Faster CakeML compilation with a verified linear scan register allocator

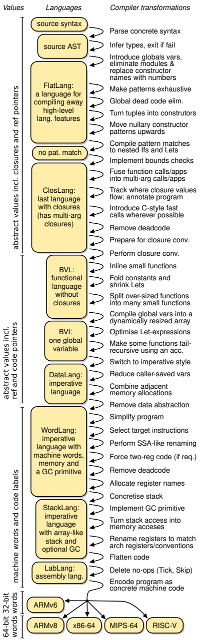
Théophile Wallez

04/09/2018

ÉCOLE NORMALE SUPÉRIEURE

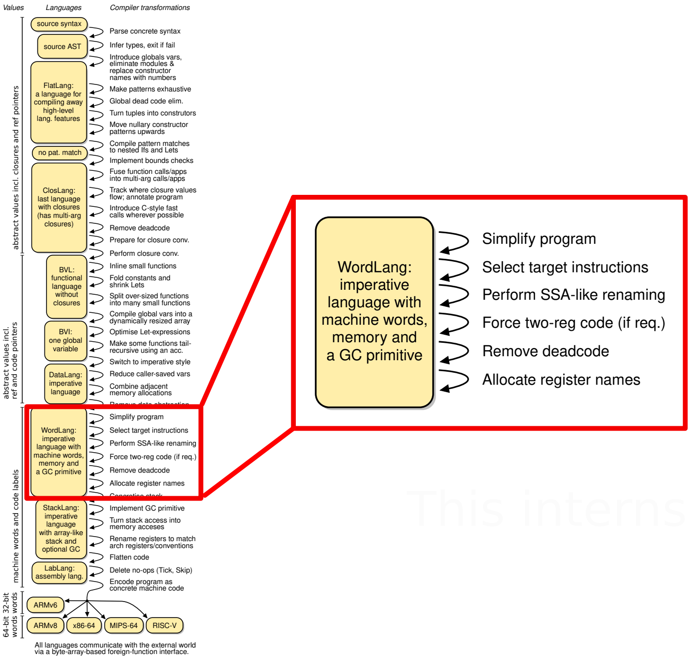
CAKEML
A Verified Implementation of ML

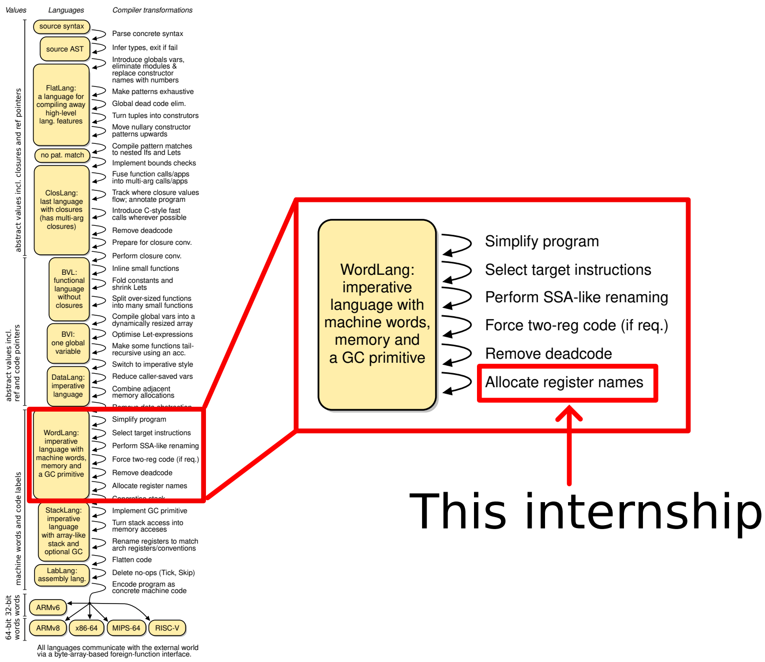
CHALMERS
UNIVERSITY OF TECHNOLOGY



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

This internship


Values
Languages
Compiler transformations
abstract values incl. closures and ref pointers
abstract values incl. ref and code pointers
machine words and code labels
32-bit words
64-bit words
source syntax
Parse concrete syntax
source AST
Infer types, exit if fail
Introduce globals vars, eliminate modules & replace constructor names with numbers
FlatLang: a language for compiling away high-level lang. features
Make patterns exhaustive
Global dead code elim.
Turn tuples into construtors
Move nullary constructor patterns upwards
Compile pattern matches to nested Ifs and Lets
no pat. match
Implement bounds checks
Fuse function calls/apps into multi-arg calls/apps
ClosLang: last language with closures (has multi-arg closures)
Track where closure values flow; annotate program
Introduce C-style fast calls wherever possible
Remove deadcode
Prepare for closure conv.
Perform closure conv.
BVL: functional language without closures
Inline small functions
Fold constants and shrink Lets
Split over-sized functions into many small functions
Compile global vars into a dynamically resized array
BVI: one global variable
Optimise Let-expressions
Make some functions tail-recursive using an acc.
Switch to imperative style
DataLang: imperative language
Reduce caller-saved vars
Combine adjacent memory allocations
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 acceses
Rename registers to match arch registers/conventions
Flatten code
LabLang: assembly lang.
Delete no-ops (Tick, Skip)
Encode program as concrete machine code
ARMv6
ARMv8
x86-64
MIPS-64
RISC-V
All languages communicate with the external world via a byte-array-based foreign-function interface.
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



Values
Languages
Compiler transformations
abstract values incl. closures and ref pointers
abstract values incl. ref and code pointers
machine words and code labels
32-bit words
64-bit words
source syntax
Parse concrete syntax
source AST
Infer types, exit if fail
Introduce globals vars, eliminate modules & replace constructor names with numbers
FlatLang: a language for compiling away high-level lang. features
Make patterns exhaustive
Global dead code elim.
Turn tuples into construtors
Move nullary constructor patterns upwards
Compile pattern matches to nested Ifs and Lets
no pat. match
Implement bounds checks
ClosLang: last language with closures (has multi-arg closures)
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.
BVL: functional language without closures
Inline small functions
Fold constants and shrink Lets
Split over-sized functions into many small functions
Compile global vars into a dynamically resized array
BVI: one global variable
Optimise Let-expressions
Make some functions tail-recursive using an acc.
Switch to imperative style
DataLang: imperative language
Reduce caller-saved vars
Combine adjacent memory allocations
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 acceses
Rename registers to match arch registers/conventions
Flatten code
LabLang: assembly lang.
Delete no-ops (Tick, Skip)
Encode program as concrete machine code
ARMv6
ARMv8
x86-64
MIPS-64
RISC-V
All languages communicate with the external world via a byte-array-based foreign-function interface.
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
This internship

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

$\xrightarrow{\text{Register allocation}}$

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?

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

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

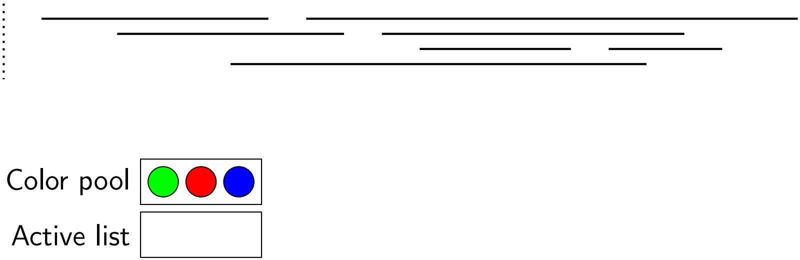
```
0 /* Live = {}          */
  a ← ...
1 /* Live = {a}         */
  b ← ...
2 /* Live = {a, b}      */
  if ... :
3 |  /* Live = {b}      */
  |  c ← b
4 |  /* Live = {c}      */
  else:
5 |  /* Live = {a}      */
  |  c ← a
6 |  /* Live = {c}      */
7 /* Live = {c}         */
  print(c)
8 /* Live = {}          */
```

Linear scan: liveness analysis

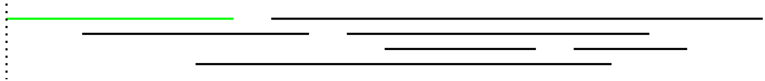
```
0 /* Live = {}          */
  a ← ...
1 /* Live = {a}         */
  b ← ...
2 /* Live = {a, b}      */
  if ... :
3 |  /* Live = {b}      */
  |  c ← b
4 |  /* Live = {c}      */
  else:
5 |  /* Live = {a}      */
  |  c ← a
6 |  /* Live = {c}      */
7 /* Live = {c}         */
  print(c)
8 /* Live = {}          */
```

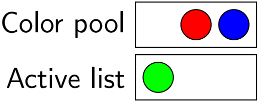
$\text{Live}(a) = \{1, 2, 5\} \subset [1, 5]$
$\text{Live}(b) = \{2, 3\} \subset [2, 3]$
$\text{Live}(c) = \{4, 6, 7\} \subset [4, 7]$

Linear scan: register allocation

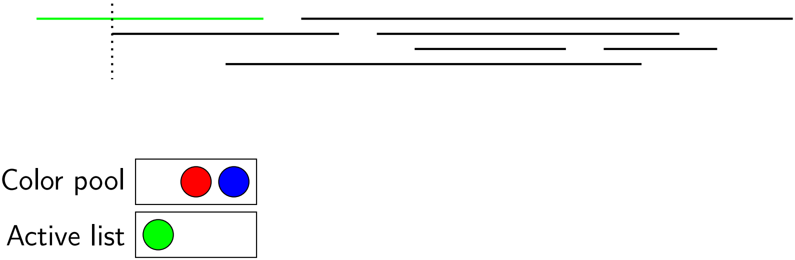


Linear scan: register allocation

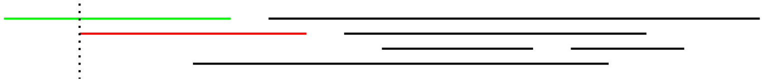


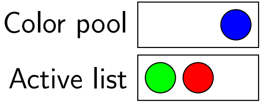


Linear scan: register allocation

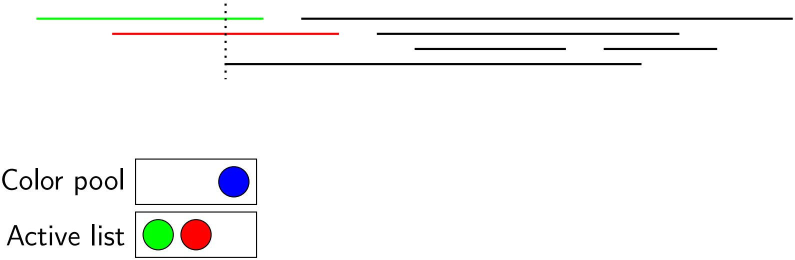


Linear scan: register allocation

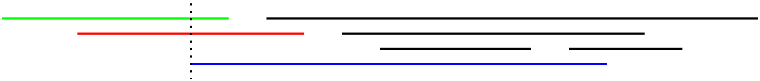


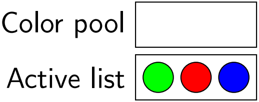


Linear scan: register allocation

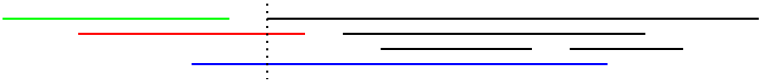


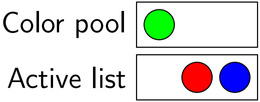
Linear scan: register allocation



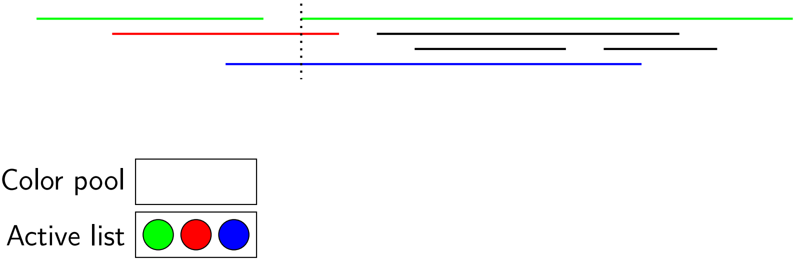


Linear scan: register allocation

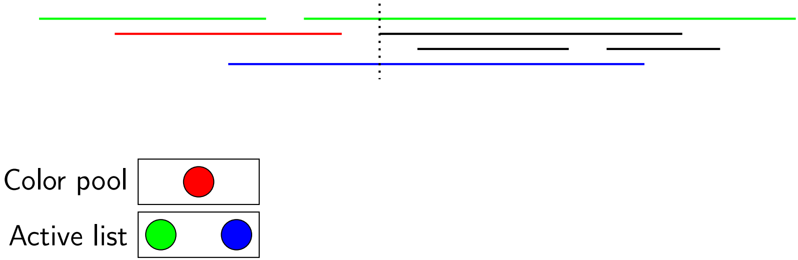




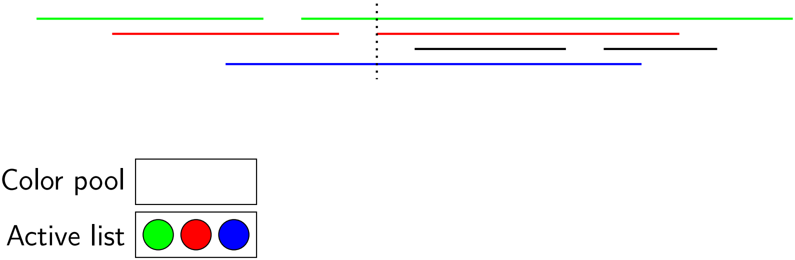
Linear scan: register allocation



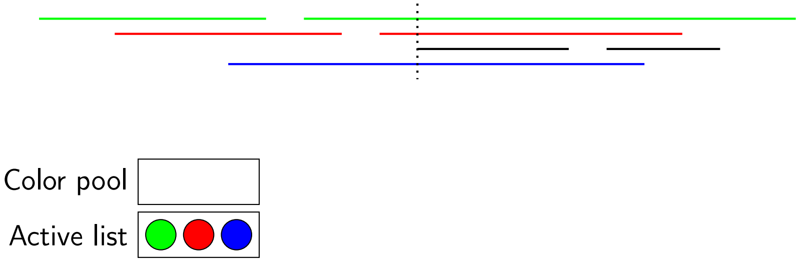
Linear scan: register allocation



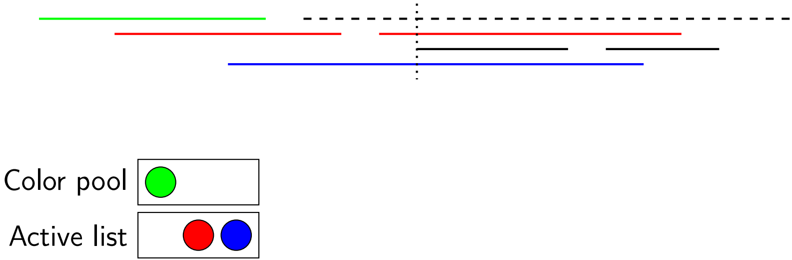
Linear scan: register allocation



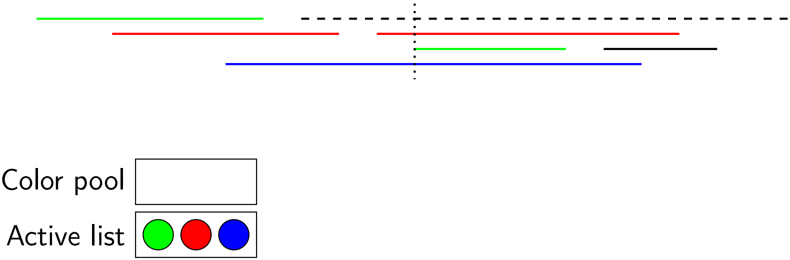
Linear scan: register allocation



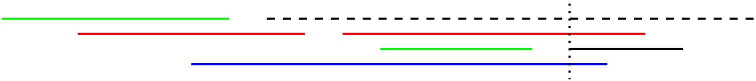
Linear scan: register allocation

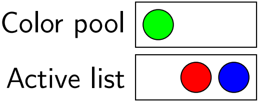


Linear scan: register allocation

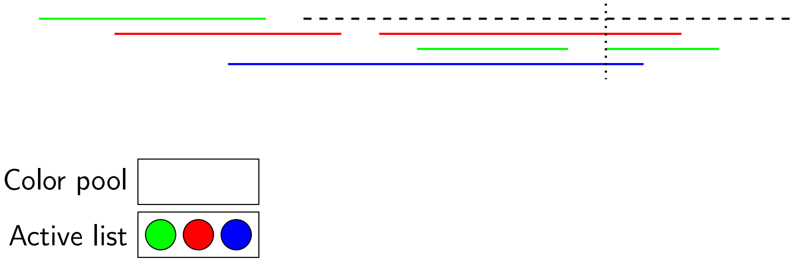


Linear scan: register allocation





Linear scan: register allocation



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
```

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*) ct_1 ct_2) *live* = *cutset*
get_live_backward_ct (Branch None ct_1 ct_2) *live* =
  (get_live_backward_ct ct_1 *live*) ∪ (get_live_backward_ct ct_2 *live*)
get_live_backward_ct (Seq ct_1 ct_2) *live* =
  get_live_backward_ct ct_1 (get_live_backward_ct ct_2 *live*)

Setup of the current register allocator

$$
\begin{array}{l}
\text{clash\_tree} \ = \\
\quad \text{Delta (num list) (num list)} \\
\ \ | \ \text{Set num\_set} \\
\ \ | \ \text{Branch (num\_set option) clash\_tree clash\_tree} \\
\ \ | \ \text{Seq clash\_tree clash\_tree}
\end{array}
$$

$$
\begin{array}{l}
\text{get\_live\_backward\_ct (Delta } \mathit{writes}\ \mathit{reads}\text{) } \mathit{live} = \\
\quad (\mathit{live} \setminus \mathit{writes}) \cup \mathit{reads} \\
\text{get\_live\_backward\_ct (Set } \mathit{cutset}\text{) } \mathit{live} = \mathit{cutset} \\
\text{get\_live\_backward\_ct (Branch (Some } \mathit{cutset}\text{) } \mathit{ct}_1\ \mathit{ct}_2\text{) } \mathit{live} = \mathit{cutset} \\
\text{get\_live\_backward\_ct (Branch None } \mathit{ct}_1\ \mathit{ct}_2\text{) } \mathit{live} = \\
\quad (\text{get\_live\_backward\_ct } \mathit{ct}_1\ \mathit{live}) \cup (\text{get\_live\_backward\_ct } \mathit{ct}_2\ \mathit{live}) \\
\text{get\_live\_backward\_ct (Seq } \mathit{ct}_1\ \mathit{ct}_2\text{) } \mathit{live} = \\
\quad \text{get\_live\_backward\_ct } \mathit{ct}_1\ (\text{get\_live\_backward\_ct } \mathit{ct}_2\ \mathit{live})
\end{array}
$$

check_clash_tree *col clashtree*

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

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 $\setminus$ *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)
  | 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 $\setminus$ *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*)

check_live_tree *col* *livetree*

Correctness theorem of get_live_tree

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*) $\Rightarrow$
check_clash_tree *col* *clashtree*

Proof.

By induction on *clashtree*, and using the lemmas:

get_live_backward_ct *clashtree* *live* $\subseteq$
get_live_backward (get_live_tree *clashtree*) *live*

and

$live_1 \subseteq live_2 \Rightarrow$
get_live_backward *clashtree* $live_1 \subseteq$ get_live_backward *clashtree* $live_2$

□

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 $\Omega(n^2)$

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 *reg* is the first line where *reg* is written to
- End of interval of *reg* is the last line where *reg* is read

Liveness intervals: a problem?

1	`read (a)`	$\mathsf{Live}(a) = [?, 1]$

Liveness intervals: a problem?

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

$\text{Live}(a) = [2, 3]$

Liveness intervals: a problem?

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

$\text{Live}(a) = [2, 3]$

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

Equivalentely,
get_live_backward *livetree* $\emptyset = \emptyset$

Liveness intervals: a property on programs

Every read must be dominated by a write

Equivalentely,
get_live_backward *livetree* $\emptyset = \emptyset$

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
```

Liveness intervals: a property on programs

Every read must be dominated by a write

Equivalentely,
get_live_backward *livetree* $\emptyset = \emptyset$

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 $\Omega(n^2)$ *)

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)
```

$\text{Live}(a) = [?, ?]$
$\text{Live}(b) = [?, ?]$
$\text{Live}(c) = [?, ?]$

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)
```

$\text{Live}(a) = [?, ?]$
$\text{Live}(b) = [10, 10]$
$\text{Live}(c) = [?, ?]$

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)
```

$\text{Live}(a) = [?, ?]$
$\text{Live}(b) = [10, 10]$
$\text{Live}(c) = [?, 9]$

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)
```

$\text{Live}(a) = [?, ?]$
$\text{Live}(b) = [10, 10]$
$\text{Live}(c) = [?, 9]$

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)
```

$\text{Live}(a) = [?, 7]$
$\text{Live}(b) = [10, 10]$
$\text{Live}(c) = [?, 9]$

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)
```

$\text{Live}(a) = [?, 7]$
$\text{Live}(b) = [6, 10]$
$\text{Live}(c) = [?, 9]$

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)
```

$\text{Live}(a) = [5, 7]$
$\text{Live}(b) = [6, 10]$
$\text{Live}(c) = [?, 9]$

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)
```

$\text{Live}(a) = [5, 7]$
$\text{Live}(b) = [6, 10]$
$\text{Live}(c) = [4, 9]$

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)
```

$\text{Live}(a) = [5, 7]$
$\text{Live}(b) = [3, 10]$
$\text{Live}(c) = [4, 9]$

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)
```

$\text{Live}(a) = [5, 7]$
$\text{Live}(b) = [3, 10]$
$\text{Live}(c) = [2, 9]$

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)
```

$\text{Live}(a) = [1, 7]$
$\text{Live}(b) = [3, 10]$
$\text{Live}(c) = [2, 9]$

Liveness intervals: proof of correctness

A problem?

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

```
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) = [?, ?]
Live(c) = [?, ?]
? = 11

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)
```

$\text{Live}(a) = [?, ?]$
$\text{Live}(b) = [10, 10]$
$\text{Live}(c) = [?, ?]$
$? = 10$

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)
```

$\text{Live}(a) = [?, ?]$
$\text{Live}(b) = [10, 10]$
$\text{Live}(c) = [?, 9]$
$? = 9$

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)
```

$\text{Live}(a) = [?, ?]$
$\text{Live}(b) = [?, 10]$
$\text{Live}(c) = [?, 9]$
$? = 8$

Liveness intervals: proof of correctness

A modified algorithm

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

$\text{Live}(a) = [?, 7]$
$\text{Live}(b) = [?, 10]$
$\text{Live}(c) = [?, 9]$
$? = 7$

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)
```

$\text{Live}(a) = [?, 7]$

$\text{Live}(b) = [6, 10]$

$\text{Live}(c) = [?, 9]$

$? = 6$

Liveness intervals: proof of correctness

A modified algorithm

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

$\text{Live}(a) = [5, 7]$
$\text{Live}(b) = [6, 10]$
$\text{Live}(c) = [?, 9]$
$? = 5$

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)
```

$\text{Live}(a) = [?, 7]$
$\text{Live}(b) = [?, 10]$
$\text{Live}(c) = [?, 9]$
$? = 5$

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)
```

$\text{Live}(a) = [?, 7]$

$\text{Live}(b) = [?, 10]$

$\text{Live}(c) = [4, 9]$

$? = 4$

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)
```

$\text{Live}(a) = [?, 7]$
$\text{Live}(b) = [3, 10]$
$\text{Live}(c) = [4, 9]$
$? = 3$

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) $= [3, 10]$
Live(c) $= [?, 9]$
$? = 3$

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)
```

$\text{Live}(a) = [?, 7]$
$\text{Live}(b) = [3, 10]$
$\text{Live}(c) = [2, 9]$
$? = 2$

Liveness intervals: proof of correctness

A modified algorithm

```
1  write(a)
2  write(c)
   if ... :
3  |  write(b)
4  |  write(c)
   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) $= [2, 9]$
$? = 1$

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 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

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

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

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

Theorem

$\text{begmod}[r] \neq ? \Rightarrow \text{beg}[r] \neq ?$

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

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

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

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

Theorem

$\text{begmod[r]} \neq ? \Rightarrow \text{beg[r]} \neq ?$

Theorem

$\text{beg[r]} \neq ? \Rightarrow \text{end[r]} \neq ?$

Theorem

$\text{end[r]} \neq ? \Rightarrow (\text{begmod[r]} \neq ? \text{ or r is live})$

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

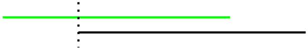
Additional requirements for CakeML's register allocator

- 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

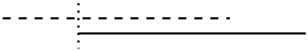
The obvious solution produces bad allocation



Additional requirements for CakeML's register allocator

- Some register must be allocated to a specific color

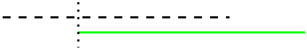
The obvious solution produces bad allocation



Additional requirements for CakeML's register allocator

- Some register must be allocated to a specific color

The obvious solution produces bad allocation

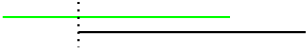


Additional requirements for CakeML's register allocator

- Some register must be allocated to a specific color

The obvious solution produces bad allocation

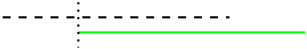
Good solution: only ensure they have different colors, find an exchange afterwards



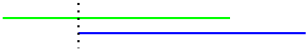
Additional requirements for CakeML's register allocator

- Some register must be allocated to a specific color

The obvious solution produces bad allocation



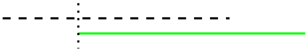
Good solution: only ensure they have different colors, find an exchange afterwards



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 function
- 20 invariants preserved during the execution

Each correctness theorem is of the form:

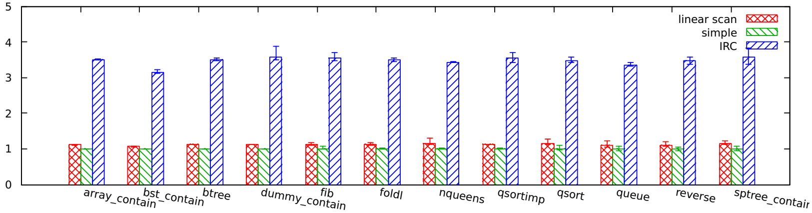
if

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

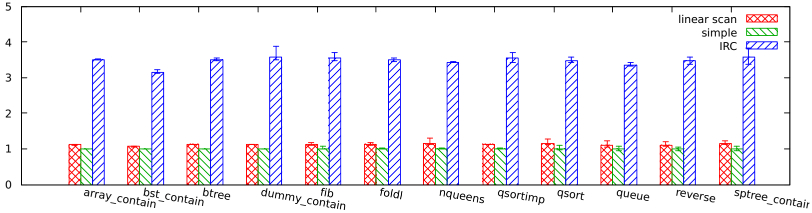
then

- 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]

Performance: compilation time

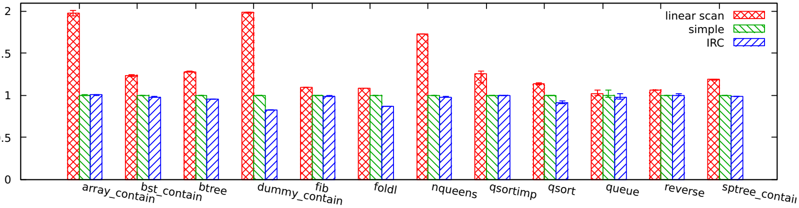


Performance: compilation time

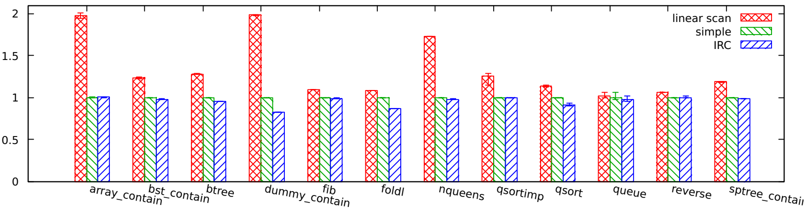


Not that bad, but we would hope better.

Performance: generated code speed

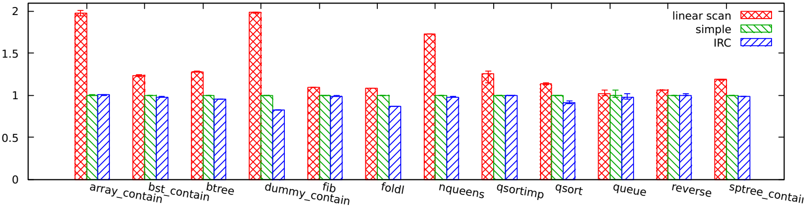


Performance: generated code speed



This is really bad.

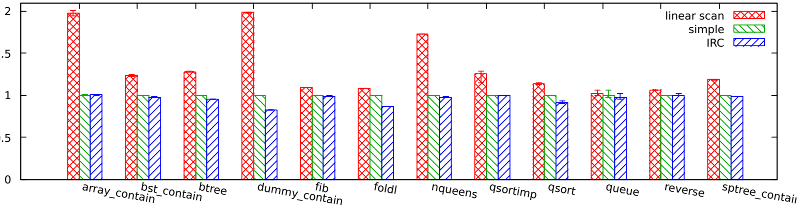
Performance: generated code speed



This is really bad.

The culprit: physical registers have absurdly long liveness intervals

Performance: generated code speed



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.
In *POPL*, pages 208–218. ACM Press, 1996.

Massimiliano Poletto and Vivek Sarkar.
Linear scan register allocation.
ACM Trans. Program. Lang. Syst., 21(5):895–913, 1999.