Faster CakeML compilation with a verified linear scan register allocator

Théophile Wallez

04/09/2018
Compiler transformations

source syntax

source AST

Languages

Values

Parse concrete syntax

Infer types, exit if fail

Introduce globals vars, eliminate modules & replace constructor names with numbers

Make patterns exhaustive

Global dead code elim.

Turn tuples into constructors

Move nullary constructor patterns upwards

Compile pattern matches to nested Ifs and Lets

Implement bounds checks

Fuse function calls/apps into multi-arg calls/apps

Track where closure values flow; annotate program

Introduce C-style fast calls wherever possible

Remove deadcode

Prepare for closure conv.

Perform closure conv.

Inline small functions

Fold constants and shrink Lets

Split over-sized functions into many small functions

Compile global vars into a dynamically resized array

Optimise Let-expressions

Make some functions tail-recursive using an acc.

Switch to imperative style

Reduce caller-saved vars

Combine adjacent memory allocations

Remove data abstraction

Simplify program

Select target instructions

Perform SSA-like renaming

Force two-reg code (if req.)

Remove deadcode

Allocate register names

Concretise stack

Implement GC primitive

Turn stack access into memory accesses

Rename registers to match arch registers/conventions

Flatten code

Delete no-ops (Tick, Skip)

Encode program as concrete machine code

This internship

All languages communicate with the external world via a byte-array-based foreign-function interface.
Compiler transformations

source syntax
source AST

Languages

Values

Parse concrete syntax
Infer types, exit if fail
Introduce globals vars, eliminate modules & replace constructor names with numbers
Make patterns exhaustive
Global dead code elm.
Turn tuples into constructors
Move nullary constructor patterns upwards
Compile pattern matches to nested Ifs and Lets
Implement bounds checks
Fuse function calls/apps into multi-arg calls/apps
Track where closure values flow; annotate program
Introduce C-style fast calls wherever possible
Remove deadcode
Prepare for closure conv.
Perform closure conv.
Inline small functions
Fold constants and shrink Lets
Split over-sized functions into many small functions
Compile global vars into a dynamically resized array
Optimise Let-expressions
Make some functions tail-recursive using an acc.
Switch to imperative style
Reduce caller-saved vars
Combine adjacent memory allocations
Remove data abstraction

WordLang:
imperative language with machine words, memory and a GC primitive

Simplify program
Select target instructions
Perform SSA-like renaming
Force two-reg code (if req.)
Remove deadcode
Allocate register names

FlatLang:
a language for compiling away high-level lang. features

No pat. match

BVL:
functional language without closures

BVI:
one global variable

DataLang:
imperative language with closures (has multi-arg closures)

ClosLang:
last language with closures

This internship
Compiler transformations

source syntax
source AST

Languages

Values

Parse concrete syntax
Infer types, exit if fail
Introduce globals vars, eliminate modules & replace constructor names with numbers
Make patterns exhaustive
Global dead code elim.

no pat. match

ClosLang: last language with closures
(has multi-arg closures)

Infer types, exit if fail
Introduce globals vars, eliminate modules & replace constructor names with numbers
Make patterns exhaustive
Compile pattern matches to nested Ifs and Lets
Implement bounds checks
Track where closure values flow; annotate program
introduce C-style fast calls wherever possible

FlatLang: a language for compiling away high-level lang. features

BVI: functional language without closures

BVI: one global variable

DataLang: imperative language

Compile global vars into a dynamically resized array
Optimise Let-expressions
Make some functions tail-recursive using an acc.
Switch to imperative style
Reduce caller-saved vars
Combine adjacent memory allocations
Reduce data abstraction

WordLang: imperative language with machine words, memory and a GC primitive

Simplify program
Select target instructions
Perform SSA-like renaming
Force two-reg code (if req.)
Remove deadcode
Allocate register names

StackLang: imperative language with array-like stack and optional GC

Implement GC primitive
Turn stack access into memory accesses
Rename registers to match arch registers/conventions
Flatten code
Delete no-ops (Tick, Skip)
Encode program as concrete machine code

LabLang: assembly lang.

Allocate register names

ARMv6
ARMv8
x86-64
MIPS-64
RISC-V

MIPS-64
This internship

All languages communicate with the external world via a byte-array-based foreign-function interface.
What is register allocation

Model used for optimisations:
infinite number of registers

Reality:
small number of fast registers
infinite number of slow registers
What is register allocation

Model used for optimisations:
infinite number of registers

Reality:
small number of fast registers
infinite number of slow registers
Motivation for a new algorithm

CakeML currently uses the iterated register coalescing algorithm [GA96]
Motivation for a new algorithm

CakeML currently uses the iterated register coalescing algorithm [GA96].
It produces good code quality, but is slow: it is the slowest part of the compiler.
Motivation for a new algorithm

CakeML currently uses the iterated register coalescing algorithm [GA96]. It produces good code quality, but is slow: it is the slowest part of the compiler.

Solution: the linear scan algorithm [PS99]. Orders of magnitude faster, only slightly worse code quality.
Linear scan: liveness analysis

Q: When can we allocate two register to the same color?
Q: When can we allocate two register to the same color?

A: When they never hold a useful value at the same time in the program
Linear scan: liveness analysis

Q: When can we allocate two register to the same color?

A: When they never hold a useful value at the same time in the program

Definition: a register lives at a point of the program iff its value is useful
Linear scan: liveness analysis

Q: When can we allocate two register to the same color?

A1: When they never hold a useful value at the same time in the program

Definition: a register lives at a point of the program iff its value is useful
A2: When they never live at the same time
Linear scan: liveness analysis

<table>
<thead>
<tr>
<th></th>
<th>Live = {}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>/* Live = {}</td>
</tr>
<tr>
<td></td>
<td>a ← ...</td>
</tr>
<tr>
<td>1</td>
<td>/* Live = {a}</td>
</tr>
<tr>
<td></td>
<td>b ← ...</td>
</tr>
<tr>
<td>2</td>
<td>/* Live = {a, b}</td>
</tr>
<tr>
<td></td>
<td>if ... :</td>
</tr>
<tr>
<td>3</td>
<td>/* Live = {b}</td>
</tr>
<tr>
<td></td>
<td>c ← b</td>
</tr>
<tr>
<td>4</td>
<td>/* Live = {c}</td>
</tr>
<tr>
<td></td>
<td>else:</td>
</tr>
<tr>
<td>5</td>
<td>/* Live = {a}</td>
</tr>
<tr>
<td></td>
<td>c ← a</td>
</tr>
<tr>
<td>6</td>
<td>/* Live = {c}</td>
</tr>
<tr>
<td>7</td>
<td>/* Live = {c}</td>
</tr>
<tr>
<td>8</td>
<td>/* Live = {}</td>
</tr>
</tbody>
</table>

Live(a) = \{1, 2, 5\} ⊂ [1, 5]
Live(b) = \{2, 3\} ⊂ [2, 3]
Live(c) = \{4, 6, 7\} ⊂ [4, 7]
Linear scan: liveness analysis

0 /* Live = {} */
a ← ...

1 /* Live = {a} */
b ← ...

2 /* Live = {a, b} */
if ...

3 /* Live = {b} */
c ← b
Live(a) = \{1, 2, 5\} \subset [1, 5]
Live(b) = \{2, 3\} \subset [2, 3]

4 /* Live = {c} */
Live(c) = \{4, 6, 7\} \subset [4, 7]

else:

5 /* Live = {a} */
c ← a

6 /* Live = {c} */

7 /* Live = {c} */
print(c)

8 /* Live = {} */
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Linear scan: register allocation

Color pool

Active list
Setup of the current register allocator

clash_tree =
    Delta (num list) (num list)
| Set num_set
| Branch (num_set option) clash_tree clash_tree
| Seq clash_tree clash_tree

get_live_backward_ct (Delta writes reads)
live =
    live \ reads

get_live_backward_ct (Set cutset)
live = cutset

get_live_backward_ct (Branch (Some cutset) ct1 ct2)
live =
    get_live_backward_ct ct1 live
    \n    get_live_backward_ct ct2 live

get_live_backward_ct (Seq ct1 ct2)
live =
    get_live_backward_ct ct1 (get_live_backward_ct ct2 live)
Setup of the current register allocator

\[
\text{clash\_tree} = \\
\quad \text{Delta (num list) (num list)} \\
\mid \text{Set num\_set} \\
\mid \text{Branch (num\_set option) clash\_tree clash\_tree} \\
\mid \text{Seq clash\_tree clash\_tree}
\]

\[
\text{get\_live\_backward\_ct (Delta writes reads) live} = \\
\quad (\text{live} \setminus \text{writes}) \cup \text{reads}
\]

\[
\text{get\_live\_backward\_ct (Set cutset) live} = \text{cutset}
\]

\[
\text{get\_live\_backward\_ct (Branch (Some cutset) ct_1 ct_2) live} = \text{cutset}
\]

\[
\text{get\_live\_backward\_ct (Branch None ct_1 ct_2) live} = \\
\quad (\text{get\_live\_backward\_ct ct_1 live}) \cup (\text{get\_live\_backward\_ct ct_2 live})
\]

\[
\text{get\_live\_backward\_ct (Seq ct_1 ct_2) live} = \\
\quad \text{get\_live\_backward\_ct ct_1 (get\_live\_backward\_ct ct_2 live)}
\]
Setup of the current register allocator

```
clash_tree =
    Delta (num list) (num list)
| Set num_set
| Branch (num_set option) clash_tree clash_tree
| Seq clash_tree clash_tree

get_live_backward_ct (Delta writes reads) live =
    (live \ writes) ∪ reads
get_live_backward_ct (Set cutset) live = cutset
get_live_backward_ct (Branch (Some cutset) ct₁ ct₂) live = cutset
get_live_backward_ct (Branch None ct₁ ct₂) live =
    (get_live_backward_ct ct₁ live) ∪ (get_live_backward_ct ct₂ live)
get_live_backward_ct (Seq ct₁ ct₂) live =
    get_live_backward_ct ct₁ (get_live_backward_ct ct₂ live)
```

check_clash_tree col clashtree
Meet the live_tree datatype

live_tree =
    | Writes (num list)  | Transformation done by get_live_tree |
    | Reads (num list)   |
    | Branch live_tree live_tree |
    | Seq live_tree live_tree |
Meet the live_tree datatype

live_tree =
    | Writes (num list)  
    | Reads (num list)  
    | Branch live_tree live_tree  
    | Seq live_tree live_tree

Transformation done by get_live_tree

get_live_backward (Writes wr) live =
    live \ wr
get_live_backward (Reads rd) live =
    live \cup rd
get_live_backward (Branch ct_1 ct_2) live =
    (get_live_backward ct_1 live) \cup (get_live_backward ct_2 live)
get_live_backward (Seq ct_1 ct_2) live =
    get_live_backward ct_1 (get_live_backward ct_2 live)
Meet the live_tree datatype

live_tree =
    Writes (num list)               Transformation done by
    | Reads (num list)               get_live_tree
    | Branch live_tree live_tree
    | Seq live_tree live_tree

get_live_backward (Writes wr) live =
    live \ wr
get_live_backward (Reads rd) live =
    live ∪ rd
get_live_backward (Branch ct₁ ct₂) live =
    (get_live_backward ct₁ live) ∪ (get_live_backward ct₂ live)
get_live_backward (Seq ct₁ ct₂) live =
    get_live_backward ct₁ (get_live_backward ct₂ live)

check_live_tree col livetree
Theorem
check_live_tree col (get_live_tree clashtree) ⇒
check_clash_tree col clashtree
Correctness theorem of get_live_tree

Theorem
check_live_tree col (get_live_tree clashtree) ⇒
check_clash_tree col clashtree

Proof.
By induction on clashtree, and using the lemmas:

get_live_backward_ct clashtree live ⊆
get_live_backward (get_live_tree clashtree) live

and

live₁ ⊆ live₂ ⇒
get_live_backward clashtree live₁ ⊆ get_live_backward clashtree live₂
Liveness intervals: naive algorithm

Naive algorithm: compute living sets at each position of the program, then compute the intervals.
Liveness intervals: naive algorithm

Naive algorithm: compute living sets at each position of the program, then compute the intervals.

Problem: it might be Ω(\(n^2\))
Liveness intervals: a faster algorithm

Insight: liveness interval start at a Writes and ends at a Reads
Liveness intervals: a faster algorithm

Insight: liveness interval start at a Writes and ends at a Reads

Fast algorithm:

- Beginning of interval of \textit{reg} is the first line where \textit{reg} is written to
- End of interval of \textit{reg} is the last line where \textit{reg} is read
Liveness intervals: a problem?

1 read \((a)\)

\[
\text{Live}(a) = [?, 1]
\]
Liveness intervals: a problem?

1. read (a)
2. write (a)
3. read (a)

Live(a) = [2, 3]
Liveness intervals: a problem?

```plaintext
if ...:
  1 | ...
else:
  2 | write (a)
  3 | read (a)

Live(a) = [2, 3]
```
Liveness intervals: a property on programs

Every read must be dominated by a write
Liveness intervals: a property on programs

Every read must be dominated by a write

Equivalently,
\[ \text{get\_live\_backward\ livetree}\ (\emptyset) = \emptyset \]
Liveness intervals: a property on programs

Every read must be dominated by a write

Equivalently,
\[
\text{get\_live\_backward livetree } \emptyset = \emptyset
\]

Not easy to prove. A brutal solution is:

\[
\begin{align*}
\text{fix\_domination } lt &= \\
&= \begin{cases} \\
\text{let live } &= \text{get\_live\_backward } lt \emptyset \text{ in} \\
\text{if live } &= \emptyset \text{ then } lt \\
\text{else Seq (Writes (list\_to\_numset live)) } lt \\
\end{cases}
\end{align*}
\]
Liveness intervals: a property on programs

Every read must be dominated by a write

Equivalently,
get_live_backward livetree ∅ = ∅

Not easy to prove. A brutal solution is:

fix_domination lt =
let live = get_live_backward lt ∅ in
if live = ∅ then lt
else Seq (Writes (list_to_numset live)) lt

(* TODO: might be Ω(n^2) *)
Liveness intervals: proof of correctness

A problem?

1 write(a)
2 write(c)
   if ... :
3     write(b)
4     write(c)  \quad \text{Live(a) = [?, ?]}
\text{Live(b) = [?, ?]}
\text{Live(c) = [?, ?]}
4     write(c)
   else:
5     write(a)  \quad \text{Live(a) = [?, ?]}
\text{Live(b) = [?, ?]}
\text{Live(c) = [?, ?]}
6     write(b)
7 read(a)
8 read(b)
9 read(c)
10 write(b)
Liveness intervals: proof of correctness

A problem?

1  write(a)
2  write(c)
   if ... :
3   |  write(b)
4   |  write(c)
   else:
5   |  write(a)
6   |  write(b)
7  read(a)
8  read(b)
9  read(c)
10 write(b)

Live(a) = [?, ?]
Live(b) = [10, 10]
Live(c) = [?, ?]
Liveness intervals: proof of correctness

A problem?

```
1  write(a)
2  write(c)
   if ... :
3       write(b)
4       write(c)
   else:
5       write(a)
6       write(b)
7  read(a)
8  read(b)
9  read(c)
10 write(b)
```

Live(a) = [?, ?]
Live(b) = [10, 10]
Live(c) = [?, 9]
Liveness intervals: proof of correctness

A problem?

```plaintext
1 write(a)  
2 write(c)  
   if ... :  
3     write(b)  
4     write(c)  
   else:  
5     write(a)  
6     write(b)  
7 read(a)  
8 read(b)  
9 read(c)  
10 write(b)  
```

Live(a) = [?, ?]  
Live(b) = [10, 10]  
Live(c) = [?, 9]
Liveness intervals: proof of correctness

A problem?

```plaintext
1  write(a)
2  write(c)
   if ... :
3   |  write(b)
4   |  write(c)
   else:
5   |  write(a)
6   |  write(b)
7  read(a)
8  read(b)
9  read(c)
10 write(b)

Live(a) = [?, 7]
Live(b) = [10, 10]
Live(c) = [?, 9]
```
Liveness intervals: proof of correctness

A problem?

```
1 write(a)
2 write(c)
    if ... :
3     write(b)
4     write(c)
    else:
5     write(a)
6     write(b)
7 read(a)
8 read(b)
9 read(c)
10 write(b)
```

Live(a) = [?, 7]
Live(b) = [6, 10]
Live(c) = [?, 9]
A problem?

```
1 write(a)
2 write(c)
   if ...:
3     write(b)
4     write(c)
else:
5     write(a)
6     write(b)
7 read(a)
8 read(b)
9 read(c)
10 write(b)
```

Live(a) = [5, 7]
Live(b) = [6, 10]
Live(c) = [?, 9]
Liveness intervals: proof of correctness

A problem?

```plaintext
1 write(a)
2 write(c)
   if ... :
3      write(b)
4      write(c)
   else:
5      write(a)
6      write(b)
7 read(a)
8 read(b)
9 read(c)
10 write(b)

Live(a) = [5, 7]
Live(b) = [6, 10]
Live(c) = [4, 9]
```
Liveness intervals: proof of correctness

A problem?

```
1 write(a)
2 write(c)
   if ...
   3    write(b)
3      write(c)
4      write(c)
   else:
5    write(a)
5      write(b)
6      write(b)
7 read(a)
8 read(b)
9 read(c)
10 write(b)
```

Live(a) = [5, 7]
Live(b) = [3, 10]
Live(c) = [4, 9]
Liveness intervals: proof of correctness

A problem?

1 write(a)
2 write(c)
   if ... :
3       write(b)
4       write(c)
else:
5       write(a)
6       write(b)
7 read(a)
8 read(b)
9 read(c)
10 write(b)

Live(a) = [5, 7]
Live(b) = [3, 10]
Live(c) = [2, 9]
Liveness intervals: proof of correctness

A problem?

```
  1 write(a)
  2 write(c)
      if ... :
  3      | write(b)
  4      | write(c)
   else:
  5      | write(a)
  6      | write(b)
  7 read(a)
  8 read(b)
  9 read(c)
 10 write(b)
```

Live(a) = [1, 7]
Live(b) = [3, 10]
Live(c) = [2, 9]
Problem: what we want to prove is not true locally
Liveness intervals: proof of correctness

A problem?

Problem: what we want to prove is not true locally

Solution: Force the following property at every step:
If a is live, then beg[a] = ?
Liveness intervals: proof of correctness

A modified algorithm

```plaintext
1 write(a)
2 write(c)
   if ...:
3     write(b)
4     write(c)  Live(a) = [?, ?]
5     write(c)  Live(b) = [?, ?]
6     write(b)  Live(c) = [?, ?]
7 read(a)     ? = 11
8 read(b)
9 read(c)
10 write(b)
```
Liveness intervals: proof of correctness

A modified algorithm

1 write(a)
2 write(c)
   if ... :
3     write(b)
4     write(c)
   else:
5     write(a)
6     write(b)
7 read(a)
8 read(b)
9 read(c)
10 write(b)

Live(a) = [?, ?]
Live(b) = [10, 10]
Live(c) = [?, ?]
? = 10
Liveness intervals: proof of correctness

A modified algorithm

```
1 write(a)
2 write(c)
3     if ... :
4         write(b)  Live(a) = [?, ?]
5         write(c)  Live(b) = [10, 10]
6     else:  Live(c) = [?, 9]
7         write(a)  ? = 9
8         write(b)
9 read(c)
10 write(b)
```
Liveness intervals: proof of correctness

A modified algorithm

```
1 write(a)
2 write(c)
   if ... :
3     write(b)
4     write(c)
   else:
5     write(a)
6     write(b)
7 read(a)
8 read(b)
9 read(c)
10 write(b)
```

Live(a) = [?, ?]
Live(b) = [?, 10]
Live(c) = [?, 9]
?

? = 8
Liveness intervals: proof of correctness

A modified algorithm

    1 write(a)
    2 write(c)
      if ... :
      3     write(b)
      4     write(c)           Live(a) = [?, 7]
      5     write(a)           Live(b) = [?, 10]
      6     write(b)           Live(c) = [?, 9]
    ? = 7
    7 read(a)
    8 read(b)
    9 read(c)
   10 write(b)
Liveness intervals: proof of correctness

A modified algorithm

1 write(a)
2 write(c)
   if ... :
3       write(b)
4       write(c)
else:
5       write(a)
6       write(b)

7 read(a)
8 read(b)
9 read(c)
10 write(b)

Live(a) = [?, 7]
Live(b) = [6, 10]
Live(c) = [?, 9]

? = 6
Liveness intervals: proof of correctness

A modified algorithm

1 write(a)
2 write(c)
   if ... :
3     write(b)
4     write(c)       Live(a) = [5, 7]
5     write(a)       Live(b) = [6, 10]
6     write(b)       Live(c) = [?, 9]
7 read(a)
8 read(b)
9 read(c)
10 write(b)

? = 5
Liveness intervals: proof of correctness

A modified algorithm

```
1 write(a)
2 write(c)
   if ... :
3     write(b)
4     write(c)  Live(a) = [?, 7]
5     else:      Live(b) = [?, 10]
6     write(a)  Live(c) = [?, 9]
7     write(b)  ? = 5
8 read(a)
9 read(b)
10 read(c)
```
Liveness intervals: proof of correctness

A modified algorithm

```plaintext
1 write(a)
2 write(c)
   if ...:
3       write(b)
4       write(c)  \ Live(a) = [?, 7]
5       write(b)  \ Live(b) = [?, 10]
6       write(b)  \ Live(c) = [4, 9]
7 read(a)  \ ? = 4
8 read(b)
9 read(c)
10 write(b)
```
A modified algorithm

```plaintext
1 write(a)
2 write(c)
   if ... :
3     write(b)
4     write(c)  \ Live(a) = [?, 7]
5   else:
6     write(a)  \ Live(b) = [3, 10]
7     write(b)  \ Live(c) = [4, 9]
8 read(a)  \ ? = 3
9 read(b)
10 read(c)
11 write(b)
```
Liveness intervals: proof of correctness

A modified algorithm

1. write(a)
2. write(c)
   if ...:
3.    write(b)
4.    write(c)
5.  else:
6.    write(a)
7.    write(b)
8. read(a)
9. read(b)
10. read(c)

Live(a) = [?, 7]
Live(b) = [3, 10]
Live(c) = [?, 9]

? = 3
Liveness intervals: proof of correctness

A modified algorithm

```
1 write(a)
2 write(c)
3 if ...:
4    write(b)
5    write(c)
6 else:
7      write(a)
8      write(b)
9 read(a)
10 read(b)
11 read(c)
12 write(b)
```

Live(a) = [?, 7]
Live(b) = [3, 10]
Live(c) = [2, 9]
? = 2
Liveness intervals: proof of correctness

A modified algorithm

```
1 write(a)
2 write(c)
   if ... :
3    write(b)
4    write(c)  Live(a) = [1, 7]
5     write(a)  Live(b) = [3, 10]
6     write(b)  Live(c) = [2, 9]
7    read(a)
8    read(b)
9    read(c)
10   write(b)
```
Liveness intervals: proof of correctness

Prove that the two algorithm compute the same thing

Problem: The modified algorithm is easy to prove correct, but is slow
Liveness intervals: proof of correctness
Prove that the two algorithm compute the same thing

Problem: The modified algorithm is easy to prove correct, but is slow
Solution: Prove that the original and the modified algorithm compute the same thing
Liveness intervals: proof of correctness

Prove that the two algorithms compute the same thing

Problem: The modified algorithm is easy to prove correct, but is slow
Solution: Prove that the original and the modified algorithm compute the same thing

Theorem
(begmod[r] \neq ? \land \text{beg[r] } \neq ?) \implies \text{beg[r] } = \text{begmod[r]}

Liveness intervals: proof of correctness

Prove that the two algorithms compute the same thing

Problem: The modified algorithm is easy to prove correct, but is slow
Solution: Prove that the original and the modified algorithm compute the same thing

**Theorem**
\[(\text{begmod}[r] \neq ? \text{ and beg}[r] \neq ?) \implies \text{beg}[r] = \text{begmod}[r]\]

**Theorem**
\[\text{begmod}[r] \neq ? \implies \text{beg}[r] \neq ?\]
Liveness intervals: proof of correctness

Prove that the two algorithms compute the same thing

Problem: The modified algorithm is easy to prove correct, but is slow
Solution: Prove that the original and the modified algorithm compute the same thing

Theorem
\((\text{begmod}[r] \neq ? \text{ and } \text{beg}[r] \neq ?) \Rightarrow \text{beg}[r] = \text{begmod}[r]\)

Theorem
\(\text{begmod}[r] \neq ? \Rightarrow \text{beg}[r] \neq ?\)

Theorem
\(\text{beg}[r] \neq ? \Rightarrow \text{end}[r] \neq ?\)
Liveness intervals: proof of correctness

Prove that the two algorithm compute the same thing

Problem: The modified algorithm is easy to prove correct, but is slow
Solution: Prove that the original and the modified algorithm compute the same thing

Theorem
\((\text{begmod}[r] \neq ? \text{ and beg}[r] \neq ?) \Rightarrow \text{beg}[r] = \text{begmod}[r]\)

Theorem
\(\text{begmod}[r] \neq ? \Rightarrow \text{beg}[r] \neq ?\)

Theorem
\(\text{beg}[r] \neq ? \Rightarrow \text{end}[r] \neq ?\)

Theorem
\(\text{end}[r] \neq ? \Rightarrow (\text{begmod}[r] \neq ? \text{ or } r \text{ is live})\)
Additional requirements for CakeML’s register allocator

- Some type of register must be allocated on the stack
Additional requirements for CakeML’s register allocator

- Some type of register must be allocated on the stack
  Simply spill them automatically
Additional requirements for CakeML’s register allocator

- Some type of register must be allocated on the stack
  Simply spill them automatically
- Stack frame size should be minimized
Additional requirements for CakeML’s register allocator

- Some type of register must be allocated on the stack
  Simply spill them automatically
  - Stack frame size should be minimized
    Do a second pass to reallocate registers on the stack
Additional requirements for CakeML’s register allocator

- Some type of register must be allocated on the stack
  - Simply spill them automatically
- Stack frame size should be minimized
  - Do a second pass to reallocate registers on the stack
- Some pair of registers should have the same color (if possible)
Additional requirements for CakeML’s register allocator

- Some type of register must be allocated on the stack
  Simply spill them automatically
- Stack frame size should be minimized
  Do a second pass to reallocate registers on the stack
- Some pair of registers should have the same color (if possible)
  Check these colors first in the colorpool when allocating the second register
Additional requirements for CakeML’s register allocator

- Some type of register must be allocated on the stack
  Simply spill them automatically
  - Stack frame size should be minimized
  Do a second pass to reallocate registers on the stack
  - Some pair of registers should have the same color (if possible)
  Check these colors first in the colorpool when allocating the second register
  - Some pair of registers must not have the same color
Additional requirements for CakeML’s register allocator

- Some type of register must be allocated on the stack
  Simply spill them automatically
  - Stack frame size should be minimized
  Do a second pass to reallocate registers on the stack
  - Some pair of registers should have the same color (if possible)
  Check these colors first in the colorpool when allocating the second register
  - Some pair of registers must not have the same color
  Remove these colors from the colorpool when allocating the second register
Additional requirements for CakeML’s register allocator

- Some register must be allocated to a specific color
Additional requirements for CakeML’s register allocator

- Some register must be allocated to a specific color
  
  The obvious solution produces bad allocation
Additional requirements for CakeML’s register allocator

- Some register must be allocated to a specific color

The obvious solution produces bad allocation
Additional requirements for CakeML’s register allocator

- Some register must be allocated to a specific color

The obvious solution produces bad allocation
Additional requirements for CakeML’s register allocator

- Some register must be allocated to a specific color

The obvious solution produces bad allocation

Good solution: only ensure they have different colors, find an exchange afterwards
Additional requirements for CakeML’s register allocator

- Some register must be allocated to a specific color

The obvious solution produces bad allocation

Good solution: only ensure they have different colors, find an exchange afterwards
Additional requirements for CakeML’s register allocator

- Some register must be allocated to a specific color

The obvious solution produces bad allocation

Good solution: only ensure they have different colors, find an exchange afterwards
Correctness proof for the linear scan algorithm

- Algorithm split in 16 elementary function
- 20 invariants preserved during the execution
Correctness proof for the linear scan algorithm

- Algorithm split in 16 elementary functions
- 20 invariants preserved during the execution

Each correctness theorem is of the form:

if

- [some condition on the input]
- invariants are verified before calling the function

then

- the functions succeeds (i.e. no array out-of-bounds)
- [some property on the output]
- invariants are verified after calling the function
- [specify which colors might have changed]
Not that bad, but we would hope better.
Performance: compilation time

Not that bad, but we would hope better.
This is really bad. The culprit: physical registers have absurdly long liveness intervals. Solution: place the allocator before calling conventions are enforced.
This is really bad.
This is really bad.
The culprit: physical registers have absurdly long liveness intervals.
This is really bad.
The culprit: physical registers have absurdly long liveness intervals
Solution: place the allocator before calling conventions are enforced
Conclusion

I implemented and verified end-to-end a new register allocator, which might become the default allocator in CakeML.

There is still some work to do to make it useful.
References

Lal George and Andrew W. Appel.
Iterated register coalescing.

Massimiliano Poletto and Vivek Sarkar.
Linear scan register allocation.