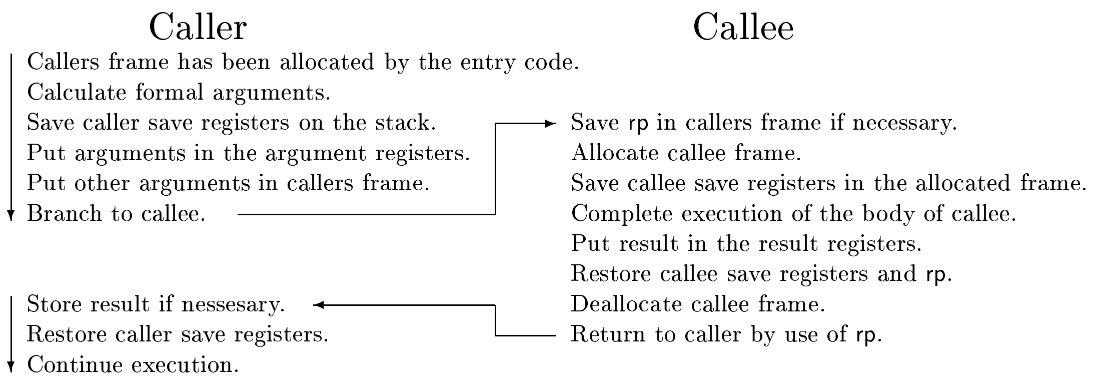

Caller
Callee
Callers frame has been allocated by the entry code.
Calculate formal arguments.
Save caller save registers on the stack.
Put arguments in the argument registers.
Put other arguments in callers frame.
Branch to callee.
Save rp in callers frame if necessary.
Allocate callee frame.
Save callee save registers in the allocated frame.
Complete execution of the body of callee.
Put result in the result registers.
Restore callee save registers and rp.
Deallocate callee frame.
Return to caller by use of rp.
Store result if nessesary.
Restore caller save registers.
Continue execution.

Figure 8: *Control flow of a standard procedure call.*

For reducing register savings over procedure calls the four caller-save registers (tempReg0 - tempReg3) are used as temporary registers locally in each block and are not used across procedure boundaries. Of the other 16 callee-save registers it is only necessary for callee to save the registers which are actually used by callee, hence these registers may safely be used in the procedure code.

The four argument registers (arg0-arg3), rp and mrp may not be mapped to variables by the register allocator, because we perform the register allocation on Kit backend instructions. We do not have enough information about when and where it is safe to use these registers. To allow register allocation to map these registers, register allocation must be performed on the HPPA machine code or another intermediate language containing enough information. Another possibility would be to use arg0-arg3, rp and mrp on variables with a life time not spanning over procedure calls.

8.2.3 Assembler code for procedure calls

We use some assembler directives to describe the calls and procedures we use, such that the assembler generates the appropriate code for the call. The most important directives are .CALL, .CALLINFO, .LEAVE and .ENTER which directs the allocation and storing of frames and registers onto the stack and how arguments are passed. The .LEAVE and .ENTER directives create the entry and exit code for a procedure corresponding to the information in the .CALLINFO directive, see [Pac91a, page 3-10].

An example of a non leaf procedure is:

```
ProcLab
  .PROC
  .CALLINFO CALLS, FRAME=0, SAVE_RP, ENTRY_GR=18.
  .ENTER
     code
  .LEAVE
  .PROCEND
```

CALLS tells the assembler that it is a non leaf procedure and a caller frame is allocated. No storing is actually performed.

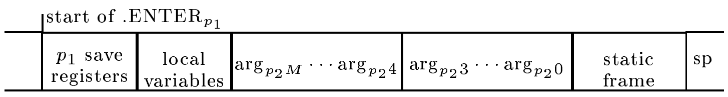
FRAME=0 specifies that there are no variable argument list or local variables before the frame, but any size (in bytes) can be allocated. The number allocated has to be a multiple of 8 bytes.

Note that the allocation of the frame and fixed argument area are controlled by `CALLS`. If we use `NO_CALLS` instead of `CALLS` no frame and fixed argument area are allocated. The area is allocated by `.ENTER` and deallocated by `.LEAVE`.

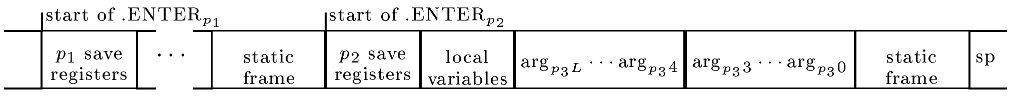
`SAVE_RP` stores the value of rp in callers frame. The storing is performed when the `.ENTER` directive is encountered and restored by `.LEAVE`.

`ENTRY_GR=`*No* specifies the high end boundary for which registers are stored by callee (Gr3...Gr*No*). In our example, `ProcLab` saves all registers (Gr3-Gr18). Code for this is also generated by `.ENTER` and `.LEAVE`.

If $\mathsf{p}_1(\mathsf{arg}_{p_1 0}, \cdots, \mathsf{arg}_{p_1 N})$ and $\mathsf{p}_2(\mathsf{arg}_{p_2 0}, \cdots, \mathsf{arg}_{p_2 M})$ are two non leaf procedures where p_1 calls p_2 and p_2 calls another procedure p_3, then the $\texttt{.ENTER}_{p_1}$ directive will generate code that allocate the following on the stack, when p_1 is called.


start of .ENTER$_{p_1}$
p_1 save registers
local variables
$\mathrm{arg}_{p_2 M} \cdots \mathrm{arg}_{p_2 4}$
$\mathrm{arg}_{p_2 3} \cdots \mathrm{arg}_{p_2 0}$
static frame
sp

When p_2 is called from p_1 the stack looks like:



where $\mathsf{arg}_{p_2 M} \cdots \mathsf{arg}_{p_2 0}$ are the formal arguments to p_2, and $\mathsf{arg}_{p_3 L} \cdots \mathsf{arg}_{p_3 0}$ are formal parameters to procedure p_3. If p_2 was a leaf procedure (`NO_CALLS` in the `.CALLINFO` directive) the allocation in $\texttt{.ENTER}_{p_2}$ was omitted.

When an external procedure[21] is called the linker needs to know how arguments and results are passed, because the procedure calling conventions allow floating point arguments and results to be passed in both general and floating point registers. When the program is linked the linker makes sure that caller and callee agree on which registers are used for the argument and result passing. If the two procedures do not agree the linker inserts a *reallocation stub* [Pac91b, chapter 3].

The `.CALL` directive, which is placed before the jump instruction, describes which registers the arguments reside in and in which return registers caller expects to find the result. With three arguments and a 32 bit result the following directive can be used:

```
.CALL ARGW0=GR, ARGW1=FR, ARGW2=FU, RTNVAL=GR
BL    procLab, rp
```

Here the first argument is passed in a general register (arg0), the second in a floating point register (argFloat1) and the third in the upper half of a floating point register (argFloat2). The result is returned in a general register (ret0).

The `.EXPORT` directive, describes the argument and result registers used by the exported procedure. An example of an export declaration in the assembler generated runtime system runtimeHPPA.s is:

```
.EXPORT    allocateRegion, ENTRY, PRIV_LEV=3, ARGW0=GR, RTNVAL=GR
```

The label `allocateRegion` is available to the linker as an entry point and it expects one argument in a general register and returns one result in a general register.

Because the runtime system only calls code in module hpcode.s once all frame allocation are performed by the runtime system except the initial frame allocation in code. When code is called we use

```
.CALLINFO CALLS, FRAME=16, SAVE_RP, SAVE_SP, ENTRY_GR=18
```

Here all callee save registers (Gr3 ... Gr18) are stored. We have `FRAME=16` such that the total allocated area is 64 bytes aligned (16 for `FRAME`, 64 for `ENTRY_GR`, 16 for static argument area and 32 for static frame).

[21] A procedure which resides in another program file (module).

8.2.4 Using the machine stack in co-operation with the runtime system

For efficient stack allocation we decided to use the machine stack in co-operation with the runtime system. When calling a function in the runtime system it seems that the operating system can put some signals onto the stack beneath the current stack pointer (when the call were invoked) and hence destroy elements on the stack[22]. Therefore, before any call to a function in the runtime system, the stack pointer is moved upwards, such that there are room for these signals. The stack pointer is restored upon return.

Offsetting sp by 1024 bytes seems to work fine but we also have to make sure that sp is 64 byte aligned (see [Pac91b, page 2-4]). Alignment has to be checked at runtime, using the method illustrated in figure 9.

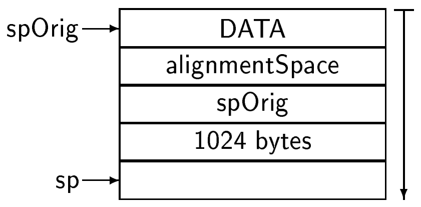


Figure 9: *Layout of the stack when a call to C is performed.*

Notice that the needed alignment space is calculated.

After the area alignmentSpace we save the original sp such that it easily can be restored. Before call we move sp 1024 bytes further up. The following code is used for alignment:

```
COPY   sp,tempReg1                  ; tempReg1 := sp
LDI    60,tempReg3                  ; tempReg3 := 60
ANDCM, =, tempReg3, sp, tempReg3    ; tempReg3 := ~sp AND tempReg3
LDO    tempReg3(sp),sp              ; sp := sp + tempReg3
STWM   tempReg1, 1028(sp)           ; [sp] := tempReg1. sp := sp + 1028
```

After the call we restore the stack with the following code:

```
LDWM   -1028(sp), sp                ; sp := [sp-1028]
```

To show that this works we have to show that the address obtained by adding spOrig to the constant alignmentSpace added to 4 is 64 byte aligned. Because spOrig is an address we have (b_0 is most significant bit):

$$\mathsf{spOrig} = 0_{31}|0_{30}|b_{29}|b_{28}|b_{27}|b_{26}|b_{25}|\dots|b_0,$$

and the result has to be:

$$\mathsf{sp} = 0_{31}|0_{30}|0_{29}|0_{28}|0_{27}|0_{26}|b'_{25}|\dots|b'_0.$$

Adding spOrig and alignmentSpace $= \sum_{i=29}^{26} \tilde{b}_i 2^{31-i}$ ($\tilde{b}$ is the complement of b) gives

$$\mathsf{sp}' = 0_{31}|0_{30}|1_{29}|1_{28}|1_{27}|1_{26}|b_{25}|\dots|b_0.$$

If we to sp′ add 4 we have a 64 byte aligned address. The constant alignmentSpace is calculated by

```
ANDCM, =, tempReg3, sp, tempReg3 ; tempReg3 := ~sp AND tempReg3,
```

because ANDCM computes the complement to sp and ands the result with 60 (= 0|0|1|1|1|1|0|...|0).

8.3 Jump resolving

The jump resolver traverses the code and determines the appropriate jumps to use (long or short). Blocks are not moved around to obtain a better result (less long jumps). There are two different target labels, *block labels* (one for each basic block) and *local block labels* (generated by conditionals and switches inside a basic block).

[22] We are not aware of the details of this flaw.

The jump resolver only has to determine the size of the jump and then insert the appropriate jump code.

For calls to labels not being block or local block labels (runtime calls) we are conservative and insert long calls.

Before describing the algorithm for the jump resolver we show the `META` jump instructions and examine the PARISC jump instructions. The five `META` jumps are as follows:

`META_IF`(*cond*, reg_1, reg_2, *targetLab*): Code that first evaluate reg_1 *cond* reg_2, and then jump to *targetLab* if the result is false is generated. Otherwise the jump instructions to *targetLab* will be skipped, and the code following the `META_IF` instruction will be executed.

`META_IF_BIT`(*reg*, *bitNo*, *targetLab*): The meta instruction is compiled into code, that first test bit number *bitNo* in *reg*, and if not set then jumps to *targetLab*. Otherwise the jump instructions to *targetLab* will be skipped, and the code following the `META_IF_BIT` instruction is executed.

`META_BL`(*nullify, targetLab, rpLink, callStr*), is compiled into a `.CALL` *callStr* assembler directive, and a jump instruction to *targetLab* with the specified link register. The argument *nullify* specifies whether or not the delay slot instruction has to be skipped. Note that there always has to be a delay slot instruction after a jump instruction [Pac91a, page 4-10]. Because we decided to let the jump resolver fill in the delay slots, it is a dummy flag in the eyes of the Kit backend to HPPA code compiler, and always considered to be false (do *not* nullify)

`META_BV`(*nullify, dispReg, baseReg*): Code to first calculate the address *dispReg+baseReg* and then jump to the address is generated. Also here *nullify* is a dummy flag, and considered false.

`META_B`(*nullify, targetLab*), Jump instructions to *targetLab* is generated, and again *nullify* is a dummy flag considered false.

The HPPA have two categories of branch instructions, *inter-space* (external) and *intra-space* (local). With inter-space branch instructions we can jump to another address space [Pac91a, page 2-1]. This is not needed in our implementation because the module hpcode.s and the runtime system are placed in the same address space.[23] We do not distinguish between inter- and intra-space calls in this description.

The target address can be computed in one of four different ways:

1. IA-relative (Instruction Address relative) with static displacement (displacement is fixed at compile time). A target label with a displacement of 14 (± 8 KB) or 19 (± 256 KB) bits can be reached.

Instruction name	**Displacement**	**Notes**
`BL`(*tLab, lReg*)	± 256 KB	links with *lReg* register.
`B`(*tLab*)	± 256 KB	Is a `BL` jump with *lReg* = Gr0
`MOVB`(*cond*, r_1, r_2, *tLab*)	± 8 KB	$r_2 = r_1$. Jump if *cond* is met.
`MOVIB`(*cond, immed, r, tLab*)	± 8 KB	$r = immed$. Jump if *cond* is met.
`COMB`(*cond*, r_1, r_2, *tLab*)	± 8 KB	if r_1 *cond* r_2 is satisfied jump to *tLab*.
`COMIB`(*cond, immed, r, tLab*)	± 8 KB	if *immed condition r* is satisfied jump to *tLab*.
`ADDB`(*cond*, r_1, r_2, *tLab*)	± 8 KB	$r_2 = r_1 + r_2$. If $cond(r_2)$ jump to *tLab*.
`ADDIB`(*cond, immed, r, tLab*)	± 8 KB	$r = immed + r$. If $cond(r)$ jump to *tLab*.
`BB`(*cond, r, bitNo, tLab*)	± 8 KB	Test bit *bitNo* in *r*. Jump to *tLab* if *cond* is met.
`BVB`(*cond, r, tLab*)	± 8 KB	Test a bit in *r*.[24]Jumps to *tLab* if *cond* is met.

2. IA-relative with dynamic displacement (displacement contained in a general register).

Instruction name	**Displacement**	**Notes**
`BLR`(*dReg,lReg*)	*dReg* shifted left three bits.	With *lReg* = Gr0 it is a jump without link.

3. Base relative (base address contained in a general register) with dynamic displacement.

[23]The inter-space jumps are for example used to call shared libraries, where the shared code are placed in another address space [Pac91b, chapter 5].

[24]The bit number is given in a special Shift Amount Register (SAR).

Instruction name	Displacement
`BV`(*dReg, bReg*)	*dReg* shifted left three bits.

4. Base relative with static displacement.

Instruction name	Displacement	Notes
`BLE`(*disp,sReg,bReg*)	± 256 KB	`Gr31` is link register. *sReg* is space with target instruction.
`BE`(*disp,sReg,bReg*)	± 256 KB	*sReg* is space with target instruction.

8.3.1 Jump resolving algorithm

The jump resolving algorithm traverses the code twice but with use of backtracking one traversal would suffice. First the two maps

$$\begin{aligned} bMap &\in BlockMap = BlockLabel \xrightarrow{\text{fin}} Addr \\ lbMap &\in LocalBlockMap = LocalBlockLabel \xrightarrow{\text{fin}} Addr \end{aligned}$$

are generated. The two sets *BlockLabel* and *LocalBlockLabel* are the sets of block- and local block labels respectively. To generate the two maps we need another map

$$\begin{aligned} inst &\in Inst = PARISCinst \cup METAinst \\ size &\in Inst \xrightarrow{\text{fin}} InstSize \end{aligned}$$

which given a PARISC or META instruction (*inst*) gives the maximal number of PARISC instructions that *inst* will be compiled into by the jump resolver. For PARISC instructions this is one and for META instructions this can be more than one.

The two maps are then generated by the following semantic function. The first instruction has address zero ($addr = 0$).

$$\mathcal{O}\ [\![[]]\!]\ bMap\ lbMap\ addr = (bMap,\ lbMap)$$

$$\mathcal{O}\ [\![bLab :: C]\!]\ bMap\ lbMap\ addr = \mathcal{O}\ [\![C]\!]\ (bMap + \{bLab \mapsto addr\})\ lbMap\ addr$$

$$\mathcal{O}\ [\![lbLab :: C]\!]\ bMap\ lbMap\ addr = \mathcal{O}\ [\![C]\!]\ bMap\ (lbMap + \{lbLab \mapsto addr\})\ addr$$

$$\mathcal{O}\ [\![inst :: C]\!]\ bMap\ lbMap\ addr = \mathcal{O}\ [\![C]\!]\ bMap\ lbMap\ (addr + size(inst))$$

The second forward pass is explained by the following semantic function:

$$\mathcal{JR}\ [\![[]]\!]\ addr = []$$

```
$\mathcal{JR}\ [\![ \texttt{META\_IF}(cond, r_1, r_2, tLab) :: C ]\!]\ addr =$
    let
      val $js = jumpSize(tLab, addr)$
      val $C' = \mathcal{JR}\ [\![ C ]\!]\ (addr + size(\texttt{META\_IF}(cond, r_1, r_2, tLab)))$
    in
      $(genCode(\texttt{META\_IF}(cond, r_1, r_2, tLab), js, C')$
    end
```

The translation scheme for the other META instructions follow easily from the `META_IF` instruction above. For all $inst \in PARISCinst$ we have:

```
$\mathcal{JR}$ $[\![inst :: C]\!]$  $addr$ =
        let
            val $C' = \mathcal{JR}\ [\![C]\!]\ \ (addr + size(inst))$
        in
            $(inst :: C')$
        end
```

The function *jumpSize* calculates the displacement between the target label and the current instruction address.

$$\begin{aligned} & & Label = BlockLabel \cup LocalBlockLabel \cup TextLabel \\ jumpSize &\in & Label \times Addr \xrightarrow{\text{fin}} Displacement \\ jumpSize(bLab, addr) &= & bMap(bLab) - addr \\ jumpSize(lbLab, addr) &= & lbMap(lbLab) - addr \\ jumpSize(tLab, addr) &= & maxDisplacement \end{aligned}$$

The set *TextLabel* consists of all labels not being block or local block labels. The constant *maxDisplacement* is big enough to have the algorithm choose a long jump instructions.

When the jump size *js* is known *genCode* can generate a list of PARISC instructions for the META instruction. The function *genCode* have the continuation code as the last parameter.

$$genCode \in METAinst \times Displacement \times List(PARISCinst) \xrightarrow{\text{fin}} List(PARISCinst)$$

The *size* function used in $\mathcal{JR}$ is the same used in $\mathcal{O}$ because we to *addr* have to add the same size as used when generating the two maps *bMap* and *lbMap*. If we just add the number of instructions generated by *genCode* we also have to update the two label maps for having a correct displacement.

8.4 Delay slot optimization

When a jump instruction is executed it has a delay slot where the following instruction is executed before the jump is taken. Optionally, the delay slot instruction can be nullified (skipped) or a `NOP` instruction can be inserted in the slot. It is more efficient however, if useful instructions are inserted in delay slots.

After the jump resolving phase `NOP` instructions are located in the delay slots and it is the purpose of the delay slot optimization phase to replace `NOP` instructions with useful instructions. We do not expect large improvements (speedups) by this optimization, so we keep it simple and fast. We traverse the code backwards and each `NOP` instruction is replaced by an instruction further up if possible.

There are a few restrictions to when an instruction may be inserted in a delay slot:

1. The instruction before an instruction to put in a delay slot may not have the opportunity to nullify the instruction to put in the delay slot. For example, in the block

```
BlockLab17
   LDW        4(%sr0, %r5), %r9
   LDW        8(%sr0, %r5), %r10
   COMB,<>,n  %r9, %r10, L362
   LDI        1, %r8  (* must not be moved into delay slot. *)
   B          BlockLab380
   NOP
```

   the instruction `LDI 1, %r8` cannot be put in the delay slot since the instruction `COMB,<>,n %r9, %r10, L362` may optionally nullify the following instruction.

2. The instruction to put in the delay slot must not itself have the opportunity to nullify the following instruction. In the example

```
   COMCLR,=   %r12, %r9, %r1
   B          BlockLab113
   NOP
```

the instruction `COMCLR` can not be moved into the delay slot for the `B` instruction.

3. The delay slot instruction must not define registers used by the branch and vice versa. In the block

```
BlockLab15
   LDW        4(%sr0, %r5), %r8
   LDWM       -4(%sr0, %r30), %r9 (* must not be moved into delay slot *)
   BV         %r0(%r9)            (* because of def/use dependency. *)
   NOP
```

   we could put the `LDW` instruction in the delay slot, but to keep the optimization phase simple it stops finding a substitute when it recognizes the `LDWN` instruction. Instead a good heuristic for a scheduler is to put instructions with no def and use dependencies to the branch instruction, before the branch instruction. After scheduling we would then have the block

```
BlockLab15
   LDWM       -4(%sr0, %r30), %r9
   LDW        4(%sr0, %r5), %r8   (* may be moved in delay slot *)
   BV         %r0(%r9)
   NOP
```

   and the `NOP` instruction could be replaced with the `LDW` instruction by the delay slot optimization phase.

Because we traverse the code backwards we can throw away dead code which is all code following an unconditional branch which cannot be nullified. Because of boolean idiom simplification and the fact that each branch in a Kit backend switch construct ends with a Kit backend jump instruction and since the Kit backend program to PARISC compiler inserts PARISC jump instructions in the branches for Kit backend conditionals, the following code may be produced:

```
LDI        1, %r8
B          BlockLab380
NOP
B          L363  (* dead. *)
NOP              (* dead. *)
```

The last two instructions of the instruction sequence above are dead. In the example below however, the last two instructions are not dead.

```
COMCLR,<>  %r0, %r28, %r1
B          BlockLab113
NOP
B          L1296 (* branch not dead. *)
NOP
```

Below we give the translation function $\mathcal{DSO}$ for delay slot optimization.

$$\mathcal{DSO}\ [\![C]\!]\ = \mathcal{DSO}'\ [\![rev(C), [], []]\!]$$

We traverse the code backwards by using $rev(C)$.

```
$\mathcal{DSO}'$ $[\![$NOP :: $C$, $sinceLastLab$, $result]\!]$ =
    case $C$ of
      BL {$n$, $target$, $t$} :: $C'$ ⇒
        let
          (* C' = before @ (subInst::after), if subInst <> NOP *)
          (* C' = before @ after, if subInst = NOP *)
          val ($before$, $subInst$, $after$) = $\mathcal{F}$ $[\![C', [], $BL$(n, target, t)]\!]$
        in
          if $doesFirstInstNullify$ $C'$ then
            $\mathcal{DSO}'$ $[\![before$ @ $after$, [BL$(n = false, target, t)$,$subInst$] @ $sinceLastLab$, $result]\!]$
          else
            $\mathcal{DSO}'$ $[\![before$ @ $after$, [BL$(n = false, target, t)$,$subInst$], $result]\!]$
        end
      | ··· (* Other instructions with a delay slot. *)
      | _ ⇒ $\mathcal{DSO}'$ $[\![C$, NOP :: $sinceLastLab$, $result]\!]$
```

The test *doesFirstInstNullify*(C') tests if the code in *sinceLastLab* is dead.

```
$\mathcal{DSO}'$ $[\![$LABEL$(lab)$ :: $C$, $sinceLastLab$, $result]\!]$ =
    $\mathcal{DSO}'$ $[\![C$, [], LABEL$(lab)$ :: ($sinceLastLab$ @ $result$)$]\!]$
```

We have a label hence code in *sinceLastLab* is not dead.

```
$\mathcal{DSO}'$ $[\![inst$ :: $C$, $sinceLastLab$, $result]\!]$ =
    $\mathcal{DSO}'$ $[\![C$, $inst$ :: $sinceLastLab$, $result]\!]$
```

Notice that the instruction *inst* has no delay slot.

The function $\mathcal{F}$ finds, if possible, the first instruction above the branch instruction *bInst* that can be put in the delay slot.

```
$\mathcal{F}$ $[\![[], before, bInst]\!]$ = ([], NOP, $rev(before)$)

$\mathcal{F}$ $[\![inst$ :: $C$, $before$, $bInst]\!]$ =
    if $instOkInDelaySlot(inst, bInst)$ and $not$ ($doesFirstInstNullify$ $C$) then
      ($before$, $inst$, $C$)
    else
      if $haveToStop(inst)$ then
        ([], NOP, $rev(before)$@($inst$::$C$))
      else
        $\mathcal{F}$ $[\![C$, $inst$ :: $before$, $bInst]\!]$
```

The function *instOkInDelaySlot*(*inst*, *bInst*) tests for 2 and 3 on page 33 and the function *doesFirstInstNullify* tests for 1 on page 32. The function *haveToStop*(*inst*) returns *false* if *inst* is an ordinary PARISC instruction and *true* if an assembler directive.

In the current implementation it is only necessary to optimize on `B` and `BV` instructions. When for example compiling *life* (see section 9) the optimization results in 162 delay slots for `B` instructions to be filled out of 195. Due to 3 on page 33 no instructions are inserted in delay slots of `BV` instructions, though 231 `BV` instructions exist. The 33 `B` instructions for which the delay slots have not been filled are present because of 1 on page 32 or since no delay slot instruction is available as in the following example:

```
LocalLabel423
                B     BlockLab350
                NOP
LocalLabel417
```

Such blocks could be in-lined, if possible, by an earlier optimization phase.

8.5 Instruction scheduling

We have not yet implemented instruction scheduling in this backend. Instruction scheduling is currently being developed as a part of another project. The ideas used there are mainly those described in [JM86] and [GM86]. A few things are different though. It is not possible to make the same assumptions about code generation as in [GM86]. When building the dependency DAG [GM86, page 12] we have to serialize all load and store instructions, even though there are no register dependencies between the instructions. In the small instruction sequence

```
LDW  0(%sr0, %r20), %r20
STW  %r23, 4*32(%sr0, %r4)
```

we can not assume that the address `0(%sr0, %r20)` is different from the address `4*32(%sr0, %r4)`.

Because of the delay slot optimization phase a good heuristic will be to have the instruction placed before a branch, to not interlock with the branch. Another heuristic will be to perform peephole optimization on the instructions in the *candidate* list [GM86, page 13].

9 Experiments and Tests

We have performed runtime tests and measurements for speed and memory usage for six different programs both for the region based ML Kit compiler and the Standard ML of New Jersey compiler. In section 9.2 we compare the test results obtained for the two compilers and in section 9.3 we investigate the effect of various optimizations. Statistics on regions for the six programs are shown in section 9.4. First, in the following section, we show as an example how a simple Fibonacci program is compiled and how it is represented at different stages in the compiler.

9.1 Compilation of a simple Fibonacci program

The Standard ML source code for a simple Fibonacci program is as follows:

```
let
  infix + - <
  fun fib n = if n < 1 then 1 else fib(n-1) + fib(n-2)
in
  fib 35
end;
```

This program is parsed, elaborated and compiled into a typed lambda language as shown in figure 1. After storage mode analysis the annotated lambda program looks as follows[25]:

```
let val it=
  letregion r2
  in let fun fib[r3*] =
       fn v25 =>
       letregion r4
       in let val v27=
            letregion r5
            in prim(<, [v25,1 atbot r5]) atbot r4
            end
          in switch v27
            of true => 1 sat r3*
             | false =>
                letregion r6, r7
                in prim(+, [letregion r8, r9
                            in fib(*r7*) [atbot] atbot r9
                              letregion r10
                              in prim(-, [v25,1 atbot r10]) atbot r8
```

[25]The output from the pretty-printer has been modified slightly. For example, most type expressions have been left out.

```
                                  end
                                end,
                                letregion r11, r12
                                in fib(*r6*) [atbot] atbot r12
                                  letregion r13
                                  in prim(-, [v25,2 atbot r13]) atbot r11
                                  end
                                end]) attop r3*
                     end
                 end
               end at r2
             in letregion r14, r15
                in ((fib(*r1*) [atbot] atbot r15) 35 atbot r14)
                end
             end
           end
         in {|it: (int,r1)|} attop r1
         end
```

The above program is now compiled by the lambda compiler using information obtained during physical size inference into a Kit backend program. After various optimizing transformations and register allocation (see figure 2) the program looks as follows:

```
LABEL 1:                                        // Start label.
   AllocRegion(R9);                             // Allocate global region.
   Move(R9,V35);
   Move(R30(stacktop),R9);                      // Setup top level handler.
   Push(Lab4);
   Push(R9);
   Push(R3(exnPtr));
   Move(R30(stacktop),R3(exnPtr));
   Push(Lab2);                                  // Push return address.
   Jmp(Lab3)                                    // Jump to main.
LABEL 3:
   Move(R30(stacktop),R9);
   Offset(R30(stacktop),4,R30(stacktop));       // Allocate storage for
   Move(V35,R10);                               // region closure.
   StoreIndexL(R10,R9,0);
   Push(Lab6);
   Move(71,R5(stdArg));                         // Load argument (35).
   Move(R9,R7(stdClos));
   Jmp(Lab5)                                    // Jump to function.
LABEL 4:
   FetchIndexL(R5(stdArg),0,R5(stdArg));        // Top level handler.
   FetchIndexL(R5(stdArg),1,R5(stdArg));
   Die(R5(stdArg))
LABEL 5:
   if R5(stdArg) < 3 then
      Move(3,R8(stdRes)); Jmp(Lab7)
   else
      Push(R5(stdArg));
      Push(R7(stdClos));
      Push(Lab9);
      Subi(R5(stdArg),3,R5(stdArg));
      Jmp(Lab5)                                 // fib(n-1)
   endif
LABEL 6:
   Offset(R30(stacktop),~4,R30(stacktop));
   Move(V35,R9);
   AllocMemL(R9,1,R9);                          // Allocate storage for
   StoreIndexL(R8(stdRes),R9,0);                // result in global region.
   Move(R9,R8(stdRes));                         // Save framed result.
   Pop(R9);                                     // Return.
   Jmp(R9)
```

```
LABEL 7:
   Pop(R9);
   Jmp(R9)
LABEL 8:
   Pop(R9);
   Addi(R9,R8(stdRes),R8(stdRes));          // +.
   Jmp(Lab7)
LABEL 9:
   Pop(R7(stdClos));
   Pop(R5(stdArg));
   Push(R8(stdRes));
   Push(Lab8);
   Subi(R5(stdArg),5,R5(stdArg));
   Jmp(Lab5)                                // fib(n-2).
```

The Kit backend instruction set is described in section 4.3. Notice that in-lining of blocks has not been considered in the current backend. In the above example it would be a good idea to in-line block seven in block eight and block five, since block seven only includes two small instructions. Also notice, how integers (represented as $2 * i + 1$) are propagated to arguments in instructions. Further notice, that because of boolean idiom simplification the original switch construct in block five has been translated into an if construct. Hence only a single test is generated for $<$ in block five.

9.2 Comparison against the SML/NJ compiler

In this section we compare the region based ML Kit compiler against the Standard ML of New Jersey compiler version 0.93. Measurements on the Standard ML of New Jersey compiler are performed by first exporting a compiled function by use of `exportFn`. All measurements are performed on a machine equipped with a HP RISC 735 processor and 256 Mb of RAM. The runtime system for the region based ML Kit compiler is compiled using the HP `cc` compiler with option `+O4`.

Various kinds of optimizations may be disabled/enabled in the region based ML Kit compiler. When comparing the region based compiler against the Standard ML of New Jersey compiler the following optimizations/transformations are enabled:

- Copy propagation.
- Dead code elimination.
- Register allocation.
- In-lining of allocation and deallocation of regions.
- In-lining of allocation in regions.
- Delay slot optimization.

It is important to notice that our goal here is not to perform better than the Standard ML of New Jersey compiler in general. This would indeed be a very hard task. Instead, we hope to show that "compiling with regions" is a promising future alternative.

9.2.1 Test programs

Measurements are shown for the six different programs described below.

`tmergesort` is a program for sorting 50.000 pseudo-random integers.

`fib35` is the simple Fibonacci program shown in section 9.1.

`life` is a program that three times computes 50 generations of The game of Life and prints out the result. Originally written by Chris Reade [Rea89] and later modified slightly by Mads Tofte to increase memory performance.

`knuth-bendix` is an implementation of the Knuth-Bendix completion algorithm, processing some axioms of geometry (see [App92, page 179]). It has been modified slightly by Mads Tofte to increase memory performance. The completion is computed three times.

`mandelbrot` computes a dense Mandelbrot set three times. This program uses floating points extensively.

`simple` is the largest program of the six. It is a spherical fluid-dynamics program, originally developed as a *realistic* FORTRAN benchmark. It has been translated into Standard ML and perform a series of complex floating point operations. No modifications has been made to increase memory performance.

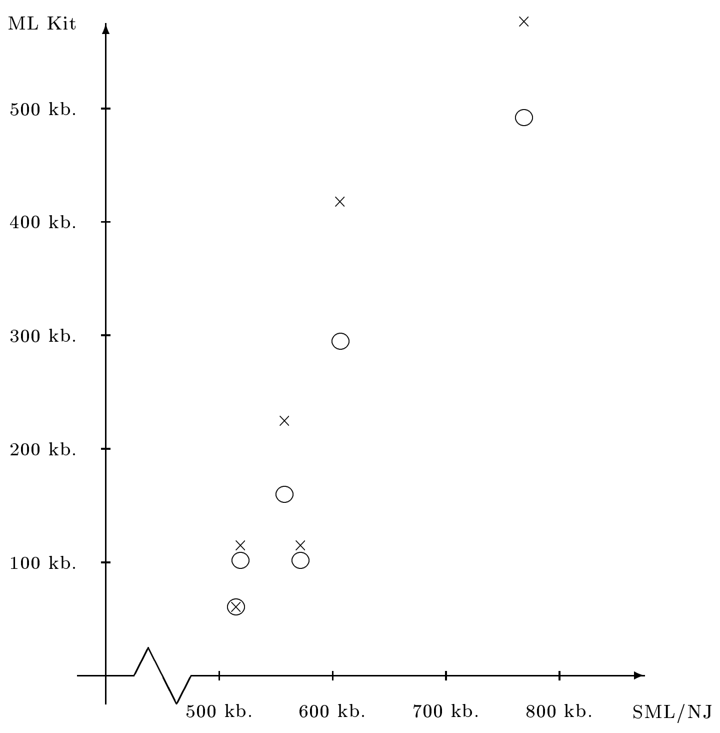
9.2.2 Program sizes

The sizes of the executable object files are listed in the table below.

Program size	**ML Kit I**	**ML Kit II**	**SML/NJ**
`tmergesort`	115 kb.	102 kb.	572 kb.
`fib35`	61 kb.	61 kb.	515 kb.
`life`	225 kb.	160 kb.	558 kb.
`knuth-bendix`	418 kb.	295 kb.	607 kb.
`mandelbrot`	115 kb.	102 kb.	519 kb.
`simple`	577 kb.	492 kb.	769 kb

The column `ML Kit I` shows results obtained when fast code is generated on the cost of program size (in-lining of region manipulation primitives) whereas the column `ML Kit II` shows results obtained when small code is generated on the cost of running time.

When taking into account that the runtime system of the Standard ML of New Jersey system is considerably larger than the runtime system of the region based ML Kit compiler it turns out that the ML Kit compiler generates about twice the amount of code as the Standard ML of New Jersey compiler. The graph below shows the size of the executable files generated by the region based compiler as a function of the size of the executable files generated by the Standard ML of New Jersey compiler.



Circles show the results from column `ML Kit II` in the table above, whereas crosses show results from column `ML Kit I`.

9.2.3 Running time

The running time is for each program measured using the UNIX `time` command[26]. The time measurements are listed below.

Running time	**ML Kit**	**SML/NJ**	**Percentage**
`tmergesort`	2.8	2.7	103.7
`fib35`	10.8	27.9	38.7
`life`	11.5	7.9	145.6
`knuth-bendix`	32.4	27.8	116.5
`mandelbrot`	43.4	24.9	174.3
`simple`	62.2	17.4	357.5

The percentages show running time spend by the ML Kit executables compared to running time spend by the SML/NJ executables. Except for the Fibonacci program the region based ML Kit compiler is slower than the Standard ML of New Jersey system. Only for the program `simple` that uses floating point operations extensively is the ratio larger than two.

9.2.4 Memory usage

Maximal memory usage is measured using the UNIX program `top`. The two different values for given entries in the table below denote the maximum of the total size of the process and the maximum of the resident amount of memory used, respectively.

Memory usage	**ML Kit**	**SML/NJ**
`tmergesort`	4.1 Mb. / —	6.4 Mb. / —
`fib35`	80 kb. / 92 kb.	1.7 Mb. / 1.0 Mb.
`life`	376 kb. / 292 kb.	1.9 Mb. / 1.6 Mb.
`knuth-bendix`	4.2 Mb / 4.0 Mb	2.5 Mb / 2.3 Mb.
`mandelbrot`	364 kb. / 352 kb.	1.7 Mb. / 852 kb.
`simple`	2.4 Mb. / 2.1 Mb.	2.9 Mb. / 2.2 Mb.

Since region inference is not always good at predicting when deallocation of memory is possible it may be necessary to modify a program manually to avoid space leaks. Such modifications have been made for the programs `tmergesort`, `life`, `knuth-bendix` and `mandelbrot` whereas no modifications have been made for the other programs.

As mentioned in section 5 important optimizing transformations such as argument flattening is not performed by the region based ML Kit compiler. Also, region variables are given as parameters to region polymorphic functions at runtime. It is not clear how this in general affects the object code size and the running time. Despite these facts the compiler generates fairly good code when compared to the Standard ML of New Jersey compiler.

9.3 Optimizations

In this section we investigate how various optimizing transformations improve the quality of the code generated by the region based ML Kit compiler. Tests are performed for each of the six programs presented in section 9.2.1. We investigate both improvement in code size and improvement in running time. Since copy propagation is closely related to dead code elimination we do not divide these optimizing transformations in the presentation of the test results. Below we show the configurations of the measured versions of the compiler, gradually improving (see also figure 2):

[26]For all measurements performed using the UNIX time command the user time (in seconds) is used.

ML Kit I: No optimizing transformations applied.

ML Kit II: Register allocation. No further optimizing transformations.

ML Kit III: Register allocation, copy propagation and dead code elimination.

ML Kit IV: Register allocation, copy propagation, dead code elimination and delay slot optimization.

Notice that in-lining of allocation/deallocation of regions and deallocations in regions is enabled in all versions of the compiler. Since absolute performance data of the various tests tends to emphasize the architecture rather than the optimizing transformations (at least regarding running time performance), we express effectiveness of an optimizing transformation as an *improvement* over the unoptimized, or *baseline*, counterpart (in all cases ML Kit I). Percentage of improvement is calculated by the formula $(1 - \frac{optimized}{baseline}) * 100$ (see [JM86]).

Results of the measurements for program size improvement is given in the table below. Notice that we by program size mean the size of the executable program (including the runtime system).

Program size	**ML Kit I**	**ML Kit II**	**ML Kit III**	**ML Kit IV**
`tmergesort`	0 %	3 %	7 %	7 %
`fib35`	0 %	0 %	0 %	0 %
`life`	0 %	13 %	14 %	14 %
`knuth-bendix`	0 %	21 %	23 %	23 %
`mandelbrot`	0 %	7 %	7 %	7 %
`simple`	0 %	25 %	27 %	27 %

Since the runtime system is a constant part of the executable code the improvement increases when the total program size increases.

Results of the measurements for running time improvement is given below.

Running time	**ML Kit I**	**ML Kit II**	**ML Kit III**	**ML Kit IV**
`tmergesort`	0 %	50 %	53 %	53 %
`fib35`	0 %	47 %	62 %	64 %
`life`	0 %	57 %	60 %	61 %
`knuth-bendix`	0 %	49 %	50 %	50 %
`mandelbrot`	0 %	47 %	54 %	54 %
`simple`	0 %	48 %	58 %	59 %

Register allocation is the far most important optimizing transformation in the new backend for the region based compiler, both regarding executable program size and running time performance. Also copy propagation and dead code elimination are valuable optimizing transformations, whereas almost no improvement has been gained due to delay slot optimization.

9.4 Region statistics

For each of the six different programs presented in section 9.2.1 we have performed some statistics on the use of regions. In the table below the field `AllocRegion` shows the number of times regions are allocated and deallocated at runtime. When the program has finished every non global region that has been allocated is also deallocated.

Statistics	`tmergesort`	`fib35`	`life`	`knuth-bendix`	`mandelbrot`	`simple`
`AllocRegion`	300.027	1	219.385	6.037.399	56	136.529
`AllocMemL`	2.684.231	1	2.184.803	8.686.886	346	728.716
`ResetRegion`	300.002	0	51.739	8.184	6	26
`Max.regions`	99	1	189	614	10	76
`Mem.regions`	2.9 Mb.	24 kb.	189 kb.	3.9 Mb.	24 kb.	2.0 Mb.

The field `AllocMemL` shows the number of allocations in all regions whereas the field `ResetRegion` shows the number of times regions have been reset. The field `Max.regions` shows the maximal number of regions that have been live at the same time during execution whereas the field `Mem.regions` shows the maximal amount of memory used by regions.

No attempt has been made to investigate how the stack behaves during execution. To understand in practice where important space-leaks are located in a specific program better profiling tools is of great importance.

10 Acknowledgements

Thanks to Tommy Højfeld Olesen, Martin Koch, Mads Tofte, Lars Birkedal and Henning Niss for valuable discussions. Also, the work described in this report could not have been done without previous work on the ML Kit by Nick Rothwell, Lars Birkedal, David N. Turner and Mads Tofte.

11 Conclusion and Further Work

In this paper we have described the implementation of a backend for a region based Standard ML compiler. Various optimizations and register allocation performed on an intermediate language have been described. Further, we have described how primitives for manipulating a stack of regions are implemented together with other primitives supported by a Standard ML Compiler.

We have shown that optimizing transformations performed in this backend improves the generated code significantly both with respect to program size and with respect to execution time. It is still possible though, to improve on the generated code by further optimization.

There are several possibilities for future work. First, it is an important feature of a compiler to support incremental compilation and since most of the compiler already supports incremental compilation it does not seem too difficult to implement. Second, it is not at present possible to compile phrases of the Modules language and it is not clear at the moment how functors may be compiled separately in a region based compiler. As an alternative to separate compilation of functors, functors may be unfolded (specialized) for every application [Els95].

Third, different analyses such as box/unbox analysis [Ler92,HJ94] may be implemented to increase the quality of the generated code. Finally, there are many phases of the compiler for which more efficient algorithms need to be applied for the compiler to be more efficient.

Though the region based compiler described in this paper is not competitive to some other industrialized Standard ML compilers regarding tools and features, experiments suggest (see section 9), that “compiling with regions” is a promising alternative.

References

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. Technical report, Princeton University and AT&T Bell Laboratories, 1991. Documentation of the Standard ML of New Jersey compiler (v. 2).

[App92] Andrew W. Appel. *Compiling With Continuations.* Cambridge University Press, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Ber91] Dave Berry. The Edinburgh SML library. Documentation and description of the Edinburgh SML Library, April 1991.

[Bir94] Lars Birkedal. The ML Kit compiler – working note. Technical report, DIKU, Department of Computer Science, University of Copenhagen, Juli 1994. An overview of important design decisions in the ML Kit compiler.

[Bra95] Marc Michael Brandis. *Optimizing Compilers for Structured Programming Languages.* PhD thesis, Swiss Federal Institute of Technology Zurich, 1995.

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit version 1. An implementation of Standard ML written in Standard ML, March 1993.

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 85–94, June 1982.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 192–203, June 1991.

[Els95] Martin Elsman. Unfolding modules in a type based Standard ML compiler. Department of Computer Science (DIKU), University of Copenhagen, June 1995.

[GM86] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *SIGPLAN Symposium on Compiler Construction*, June 86.

[HJ94] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Portland, Oregon*, pages 213–226, January 1994.

[JM86] Mark Scott Johnson and Terrence C. Miller. Effectiveness of a machine-level, global optimizer. In *SIGPLAN Symposium on Compiler Construction*, June 86.

[Ler90] Xavier Leroy. The ZINC experiment: An economical implementation of the ML language. Technical report, INRIA, February 1990.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Principles of Programming Languages*, pages 177–188, 1992.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Pac91a] Hewlett Packard. *PA-RISC Assembly Language Reference Manual.* Hewlett Packard, January 1991. Fourth edition.

[Pac91b] Hewlett Packard. *PA-RISC Procedure Calling Conventions Reference Manual.* Hewlett Packard, January 1991. Second edition.

[Pac94] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Hewlett Packard, February 1994. Third edition.

[Rea89] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

[SA94] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. Technical report, Yale University and Princeton University, November 1994.

[Ses91] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, University of Copenhagen, Department of Computer Science, October 1991.

[Tof94a] Mads Tofte. Space profiling of region-based programs. This note descibes space profiling of region based programs, August 1994.

[Tof94b] Mads Tofte. Storage mode analysis. This note descibes a storage mode analysis developed by Birkedal and Tofte and used in the region based ML Kit compiler, October 1994.

[TT93] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical report, University of Copenhagen and European Computer-Industry Research Center, July 1993.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *21st ACM Symposium on Principles of Programming Languages*, January 1994.

A Using The System

This appendix describes shortly how the region based Standard ML compiler for the HPPA architecture is build and how the system may be used in practice.

A.1 Building the compiler

The ML Kit is built using the Make system of the Edinburgh library [Ber91]. For this purpose a *consult file* containing all names of the source files is needed. Notice that it is a prerequisite that the Edinburgh library has been loaded before building the ML Kit.

When the file ML_CONSULT_HPPA is the *consult file* mentioned above, the following declarations will build the ML Kit:

```
- open Make;
- setTempfile "%Make.tmp.hp%";
- loadFrom "ML_CONSULT_HPPA";
- make "K";
```

For convenience, when using the Standard ML of New Jersey compiler the following declarations may also be appropriate:

```
- SML_NJ.Print.signatures := 0;
- SML_NJ.Control.Runtime.softmax := 50000000;
```

A.2 The runtime system

The runtime system consists of two files runtimeHPPA.h and runtimeHPPA.c. In the include file it is possible to set some flags to control the behaviour of the runtime system:

CHECKREGION: The flag is used by alloc and if set alloc will make sure that there are space in the current region page for the words to be allocated. This flag should always be set.

STAT: If this flag is set the runtime system automatically collects statistics on the regions. Statistics only works correctly when all region manipulations are performed by functions in the runtime system (the user must disable in-lining flags in the ML Kit).

FREESTAT: This flag requires STAT to be set and when set the free-list is printed each time statistics are printed (see section 7.6)

REGION_PAGE_SIZE: This constant holds the number of words in each region page.

BYTES_ALLOC_BY_SBRK: This constant holds the number of bytes that are allocated by sbrk. It must be equivalent to a whole number of region pages.

The runtime system is compiled with either HP's own C compiler by `cc -Aa -c runtimeHPPA.c` or by the Gnu C compiler by `gcc -ansi -c runtimeHPPA.c`.

A.3 Compiling programs

Since separate or incremental compilation is not yet available one need to insert an *ML prelude* in the beginning of the Standard ML Core program to compile[27]. This prelude defines all primitive operations as described in [MTH90, appendix C].

[27] Modules are not yet supported.

Compiler flags

To control compilation it is possible to adjust a number of different flags. In particular, it is possible to enable optimizations such as copy propagation and dead code elimination. There are also flags for controlling in-lining of various region instructions. In-lining of region function calls speeds up the generated code on the cost of increased code size. Flags may be controlled using the ML Kit function `interact()`.

Files and directories

To change the file name of the target assembler code write:

```
  - Flags.ccode_filename := "/.../myfile.s";
```

Similarly, to change the directory name of the *source* directory (where source files are read from) write:

```
  - Flags.directory := "/.../mydirectory/";
```

Compilation

To compile a source file myfile.sml including a Core language declaration (also holding the *prelude* mentioned above) write:

```
  - evalFile "myfile";
```

As a result a target assembler file is constructed.

Assembling, linking and execution

In order to assemble a file (hpcode.s, say) generated by the backend and to link the object code to the runtime system, HP's C compiler may be used:

```
  > cc runtimeHPPA.o hpcode.s -lm
```

As a result an executable file a.out is constructed.

B HPPA RISC Instructions Used

In this appendix we show the PARISC instructions used in the translation of the Kit backend program. For each instruction we present a short description and show the registers which are defined and used, respectively. We do not show all status registers in the def and use columns. Also, we show a few instruction sequences used in the backend. The META[28] and branch instructions are also explained in section 8.3.

In table 3 we present the ordinary fixed point, load, store and branch instructions.

Instruction name	**Description**	**Reg. def.**	**Reg. used**
ADD$(cond, r_1, r_2, t)$	$t = r_1 + r_2$	t	r_1, r_2
ADDI$(cond, i, r, t)$	$t = i + r$	t	r
ADDIL(i, r)	Gr1 $= r + i$	Gr1	r
AND$(cond, r_1, r_2, t)$	$t = r_1$ AND r_2	t	r_1, r_2
ANDCM$(cond, r_1, r_2, t)$	$t = r_1$ AND $\tilde{r}_2$ (complement to r_2)	t	r_1, r_2
B$(n, target)$	$Jmp(target)$		
BB$(n, cond, r, p, target)$	if $cond$(bit p in r) then $Jmp(target)$		r
BL$(n, target, t)$	$Jmp(target), t = Addr(nextI)$	t	
BLE(n, wd, sr, b)	$JmpE(sr, b + wd)$		b
BV(n, x, b)	$Jmp(M[b + x])$		b, x
COMB$(cond, n, r_1, r_2, target)$	if r_1 $cond$ r_2 then $Jmp(target)$		r_1, r_2
COMCLR$(cond, r_1, r_2, t)$	r_1 $cond$ r_2, t=0	t	r_1, r_2
COPY(r, t)	$t = r$	t	r
DEPI$(cond, i, p, len, t)$	$t = SignExt(i, p, len)$	t	t
LDI(i, t)	$t = i$	t	
LDIL(i, t)	$t = i$	t	
LDO(d, b, t)	$t = b + d$	t	b
LDW(d, s, b, t)	$t = M[s, b + d]$	t	b
LDWM(d, s, b, t)	$t = M[s, b(+d)^{29}], b = b + d$	b, t	b
NOP()			
OR$(cond, r_1, r_2, t)$	r_1 OR r_2		r_1, r_2
SH2ADD$(cond, r1, r2, t)$	$t = shiftLeft_2(r_1) + r_2$	t	r_1, r_2
SHD$(cond, r_1, r_2, p, t)$	$t = ShiftRight(concat(r_1, r_2), p)$	t	r_1, r_2
SUB$(cond, r_1, r_2, t)$	$t = r_1 - r_2$	t	r_1, r_2
SUBI$(cond, i, r, t)$	$t = r - i$	t	r
STW(r, d, s, b)	$M[s, b + d] = r$		b, r
STWM(r, d, s, b)	$M[s, b(+d)] = r, b = b + d$	b	b, r

Table 3: *The PARISC instructions used in the backend.*

In table 3 we use t as target register, r and x as operand registers, i as immediates, $target$ as target labels, p and len as small constants, b as base register, d and wd as immediate displacements, sr and s as space registers and $nextI$ as the instruction following the current one. The functions used are *Jmp* which jumps to the target label, *JmpE* which jumps external (to another address space), *Addr* that returns the address of an instruction, *SignExt* which sign extends a given word, $shiftLeft_2$ which shifts a register two bits to the left, *ShiftRight* which shifts two concatenated registers right and *concat* that concatenates two registers. Memory is represented by the map M which maps addresses to values. An address can be given with or without a space register.

For all instructions containing the *cond* variable, the next instruction is nullified, if the condition is satisfied. The possible conditions are given below, but not all conditions can be used for each instruction.

[28]The META instructions are not PARISC instructions, but used in the translation process from the Kit backend program to the final PARISC program.

[29]If $d < 0$ then $b + d$ else b. Also used in STWM to push and pop, where the base register is the stack pointer. The stack pointer points at the next available address.

- NEVER
- ALWAYS
- EQUAL
- NOTEQUAL
- GREATERTHAN
- GREATEREQUAL
- LESSTHAN
- LESSEQUAL
- ODD
- EVEN

A large constant (32 bit) is loaded into a register by the instructions

```
LDIL  L'constant, %r1
LDO   R'constant(%r1), %r20
```

where `L'` and `R'` means the left and right part of the constant, respectively. To push and pop an address on the machine stack the following instructions are used (`%r30` is the stack pointer)

```
STWM  %r26, 4(%sr0, %r30)  ; Push(%r26). Increment %r30 afterwards.
LDWM  -4(%sr0, %r30), %r19 ; Pop(%r19), but first decrement %r30.
```

An external call (to the procedure `callSbrk`) is implemented by the instruction sequence

```
.CALL
LDIL  L'callSbrk, %r1
BLE   R'callSbrk(%sr4, %r1)
COPY  %r31, %r2
```

where `%r31` is link register for the `BLE` instruction and therefore copied to `%r2`. To load the address of a label into a register the intruction sequence

```
ADDIL L'topRegion-$global$, %r27
LDO   R'topRegion-$global$(%r1), %r20
```

can be used (address of label `topRegion` is loaded into `%r20`). For compatibility purposes with the HP-UX operating system the constant `$global$` and `%r27` are used [Pac91a, page 2–7].

Table 4 presents the floating point instructions used in the backend. Only a small number of the available PARISC floating point instructions are used.

Instruction name	**Description**	**Reg. def.**	**Reg. used**
FABS(fmt, r, t)	$t = \|r\|$	t	r
FADD(fmt, r_1, r_2, t)	$t = r_1 + r_2$	t	r_1, r_2
FCMP($fmt, cond, r_1, r_2$)	$FStatusReg = r_1 \ cond \ r_2$	$FStatusReg$	r_1, r_2
FLDDS($complt, d, s, b, t$)	$t = M[s, b(+d)], (b = b + d)$	b, t	b
FMPY(fmt, r_1, r_2, t)	$t = r_1 * r_2$	t	r_1, r_2
FSTDS($complt, r, d, s, b$)	$M[s, b(+d)] = r, (b = b + d)$	b	b, r
FSUB(fmt, r_1, r_2, t)	$t = r_1 - r_2$	t	r_1, r_2
FTEST()	If FTEST(*FStatusReg*) then *nullify*(*nextI*)		$FStatusReg$
XMPYU(r_1, r_2, t)	$t = r_1 *' r_2$	t	r_1, r_2

Table 4: *The PARISC floating point instructions used in the backend.*

The variable *complt* can be any of the constants `EMPTY`, `MODIFYBEFORE` or `MODIFYAFTER`. The instruction *complt* determines if the base register b have to be modified. For `FLDDS` we have $t = M[s, b + d]$ if we use `MEDIFYBEFORE` and $t = M[s, b]$ if we use `EMPTY` or `MODIFYAFTER`. The variable *fmt* can be any of the constants `DBL`, `SGL` or `QUAD`. It specifies the witdth of the source and destination registers (`SGL` is 32 bit, `DBL` is 64 bit and

QUAD is 128 bit). The $*'$ operator is unsigned fixed point multiplication, where the floating point source and destination registers are interpreted as unsigned 32 bit integers.

Table 5 shows the META instructions which are used in our optimization phases and later removed by the jump resolver.

Instruction name	**Description**
META_IF($cond, r_1, r_2, target$)	If not (r_1 $cond$ r_2) then $Jmp(target)$
META_BL(n, $target$, $rpLink$, $callStr$)	$Jmp(target)$
META_BV(n, x, b)	$Jmp(M[b+x])$
META_IF_BIT(r, $bitNo$, $target$)	if not ($bitNo$ in r set) then $Jmp(target)$
META_B(n, $target$)	$Jmp(target)$

Table 5: *The META instructions used in the translation from the Kit backend program to to the PARISC program.*

Instruction name	**Description**
LABEL(lab)	Label lab is introduced
COMMENT(s)	Comment s is introduced
NOT_IMPL(s)	Notification that s is not implemented
DOT_ALIGN(i)	Next address is i bytes aligned (.ALIGN i)
DOT_BLOCKZ(i)	Reserves i zeroed bytes (.BLOCKZ i)
DOT_CALL(s)	Introduce .CALL s
DOT_CALLINFO(s)	Introduce .CALLINFO s
DOT_CODE()	Introduce .CODE (unmodifyable code segment)
DOT_DATA()	Introduce .DATA (modifyable data segment)
DOT_DOUBLE(s)	Reserves a double word floating point value (.DOUBLE s)
DOT_END()	Introduce .END
DOT_ENTER()	Introduce .ENTER
DOT_EXPORT(seg, sym)	Introduce .EXPORT *seg, sym*
DOT_IMPORT(sym, seg)	Introduce .IMPORT *sym, seg*
DOT_LEAVE()	Introduce .LEAVE
DOT_PROC()	Introduce .PROC
DOT_PROCEND()	Introduce .PROCEND
DOT_STRINGZ(s)	Reserves a string s (.STRINGZ s)
DOT_WORD(w)	Reserves a word w (.WORD w)

Table 6: *The assembler directives used in the translation from the Kit backend program to to the PARISC program.*

Note that NOT_IMPL(s) in table 6 is printed as a comment, and COMMENT(s) is pretty printed as ;s.

To allocate global data the following instruction sequence can be used:

```
topRegion         ; Variable topRegion is allocated.
        .ALIGN 4
        .BLOCKZ 4
$PRINTF_DATA$     ; Global string $PRINTF_DATA$ is allocated.
        .ALIGN 4
        .STRINGZ "   KamReg %3d = %d \n"
        .DATA
        .ALIGN 4
StringLabel20     ; A ML Kit string is allocated
        .WORD 145 ; Length and tag for string.
        .WORD 18  ; Length of first string fragment.
        .WORD 0   ; No next string fragment
        .STRINGZ "starting printing\n"
        .DATA
        .ALIGN 8
FloatLabel9         ; A ML Kit float is allocated.
        .WORD 0     ; Tag for floats
        .WORD 0     ; Alignment word
        .DOUBLE 0.0 ; The value 0.0.
```

C Source Files for the Backend

In this appendix we show all files that make up the Kit backend for the PARISC architecture. Generally all files are followed by a version number, so for *../src/Compiler/Cfg/TransformCfg.sml* the current file will look like *../src/Compiler/Cfg/TransformCfg24u.sml* where the version is *24u*. Files including signatures are written with large letters and files including functors with large and small letters.

File name	**Description**
ARCHITECTURE_INFO.sml	Describe the architecture (number of physical registers, ...)
CFG_INST.sml	Contain the Kit backend instructions.
CfgInst.sml	Contain the Kit backend instructions.
COLOR_GRAPH.sml	The color function used by the register allocation.
ColorGraph.sml	The color function used by the register allocation.
TRANSFORM_CFG.sml	Functions for optimizations on the Kit backend program.
TransformCfg.sml	Functions for optimizations on the Kit backend program.
CfgKitAM.sml	Matches KIT_ABSTRACT_MACHINE in *../src/Compiler/Kam/*

Table 7: *The files in directory ../src/Compiler/Cfg/*

File name	**Description**
KAM_BACKEND.sml	All backends have to match this signature.
KIT_ABSTRACT_MACHINE.sml	The lambda language is compiled into this machine.

Table 8: *Some of the files in directory ../src/Compiler/Kam/*

File name	**Description**
CFG_TO_HPPA.sml	Compiler from Kit backend programs to PARISC.
CfgToHpPa.sml	Compiler from Kit backend programs to PARISC.
HPPA_RISC.sml	PARISC instructions and registers.
HpPaRisc.sml	PARISC instructions and registers.
HpPaInfo.sml	Match ARCHITECTURE_INFO in *../src/Compiler/Cfg.*
HppaKAMBackend.sml	Match KAM_BACKEND in *../src/Compiler/Kam.*
RESOLVE_JUMPS.sml	Resolve jumps optimization phase.
ResolveJumps.sml	Resolve jumps optimization phase.
DELAY_SLOT_OPTIMIZATION.sml	Delay slot optimization phase.
DelaySlotOptimization.sml	Delay slot optimization phase.
Runtime/runtimeHPPA.c	Main source file for the runtime system.
Runtime/runtimeHPPA.h	Include file used in runtimeHPPA.c

Table 9: *Files in directory ../src/Compiler/Hppa/*