SMLNJ: Garbage collection API

Lal George
Rm 2c478, Bell Labs
700 Mountain Ave
Murray Hill, NJ 07974

May 21, 1999

This note describes the garbage collection interface to the runtime system implemented in version 110.15. All communication to the runtime system is through a fixed set of registers. Anything extra must be bundled up and saved in one of these fixed registers. This simplifies the runtime system significantly, reduces the ML to C context switch overhead, and barely increases the size of code generated to invoke garbage collection. The net result is a 4% improvement in compiling the compiler on the DEC Alpha.

1 CPS functions

There are three kinds of CPS functions generated by the SML/NJ compiler.

Standard: corresponds to functions exported by a module.

Continuation: corresponds to continuation functions generated by the CPS compilation[**?**].

Known: are functions whose calls sites are statically known.

Standard and continuation functions are said to be *escaping* since all their call sites are not statically known.

2 CPS machine

The CPS functions execute on an abstract machine with the following registers:

Name	**Dedicated**	**Description**
allocptr	Yes	Allocation pointer
limitptr	Yes	Limit Pointer
varptr	Yes	Var Pointer
exnptr	Yes	Current exception handler
baseptr	Yes	Code base pointer
stdlink	No	Standard Link
stdclos	No	Standard Closure
stdarg	No	Standard Argument
stdcont	No	Standard Continuation
$\text{misc}_{0..2}$	No	Callee save registers
$\text{misc}_{3..M}$	No	miscellaneous registers
exhausted	No	flag
floatregs	No	floating point registers

`alloctpr` is a pointer to the next available location in the allocation space.

`limitptr` is the upper limit on the allocation space. There is a 4096 byte skid pad beyond the `limitptr` that can be written into.

`varptr` is used for thread specific state.

`exnptr` is a pointer to a closure containing the current exception handler.

`baseptr` is used to compute the values of labels that reside in position independent code.

`stdlink` contains the code address of the function being called during a call to a standard function.

`stdcont` contains the code address to resume computation, or the code address during a call to a continuation function.

stdclos contain the environment or closure required during a call to a standard function.

stdarg is an argument register.

$\texttt{misc}_{0..M}$ is a set of miscellaneous registers, the first three of which are treated as calleesave registers.

exhausted is set when there is no more allocation space available. This register is set before a call and examined by the callee.

floatregs is a set of double precision floating point registers.

Registers that are dedicated cannot be used for other purposes. The baseptr is currently dedicated but need not be in practice.

3 Calling conventions

Since standard and continuation functions escape, they must have a fixed calling convention. A call to a standard function will have at least six arguments ($arg_1 \cdots arg_n$, $n \geq 6$). The convention is that:

Argument	Register
arg_1	stdlink
arg_2	stdclos
arg_3	stdcont
arg_{4-6}	$\texttt{misc}_{0..2}$
arg_{7-n}	(see below)

arg_{7-n} is selected from the fixed sequence of $\{\texttt{stdarg},\texttt{misc}_{3..M}\}$ or floatregs based on the type of the argument.

Continuation functions have a similar convention but do not use the stdlink or stdclos. The first argument therefore starts with the stdcont register.

Known functions whose calls site are statically known, have their calling convention specialized by the graph coloring register allocator[**?**].

4 Code generation

The prelude to all escaping functions set the exhausted and baseptr registers. Compiling a module results in a runtime code string that may get relocated by the garbage collector. The baseptr holds the start or base of the code string. The runtime address of an escaping function is in the stdlink or stdcont register. If this is at offset L from the start of the code string, then the value of the baseptr is computed from (stdlink − L) or (stdcont − L). Setting the exhausted register is the responsibility of the caller, so the callee in most cases examines the exhausted register and branches to a label to invoke garbage collection. A more complicated test is required if the function allocates more then 4096 bytes — the size of the allocation space skid pad. Note, the allocation space required by an function can be statically computed as all variable-length allocation is done off-line.

The code generated for a standard function on the DEC Alpha would look like:

```
f:
   bne exhausted, gclabel
   addl baseptr, stdlink, k-valueOf(f)
   ...
gclabel:
   <invokeGC>
   jmp stdlink
```

valueOf(f) is the relative offset of the label f from the beginning of the code string being generated, and k is an architecture-specific constant used to bias the value held in the baseptr. Since the stdlink contains the address of the function that called the garbage collector, the return is implemented as an indirect jump through stdlink.

Code generation is done in units of clusters, where two functions are in a cluster if there is a direct call from one to the other. It turns out that a lot of the garbage collection invocation code for escaping functions can be shared, so the code that actually gets generated is:

```
f:
   bne exhausted, gclabel
   addl baseptr, stdlink, k-valueOf(f)
   ...
gclabel:
   b         shared_GC_Label
```

The label shared_GC_Label (to actually invoke garbage collection) is emitted at the end of the compilation unit (after all clusters have been processed). The label `gclabel` is also shared (if possible) by escaping functions within the cluster, and contains a branch instruction to the label `shared_GC_Label`. The indirection at the (per-cluster) `gclabel` is required to avoid the possible long form of a conditional branch:

```
f:
   beq  exhausted, fall
   b    shared_GC_Label
fall:
   ...
```

Known functions cannot share their garbage collection invocation code as the calling convention is determined by the register allocator; implying that formal parameters could wind up in any register, or even be spilled to memory.

5 GC-API

All communication to the runtime system is through a fixed set of registers, namely:

- `stdlink`
- `stdclos`
- `stdcont`
- $\mathtt{misc}_{0-2}$
- `stdarg`

Anything else must be appropriately bundled up, and unbundled on return from the garbage collector. Raw 32 bit values and floating point registers are bundled into a runtime system raw64 array, and everything else is bundled into a record.

In the case of known functions, the formals are in temporary registers, so a parallel copy is emitted to move these temporaries into the fixed registers. For example, if the formals to a known function are, `t100`$\cdots$`t108` then the following code is generated (where `gcRegs` refers to the seven fixed of registers):

```
t109 := {t100, t101, t102}
gcRegs := t103 ... t109
call gc
t103 ... t109 := gcRegs
t100 := t109.0
t101 := t109.1
t102 := t109.2
<set baseptr>
b f
```

The first assignment creates a record assuming everything is boxed, while the second and third assignments are parallel copies. Copy-propagation in the register allocator may be able to eliminate the need for all of these copies. After the call to the garbage collector, the temporaries are reassigned and the record unbundled. Since the `baseptr` is not part of the fixed set of registers it must be either bundled or recomputed; I have chosen to recompute it. Finally, control jumps back to the beginning of the function; re-executimg the garbage collection test this time is really unnecessary.

6 Execution model

Code generated from C and code generated from ML are run as a co-routines. Control from C transfers to ML via a hand written assembly routine (`restoreregs`), which saves the state of the C thread and loads the state for the ML thread into registers. The inverse occurs when going in the opposite direction via an assembly routine – `saveregs`.

The routines `saveregs` and `restoreregs` are very simple as only a fixed set of registers are involved when performing this context switch. In the prior implementation of `restoreregs` all ML registers were restored as any one could have been live.

7 Results

All results are based on compiling the compiler itself using a 533MHz, 21164A Dec Alpha, with 128 MB of RAM.

7.1 Compilation speed

Version 110.15 compiling itself is 4% faster than the previous version (110.14) compiling the source code for 110.15. Version 110.15 compiles itself in about 6.5 minutes. The raw numbers are (note, `usr` includes `gc` time):

Version	gc	sys	usr	usr+sys
110.14	44.01	7.32	389.10	396.42s
110.15	34.47	6.62	374.56	381.18s
			Speedup:	4%

The code size for `ml-yacc` actually shrunk between versions 110.14 and 110.15. I don't have a rational explanation for this.

7.2 GC statistics

It is meaningful to separate the numbers for each of the different types of functions generated by the CPS compiler. There are 34,155 continuation and 10,321 standard functions, and 1,043 known functions (that require a garbage collection test) generated when compiling the compiler.

In the subsequent tables `bb/ii/ff` refers to an encoding for the number of different types of registers used in a call to the garbage collector. `bb` refers to the number of registers that contained tagged objects, `ii` refers to the number of registers that contained raw 32 bit values, and `ff` refers to the number of floating point registers that were live.

Non-zero values of `ii` or `ff` require a raw64 array, and extra registers be bundled into a record when `bb` (+ 1 if either `ii` or `ff` is non-zero) is greater than seven.

All escaping functions that use the same set of registers share a common garbage collection invocation routine, whereas each known function must have its own routine. The tables described in Sections **??** to **??** do not give a sense of the amount of sharing present within a file since they aggregates numbers over all the files that make up the compiler. The amount of sharing within a file is dependent on the file, and Section **??** gives some random samples from some of the larger files in the compiler.

7.2.1 Continuation functions

registers	count	%
5/0/0	32401	94.9
6/0/0	926	2.7
7/0/0	444	1.3
8/0/0	75	0.2
9/0/0	114	0.3
10/0/0	12	0.0
11/0/0	4	0.0
12/0/0	4	0.0
13/0/0	20	0.1
4/1/0	93	0.3
5/1/0	4	0.0
4/0/1	58	0.2

Figure 1: Continuation functions

The first entry in Figure **??** indicates that 95% of all continuation functions take a single argument (the other four registers being accounted for by the `stdcont` and the three calleesave registers). In this particular case, the unused registers must be zeroed out before invoking the garbage collector. 99% of all continuations will not need to build an extra record or raw64 array.

7.2.2 Standard functions

Again from the first entry in Figure **??**, 76% of all standard function take one argument, and all the others would require some sort of bundling of extra registers. One could have separate entry points that shared code to build up the record of extra arguments, however, the number of different types of standard functions within a file is small, and therefore opportunities for reuse would be few.

7.2.3 Known functions

The total number of known functions that require garbage collection checks is much smaller than the number of standard or continuation functions. From Figure **??** (all entries whose contribution was less than 1% have been omitted), 68% will not require extra bundling, and the register allocator may be able to target registers to minimize the size of the code generated. The copies generated to move the formal parameters into the fixed set of registers should be given low priority by the register allocator as they should not influence the allocation in the main rou-

registers	**count**	%
7/0/0	7843	76.0
8/0/0	1553	15.0
9/0/0	381	3.7
10/0/0	128	1.2
11/0/0	121	1.2
12/0/0	25	0.2
13/0/0	11	0.1
14/0/0	9	0.1
15/0/0	4	0.0
6/1/0	89	0.9
6/2/0	76	0.7
7/1/0	9	0.1
6/0/1	47	0.5
6/0/2	22	0.2
6/0/3	3	0.0

Figure 2: Standard functions

registers	**count**	%
1/0/0	215	20.6
2/0/0	20	1.9
3/0/0	31	3.0
4/0/0	36	3.5
5/0/0	80	7.7
6/0/0	215	20.6
7/0/0	110	10.5
8/0/0	86	8.2
9/0/0	68	6.5
10/0/0	44	4.2
11/0/0	32	3.1
12/0/0	20	1.9
13/0/0	12	1.2

Figure 3: Known functions

tines. Assigning priorities to moves based on profiling is not currently supported.

7.3 Per-file sharing

Figure **??** is intended to give an indication of the sharing of garbage collection invocation code on a per-file basis, for two large modules used within the compiler.

`unpickmod.sml`

type	**registers**	**sharing**
Continuation	5/0/0	863
"	7/0/0	2
"	10/0/0	2
Standard	7/0/0	872
"	8/0/0	4
"	10/0/0	1
Known	1/0/0	1

`mlriscGen.sml`

type	**registers**	**sharing**
Continuation	5/0/0	695
"	6/0/0	3
"	4/0/1	1
Standard	7/0/0	25
"	8/0/0	5
	9/0/0	8
Known	1/0/0	1
"	5/0/0	1
"	6/0/0	1
"	7/0/0	1
"	7/0/0	1
	10/0/0	1
"	11/0/0	1
"	12/0/0	1

Figure 4: Per-file sharing

8 Conclusions

The communication of live registers to the garbage collector through a fixed set of registers is beneficial because:

- Most of the time, the code generated to invoke garbage collection can be kept small and no extra bundling is required.
- The runtime system only needs to know about a small set of fixed registers.
- The overhead in the co-routine switch between ML and C is reduced.
- The size of ML state is reduced, which would benefit a multi-threaded system.

References

[App92] A. W. Appel. *Compiling with Continuations*. Cambridge Univ. Press, 1992. ISBN 0-521-41695-7.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM transacations on programming languages and systems.*, 18(3):300–324, May 1996.