École Normale Supérieure de Paris
Département d'Informatique

Master 1 internship report

Year 2017–2018

Faster CakeML compilation with a verified linear scan register allocator

Théophile Wallez

Supervised by Magnus O. Myreen

At Chalmers University, Göteborg, Sweden

Acknowledgments

I would like to express my gratitude to Magnus O. Myreen for supervising me during this internship, for always being available when I had questions, and for allowing me to work on such an interesting project.

I would like to thank Oskar Abrahamsson, Andreas Lööw, Johannes Åman Pohjola and Alejandro Gómez-Londoño for their answers to my countless questions about HOL4.

I would also like to thank Yong Kiam Tan for his precious help on the integration of my project in the current codebase.

Thanks to David Reboullet for the numerous coffee-breaks which allowed me to practice rubber-duck debugging on a real human.

Last but not least, thanks to Jean-Christophe Filliâtre for his amazing compilation course, and for telling me about the existence of the CakeML compiler.

Contents

1 Introduction

1.1 The HOL4 theorem prover

HOL4 is an interactive theorem prover for Higher Order Logic (HOL). It is a descendant of LCF, and is similar to Isabelle. Internally, all theorems in HOL4 are proved using basic inference rules [9] The absence of dependent types in HOL4 make convenient proof automation available (such as rewriting and first-order provers).

1.2 The CakeML verified compiler

CakeML is a language based on a subset of Standard ML, with a compiler is written in HOL4. The formal semantics of CakeML is defined in a functional big-step style [7].

The CakeML compiler's backend [10] transforms a source AST (which lacks type annotations) to machine code for one of the five architectures: ARMv6, ARMv8, x86–64, MIPS–64 and RISC–V. The backend has been proved to produce code that has the same behavior as the source program.

The compiler has two frontends: the first one is a traditional parser from CakeML source [5] with a type inferencer [11], both of which have been proved to be sound and complete. The second frontend is a translator from HOL functions to CakeML AST [6, 3], which produces a proof that the generated AST has the same behavior as the HOL function. It can produce code that is stateful and performs I/O.

The compiler can bootstrap itself [5]: the first frontend combined with the backend is a HOL function which transforms a CakeML source code to machine code. This function is given to the second frontend combined with the backend to become machine code which transforms CakeML source code to machine code. Every step in this process is verified end-to-end with HOL4, therefore making CakeML a fully verified compiler.

1.3 The internship goal

The algorithm used in the register allocator of CakeML is the iterated register coalescing algorithm [1], which can be slow: its complexity is at least $\Omega(n^2)$ where n is the number of registers in the program. In fact, most of the compilation time is spent in the register allocation: it is the slowest part of the compiler.

For several applications (JIT, REPL, running on a verified processor on a FPGA) it would be very useful to have a short compilation time, and implementing the linear scan register allocator [8], would be useful.

The goal of my internship was to implement and verify a linear scan register allocator, and integrate it in the CakeML code base.

2 First work on CakeML: improving the constant folding

CakeML is a big and complex project, and starting by doing the linear scan register allocator first seemed a bit too ambitious. Therefore, I started by improving the code for constant propagation. I noticed that the algorithm was able to fold expressions like $x + (1 + 2)$ to $x + 3$, but was failing to fold expressions like $1 + (x + 2)$ or even $(x + 1) + 2$.

The constant folding is done by using smart constructors. A smart constructor SmartOp *op* exp_1 exp_2 is in general equal to Op *op* exp_1 exp_2 but can be smarter in special cases: for example SmartOp Add exp_1 (Number 0) can be equal to exp_1.

The smart constructor was previously implemented with code similar to the one below:

```
SmartOp op exp1 exp2 =
 case (exp1,exp2) of
 | (Number n1,Number n2) ⇒
    if op = Add then Number (n1 + n2)
    else if op = Sub then Number (n1 − n2)
    else if op = Mul then Number (n1 × n2)
    else Op op exp1 exp2
 | _ ⇒ Op op exp1 exp2
```

We can see that this code can't fold expressions like $(x + 1) + 2$.

When implementing the smart constructor, we need to be careful not to change the order of evaluation, since expressions can have side-effects executed. The smart constructor does not need to be careful of overflows since CakeML supports big integers by default.

The algorithm applies this smart-constructor in a bottom-up way: the arguments of the smart-constructor were also constructed using the smart-constructor hence the algorithm can fold expressions like $x + (1 + (2 + (3 + 4)))$ into $x + 10$.

Previously, SmartOp *op* exp_1 exp_2 only checked if both exp_1 and exp_2 were Numbers: I extended it to check if exp_1 and exp_2 have the form Number n, Op *op* (Number n) *exp* or Op *op* *exp* (Number n).

It is possible to remove the case Op *op* *exp* (Number n) by noticing that $exp + n = n + exp$, $exp \times n = n \times exp$ and $exp - n = -n + exp$. After doing theses rewrites it is sufficient to check if exp_1 or exp_2 have the form Number n or Op *op* (Number n) *exp*.

Then, by enumerating all the cases it is possible to see that every combination of Add and Sub are foldable without changing the order of evaluation (for example, $(n_1 - x_1) - (n_2 - x_2) = (n_1 - n_2) - (x_1 - x_2)$)

I implemented this improvement in the intermediate language BVL, a functional language without closures, and I updated the proofs.

The pull request is at the following url: `https://github.com/CakeML/cakeml/pull/485`

3 Preliminaries

3.1 What is register allocation

During optimisation passes, compilers generally use a model with an infinite number of virtual registers. However, computers have a small fixed number of physical registers, hence we must map the virtual registers to use this small number of physical registers.

Since it might not be possible to fit all virtual registers on a small number of physical registers, we have a mechanism to "spill" some virtual registers to the stack. Therefore, we can still use an infinite number of physical registers, but we should avoid spilling virtual registers as much as possible since it degrades the performance of the code produced.

Two virtual registers must not be allocated to the same physical register when their value might be useful at the same time in the program. To avoid overlap, we compute the positions in the program where each virtual register might hold a useful value for the rest of the program execution.

At points the program where a virtual register might hold a useful value for the rest of the program, we say that the virtual register is live. Notice the word "might": without this word, computing the set of live registers at the position of a program is undecidable, so we compute a superset of the real live registers.

To compute the set of live registers at each position of the program, we could use Kildall's algorithm [4] which is the classical algorithm used to perform liveness analysis. However, in CakeML, the control-flow graphs have no cycles because loops are implemented as tail-recursive functions. In fact, we will see in

Section 3.4 that control-flow graphs in CakeML have a very simple shape: we don't need the complexity of Kildall's algorithm and the liveness analysis is more simple.

3.2 The linear scan register allocation algorithm

In this section, we present the linear scan register allocation algorithm [8], which is orders of magnitude faster than the iterated register coalescing algorithm [1] and produces machine code of only slightly lower quality.

First, the liveness analysis is performed, and the control-flow graph is flattened. After that, holes in the liveness intervals are removed: this is the key approximation made by the algorithm to achieve its speed.

For example, in the following example code:

```
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 = {}                                                    */
```

The liveness intervals are:
Live(a) $= \{1, 2, 5\} \subset [1, 5]$
Live(b) $= \{2, 3\} \subset [2, 3]$
Live(c) $= \{4, 6, 7\} \subset [4, 7]$

Each register has a liveness interval, and two register interfere when their liveness intervals intersect.

The algorithm iterates over all registers, sorted by increasing beginning of liveness intervals. During the algorithm, two pieces of information are kept track of:

- the list of active intervals at the current point of the program: intervals that contain the start point of the interval we are currently processing
- a color pool, which contains the physical registers that are not used by registers in the active list

The colors are assigned greedily. When there are no colors available, the algorithm spills to the stack the register with the maximum end of liveness interval.

The algorithm produces a correct coloration, because if two intervals intersect, then one contains the beginning of the other, and they cannot get assigned the same color because when processing the second interval, the first one will be in the active list hence its color won't be in the colorpool.

More precisely, the algorithm works as follow:

```
Function LinearScanRegisterAllocation()
    active = {}
    foreach live interval i, in order of increasing startpoint[i] :
        ExpireOldIntervals(i)
        if colorpool is not empty :
            col ← color removed from colorpool
            color[i] ← col
            add i to active
        else:
            SpillInterval(i)
end
Function ExpireOldIntervals(i)
    foreach live interval j in active such that endpoint[j] < startpoint[i] :
        remove j from active
        add color[j] to the colorpool
end
Function SpillInterval(i)
    tospill ← argmax(endpoint[j] for j ∈ active)
    if endpoint[tospill] > endpoint[i] :
        color[i] ← color[tospill]
        color[tospill] ← new stack location
        remove tospill from active
        add i to active
    else:
        color[i] ← new stack location
end
```

3.3 HOL4's standard library

In this report, we will use several functions and datatypes defined in HOL4's standard library.

unit is the type with one element named (), num represents a natural number (including zero), int is an integer. They both are infinite, there is no maximal value.

α list is a list of objects with type α. mem x l is true iff x is in the list l. every P l is true iff the predicate P is true on every element of l. el n l is the nth element of the list l. map and filter behave like the usual map and filter functions in other functional languages. x::xs is the list xs prepended with x. l_1 ++ l_2 is the concatenation of l_1 and l_2. all_distinct l is true iff l every element of l is unique. length l is the length of l.

$\alpha \times \beta$ represents a pair. The first and second element can be accessed with the fst and snd functions.

α option is either None or Some v. The value can be extracted by the partial function THE or the total function the: THE (Some v) = v, the d (Some v) = v and the d None = d.

α num_map represents a function from a subset of num to α, or equivalently a total function from num to α option. lookup k s returns the value associated with k in the α num_map s. The type of lookup k s is α option, and lookup k s = None means that there is no element associated with k in s. insert k v s add the new mapping $k \mapsto v$ in s. α num_map is implemented using a trie on the binary form of numbers, therefore the complexity of lookup k s or insert k v s is $O(\log_2(k))$. domain s is the domain of definition of s: domain s = $\{ k \mid \text{lookup } k\ s \neq \text{None} \}$.

num_set represents a subset of num. It is actually a type alias for unit num_map: x is in the subset iff lookup x s = Some (). The empty set is represented by LN. union s_1 s_2 computes the union of s_1 and

s_2, difference s_1 s_2 is the set of elements in s_1 but not in s_2. domain s is the set represented by s.

There is some formalisation of sets, they are purely proofs objects and don't result in any computation. image f set is the image of set by the function f. injective f set checks if the function f is injective on set.

3.4 CakeML's current register allocator

The CakeML compiler has a lot of intermediate languages, and the register allocation happens close to the end of the compilation, in an intermediate language called WordLang.

A register allocation algorithm produces a coloration such that two interfering registers don't have the same color. This is checked by the predicate colouring_ok.

There is a pre-existing theorem stating that if colouring_ok says that a coloration is good, then applying the coloration doesn't change the semantics of the program.

Only the reads and writes of the WordLang AST are relevant to the register allocation. This information is summarized in the clash_tree datatype retrieved with the get_clash_tree function.

The function check_clash_tree checks if a coloration is compatible with the clash tree. Its prototype is: check_clash_tree f *clashtree live flive* where f is the coloration function, *live* is the set of live variables after *clashtree*. Two invariants are maintained: *flive* is f applied to the set *live*, and f is injective on *live*. When the coloration is correct, the function returns Some (*livein*,*flivein*) where *livein* is the set of live variables before *clashtree*. When the coloration is not correct, the function returns None.

The following theorem says that if check_clash_tree says that a coloration is good then colouring_ok is always true. wf_cutsets *prog* is a technical hypothesis without any deep meaning (it says that the cutsets of *prog* are well-formed). LN represents the empty set of the num_set datastructure.

```
⊢ wf_cutsets prog ∧
  check_clash_tree f (get_clash_tree prog) LN LN = Some (livein,flivein) ⇒
    colouring_ok f prog LN
```

The clash_tree datatype is defined like this:

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

Delta *writes reads* represents an instruction that writes to a list of registers and reads from a list of registers.

Set *cutset* represents a cutset, the set of local variables which must be preserved past subroutine calls. It is useful for the garbage collector.

Branch *optcutset* ct_1 ct_2 represents an if condition: the two programs are the if program and the else program. *optcutset* is an optional cutset.

Seq ct_1 ct_2 represent the concatenation of two programs

The correctness theorem of the current register allocator looks like this:

$$
\begin{array}{l}
\vdash \mathsf{every}\ (\lambda\,(x,y).\ \mathsf{in\_clash\_tree}\ ct\ x \wedge \mathsf{in\_clash\_tree}\ ct\ y)\ \mathit{forced} \Rightarrow \\
\quad \exists\, \mathit{spcol}\ \mathit{livein}\ \mathit{flivein}. \\
\qquad \mathsf{reg\_alloc}\ \mathit{alg}\ \mathit{sc}\ k\ \mathit{moves}\ \mathit{ct}\ \mathit{forced} = \mathsf{Success}\ \mathit{spcol} \wedge \\
\qquad \mathsf{check\_clash\_tree}\ (\mathsf{sp\_default}\ \mathit{spcol})\ \mathit{ct}\ \mathsf{LN}\ \mathsf{LN} = \mathsf{Some}\ (\mathit{livein},\mathit{flivein}) \wedge \\
\qquad (\forall\, x. \\
\qquad\quad \mathsf{in\_clash\_tree}\ \mathit{ct}\ x \Rightarrow \\
\qquad\quad\ x \in \mathsf{domain}\ \mathit{spcol} \wedge \mathsf{if}\ \mathsf{is\_phy\_var}\ x\ \mathsf{then}\ \mathsf{sp\_default}\ \mathit{spcol}\ x = x\ \mathsf{div}\ 2 \\
\qquad\quad\ \mathsf{else\ if}\ \mathsf{is\_stack\_var}\ x\ \mathsf{then}\ k \leq \mathsf{sp\_default}\ \mathit{spcol}\ x\ \mathsf{else}\ \mathsf{T}) \wedge \\
\qquad (\forall\, x.\ x \in \mathsf{domain}\ \mathit{spcol} \Rightarrow \mathsf{in\_clash\_tree}\ \mathit{ct}\ x) \wedge \\
\qquad \mathsf{every}\ (\lambda\,(x,y).\ \mathsf{sp\_default}\ \mathit{spcol}\ x = \mathsf{sp\_default}\ \mathit{spcol}\ y \Rightarrow x = y)\ \mathit{forced}
\end{array}
$$

Some registers represent physical registers, and shouldn't be allocated to any other register. They are recognised using the predicate is_phy_var. It is useful for two things: calling conventions, and some machine instructions which requires specific registers as input (e.g. x86_64's `idivq` uses only `%rax` and `%rdx`).

Some registers should be allocated on the stack: they are recognised using the predicate is_stack_var

The list *forced* is a list of pair of registers that should not be allocated to the same physical register. It is a constraint coming from some machine instructions in MIPS, RISC-V and ARMv8, where the input should not be equal to the output.

The first part of the correctness theorem is a function call to check_clash_tree which says that the coloration produced is a correct coloration. The second part says that registers that should be allocated on the stack are on the stack, and that physical registers are allocated to themselves. The last part says that the *forced* requirements are satisfied.

To ensure that the new allocator fits nicely in the CakeML codebase, it should try to have a correctness theorem as similar as the above pre-existing one.

4 Implementation and correctness of the algorithm in HOL4

Everything presented in this section is my own work.

The code is available here: `https://github.com/CakeML/cakeml/blob/master/compiler/backend/reg_alloc/linear_scanScript.sml`

The proofs are available here: `https://github.com/CakeML/cakeml/blob/master/compiler/backend/reg_alloc/proofs/linear_scanProofScript.sml`

The first step of the algorithm is to remove cutsets, because the linear scan algorithm can't do anything smart with cutsets. Doing this also allows to have a simpler color-checking function, hence simplify the proofs. The second step is to compute the liveness intervals. The third and last step is the actual linear scan algorithm.

4.1 Step 1: remove cutsets

The set of live registers is computed backwards. The reason for this is that when a register is written to, its value might not be useful after (e.g. it might never be read). However, when a register is read, we know that its value is useful earlier in the program.

Using the clash_tree, the set of live registers is computed by induction like this:

```
get_live_backward_ct (Delta writes reads) live =
  union (difference live (list_to_numset writes)) (list_to_numset 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 =
  union (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)
```

Cutsets provide liveness information which is too precise for the rough liveness intervals that the linear scan algorithm uses. It also makes proofs harder.

Therefore, we simplify clash_tree into live_tree which does not contain any cutsets, and Delta is split in two: Writes and Reads.

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

Because this transformation loses information, it means that we can't compute sets of live variables as accurately as before. However, this is not a problem, since we can compute an over-approximation of the set of live variables.

Instead of treating cutsets as a fresh set of live registers, they are added to the set of live registers: we transform cutsets into Reads.

In fact, we don't lose much information by doing this because cutsets are already live registers: adding cutsets to the set of live registers is the same as ignoring cutsets.

Therefore, we can transform a clash_tree into a live_tree like this:

```
get_live_tree (Delta wr rd) =
  Seq (Reads rd) (Writes wr)
get_live_tree (Set cutset) =
  Reads (numset_to_list cutset)
get_live_tree (Branch None ct1 ct2) =
  Branch (get_live_tree ct1) (get_live_tree ct2)
get_live_tree (Branch (Some cutset) ct1 ct2) =
  Seq (Reads (numset_to_list cutset))
    (Branch (get_live_tree ct1) (get_live_tree ct2))
get_live_tree (Seq ct1 ct2) =
  Seq (get_live_tree ct1) (get_live_tree ct2)
```

The function to compute live variables is now very simple:

```
get_live_backward (Writes l) live =
  difference live (list_to_numset l)
get_live_backward (Reads l) live =
  union live (list_to_numset l)
get_live_backward (Branch ct1 ct2) live =
  union (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)
```

The function check_live_tree is like check_clash_tree, but for live_tree.
We can prove the following correctness theorem:

$$\vdash \mathsf{check\_live\_tree}\ col\ (\mathsf{get\_live\_tree}\ ct)\ \mathsf{LN}\ \mathsf{LN} = \mathsf{Some}\ (livein,flivein) \Rightarrow \\ \exists\, livein'\ flivein'.\ \mathsf{check\_clash\_tree}\ col\ ct\ \mathsf{LN}\ \mathsf{LN} = \mathsf{Some}\ (livein',flivein')$$

This theorem is proved using the following lemma, which is proved by induction on the clash-tree ct.

$$\vdash \mathsf{image}\ col\ (\mathsf{domain}\ live) = \mathsf{domain}\ flive \wedge \mathsf{image}\ col\ (\mathsf{domain}\ live') = \mathsf{domain}\ flive' \wedge \\ \mathsf{injective}\ col\ (\mathsf{domain}\ live') \wedge \\ \mathsf{domain}\ live \subseteq \mathsf{domain}\ live' \wedge \\ \mathsf{check\_live\_tree}\ col\ (\mathsf{get\_live\_tree}\ ct)\ live'\ flive' = \mathsf{Some}\ (livein',flivein') \Rightarrow \\ \exists\, livein\ flivein. \\ \quad \mathsf{check\_clash\_tree}\ col\ ct\ live\ flive = \mathsf{Some}\ (livein,flivein) \wedge \\ \quad \mathsf{domain}\ livein \subseteq \mathsf{domain}\ livein'$$

The hypothesis image col (domain $live$) = domain $flive$ and image col (domain $live'$) = domain $flive'$ are invariants maintained by check_clash_tree and check_live_tree. The hypothesis injective col (domain $live'$) is also an invariant maintained by check_clash_tree and check_live_tree.

We also have $livein'$ = get_live_backward (get_live_tree ct) $live'$ and $livein$ = get_live_backward_ct ct $live$ (which are properties satisfied by check_clash_tree and check_live_tree). We also saw before that by construction, domain (get_live_backward_ct ct $live$) $\subseteq$ domain (get_live_backward (get_live_tree ct) $live$). This is why the hypothesis domain $live \subseteq$ domain $live'$ is here, for the theorem to compose well when dealing with the induction case ct = Seq ct_1 ct_2.

4.2 Step 2: get the liveness intervals

4.2.1 The get_intervals function

We will now compute the liveness intervals as described in Section 3.2

The function get_intervals takes a live_tree and returns two int num_map: the beginning and the end of liveness interval for each register.

This function is very simple: when it treats a Writes the beginning of intervals are extended, when it treats a Reads the end of intervals are extended.

The function traverses the live_tree backwards, and maintains an integer n which represents the position in the program. We write it like this:

$$\mathsf{get\_intervals}\ (\mathsf{Writes}\ l)\ n\ int\_beg\ int\_end = \\ \quad (n-1,\mathsf{numset\_list\_add\_if\_lt}\ l\ n\ int\_beg,\mathsf{numset\_list\_add\_if\_gt}\ l\ n\ int\_end) \\ \mathsf{get\_intervals}\ (\mathsf{Reads}\ l)\ n\ int\_beg\ int\_end = \\ \quad (n-1,int\_beg,\mathsf{numset\_list\_add\_if\_gt}\ l\ n\ int\_end) \\ \mathsf{get\_intervals}\ (\mathsf{Branch}\ lt_1\ lt_2)\ n\ int\_beg\ int\_end = \\ \quad \mathsf{let}\ (n_2,int\_beg_2,int\_end_2)\ =\ \mathsf{get\_intervals}\ lt_2\ n\ int\_beg\ int\_end \\ \quad \mathsf{in} \\ \quad\quad \mathsf{get\_intervals}\ lt_1\ n_2\ int\_beg_2\ int\_end_2 \\ \mathsf{get\_intervals}\ (\mathsf{Seq}\ lt_1\ lt_2)\ n\ int\_beg\ int\_end = \\ \quad \mathsf{let}\ (n_2,int\_beg_2,int\_end_2)\ =\ \mathsf{get\_intervals}\ lt_2\ n\ int\_beg\ int\_end \\ \quad \mathsf{in} \\ \quad\quad \mathsf{get\_intervals}\ lt_1\ n_2\ int\_beg_2\ int\_end_2$$

The numset_list_add_if_lt function verifies the following property:

$$\begin{array}{l}\vdash \mathsf{lookup}\ r\ (\mathsf{numset\_list\_add\_if\_lt}\ l\ v\ s) = \\ \quad \mathsf{if\ mem}\ r\ l\ \mathsf{then} \\ \quad\ \mathsf{case\ lookup}\ r\ s\ \mathsf{of} \\ \quad\quad \mathsf{None}\ \Rightarrow\ \mathsf{Some}\ v \\ \quad\quad \mid \mathsf{Some}\ vr\ \Rightarrow\ \mathsf{if}\ v \leq vr\ \mathsf{then\ Some}\ v\ \mathsf{else\ Some}\ vr \\ \quad \mathsf{else\ lookup}\ r\ s\end{array}$$

The numset_list_add_if_gt function verifies a similar property, with the opposite inequality.

Now, the function which checks a coloration with the liveness intervals is quite simple:

$$\begin{array}{l}\mathsf{check\_intervals}\ f\ \mathit{int\_beg}\ \mathit{int\_end}\ \Longleftrightarrow \\ \forall\, r_1\ r_2. \\ \quad r_1 \in \mathsf{domain}\ \mathit{int\_beg} \wedge r_2 \in \mathsf{domain}\ \mathit{int\_beg} \wedge \\ \quad \mathsf{interval\_intersect}\ (\mathsf{THE}\ (\mathsf{lookup}\ r_1\ \mathit{int\_beg}), \mathsf{THE}\ (\mathsf{lookup}\ r_1\ \mathit{int\_end})) \\ \quad\ (\mathsf{THE}\ (\mathsf{lookup}\ r_2\ \mathit{int\_beg}), \mathsf{THE}\ (\mathsf{lookup}\ r_2\ \mathit{int\_end})) \wedge f\ r_1 = f\ r_2 \Rightarrow \\ \quad\ r_1 = r_2\end{array}$$

And we have the following correctness theorem:

$$\begin{array}{l}\vdash (\mathit{n\_out}, \mathit{beg\_out}, \mathit{end\_out}) = \mathsf{get\_intervals}\ (\mathsf{fix\_domination}\ \mathit{lt})\ 0\ \mathsf{LN}\ \mathsf{LN} \wedge \\ \quad \mathsf{check\_intervals}\ f\ \mathit{beg\_out}\ \mathit{end\_out} \Rightarrow \\ \quad\ \exists\, \mathit{liveout}\ \mathit{fliveout}.\ \mathsf{check\_live\_tree}\ f\ (\mathsf{fix\_domination}\ \mathit{lt})\ \mathsf{LN}\ \mathsf{LN} = \mathsf{Some}\ (\mathit{liveout}, \mathit{fliveout})\end{array}$$

(the fix_domination function will be explained in Section 4.2.2)

To prove this theorem, we can prove that at every position where a register is live, this position is in the register's liveness interval. We can start by proving that the position is less that the end of the liveness interval:

$$\begin{array}{l}\vdash (\mathit{n\_out}, \mathit{beg\_out}, \mathit{end\_out}) = \mathsf{get\_intervals}\ \mathit{lt}\ \mathit{n\_in}\ \mathit{beg\_in}\ \mathit{end\_in} \wedge \\ \quad (\forall\, r.\ r \in \mathsf{domain}\ \mathit{live\_in} \Rightarrow \exists\, v.\ \mathsf{lookup}\ r\ \mathit{end\_in} = \mathsf{Some}\ v \wedge \mathit{n\_in} \leq v) \Rightarrow \\ \quad\ \mathsf{check\_number\_property} \\ \quad\quad (\lambda\, n\ \mathit{live}.\ \forall\, r.\ r \in \mathsf{domain}\ \mathit{live} \Rightarrow \exists\, v.\ \mathsf{lookup}\ r\ \mathit{end\_out} = \mathsf{Some}\ v \wedge n + 1 \leq v)\ \mathit{lt}\ \mathit{n\_in} \\ \quad\quad \mathit{live\_in}\end{array}$$

The function check_number_property takes a predicate on positions and live variables, and checks if it is true at every position of a live_tree.

We prove this theorem by induction on live_tree, using a few simple lemmas and the following monotonicity theorem:

$$\begin{array}{l}\vdash (\forall\, n'\ \mathit{live'}.\ n - \mathsf{size\_of\_live\_tree}\ \mathit{lt} \leq n' \wedge P\ n'\ \mathit{live'} \Rightarrow Q\ n'\ \mathit{live'}) \wedge \\ \quad \mathsf{check\_number\_property}\ P\ \mathit{lt}\ n\ \mathit{live} \Rightarrow \\ \quad\ \mathsf{check\_number\_property}\ Q\ \mathit{lt}\ n\ \mathit{live}\end{array}$$

We would like to prove a similar property about the beginning of intervals, but it turns out that the induction does not work well. The reason why the induction worked well for the end of intervals is that the algorithm processes the live_tree in a backward manner, so that the position of end of interval of a register is changed before this register becomes live. It is not true for beginning of intervals: beginning of intervals are changed after the registers are live. It means that the property we want to prove is not true locally.

4.2.2 The get_intervals_withlive function

The following function is a modification of get_intervals, with better invariants:

$$
\begin{array}{l}
\vdash (\forall\, l\ n\ int\_beg\ int\_end\ live.\\
\quad \mathsf{get\_intervals\_withlive}\ (\mathsf{Writes}\ l)\ n\ int\_beg\ int\_end\ live =\\
\quad\quad (n-1,\mathsf{numset\_list\_add\_if\_lt}\ l\ n\ int\_beg,\mathsf{numset\_list\_add\_if\_gt}\ l\ n\ int\_end)) \wedge\\
\quad (\forall\, l\ n\ int\_beg\ int\_end\ live.\\
\quad\quad \mathsf{get\_intervals\_withlive}\ (\mathsf{Reads}\ l)\ n\ int\_beg\ int\_end\ live =\\
\quad\quad\quad (n-1,\mathsf{numset\_list\_delete}\ l\ int\_beg,\mathsf{numset\_list\_add\_if\_gt}\ l\ n\ int\_end)) \wedge\\
\quad (\forall\, lt_1\ lt_2\ n\ int\_beg\ int\_end\ live.\\
\quad\quad \mathsf{get\_intervals\_withlive}\ (\mathsf{Branch}\ lt_1\ lt_2)\ n\ int\_beg\ int\_end\ live =\\
\quad\quad\quad \mathsf{let}\ (n_2,int\_beg_2,int\_end_2)\ =\ \mathsf{get\_intervals\_withlive}\ lt_2\ n\ int\_beg\ int\_end\ live;\\
\quad\quad\quad (n_1,int\_beg_1,int\_end_1)\ =\\
\quad\quad\quad\quad \mathsf{get\_intervals\_withlive}\ lt_1\ n_2\ (\mathsf{difference}\ int\_beg_2\ live)\ int\_end_2\ live\\
\quad\quad\quad \mathsf{in}\\
\quad\quad\quad\quad (n_1,\\
\quad\quad\quad\quad\quad \mathsf{difference}\ int\_beg_1\\
\quad\quad\quad\quad\quad\quad (\mathsf{union}\ (\mathsf{get\_live\_backward}\ lt_1\ live)\ (\mathsf{get\_live\_backward}\ lt_2\ live)),int\_end_1)) \wedge\\
\quad \forall\, lt_1\ lt_2\ n\ int\_beg\ int\_end\ live.\\
\quad\quad \mathsf{get\_intervals\_withlive}\ (\mathsf{Seq}\ lt_1\ lt_2)\ n\ int\_beg\ int\_end\ live =\\
\quad\quad\quad \mathsf{let}\ (n_2,int\_beg_2,int\_end_2)\ =\ \mathsf{get\_intervals\_withlive}\ lt_2\ n\ int\_beg\ int\_end\ live;\\
\quad\quad\quad (n_1,int\_beg_1,int\_end_1)\ =\\
\quad\quad\quad\quad \mathsf{get\_intervals\_withlive}\ lt_1\ n_2\ int\_beg_2\ int\_end_2\ (\mathsf{get\_live\_backward}\ lt_2\ live)\\
\quad\quad\quad \mathsf{in}\\
\quad\quad\quad\quad (n_1,int\_beg_1,int\_end_1)
\end{array}
$$

Here, we removed the set of live variables to int_beg, to have the invariant that domain int_beg and domain $live$ are disjoint. This is the theorem stating that get_intervals_withlive preserves this invariant:

$$
\begin{array}{l}
\vdash (n\_out,beg\_out,end\_out) = \mathsf{get\_intervals\_withlive}\ lt\ n\_in\ beg\_in\ end\_in\ live \wedge\\
\quad (\forall\, r.\ r \in \mathsf{domain}\ live \Rightarrow r \notin \mathsf{domain}\ beg\_in) \Rightarrow\\
\quad\quad \forall\, r.\ r \in \mathsf{domain}\ (\mathsf{get\_live\_backward}\ lt\ live) \Rightarrow r \notin \mathsf{domain}\ beg\_out
\end{array}
$$

Intuitively, we don't lose information by doing this because in a well-formed program, when an instruction reads the value of a register, the register has a value, so it was written to before in the program, independently of the path taken in the control-flow graph. We can also formulate this by saying that in a well-formed program, no variable lives at the beginning.

It means that during the execution of get_intervals, when a register r is live, the beginning of its interval will be overwritten, hence this value is not important and we can delete it.

We can think of a value missing in int_beg meaning that the beginning of interval at the end of the execution of the algorithm will be less that the number associated with the current position in the program.

This modification allows us to prove this theorem by induction:

$$
\begin{array}{l}
\vdash (n\_out,beg\_out,end\_out) = \mathsf{get\_intervals\_withlive}\ lt\ n\_in\ beg\_in\ end\_in\ live\_in \wedge\\
\quad (\forall\, r\ v.\ \mathsf{lookup}\ r\ beg\_in = \mathsf{Some}\ v \Rightarrow n\_in \leq v) \wedge (\forall\, r.\ r \in \mathsf{domain}\ live\_in \Rightarrow r \notin \mathsf{domain}\ beg\_in) \Rightarrow\\
\quad\quad \mathsf{check\_number\_property}\\
\quad\quad\quad (\lambda\, n\ live.\ \forall\, r.\ r \in \mathsf{domain}\ live \Rightarrow (\mathsf{case}\ \mathsf{lookup}\ r\ beg\_out\ \mathsf{of}\ \mathsf{None}\ \Rightarrow\ n\_out \mid \mathsf{Some}\ x\ \Rightarrow\ x) \leq n)\\
\quad\quad\quad lt\ n\_in\ live\_in
\end{array}
$$

Based on the previous intuition, we would like to prove that on a well-formed program, those two functions compute the same thing.

But before that: do we actually have a proof that programs are well-formed, meaning that there are no live variables at the beginning of the program?

I found that the easiest way to ensure that this property is true is to force it, using the following function:

$$\begin{array}{l}\vdash \mathsf{fix\_domination}\ lt = \\ \quad \mathsf{let}\ live\ =\ \mathsf{get\_live\_backward}\ lt\ \mathsf{LN} \\ \quad \mathsf{in} \\ \quad\ \ \mathsf{if}\ live = \mathsf{LN}\ \mathsf{then}\ lt \\ \quad \mathsf{else\ Seq\ (Writes\ (map\ fst\ (toAList}\ live))\ lt\end{array}$$

4.2.3 Proving that the two functions compute the same thing

Let's name *int_end* the end of intervals (which are computed the same way in both functions), *int_beg* the beginning of intervals produced by get_intervals, and *int_beg_withlive* the one produced by get_intervals_withlive.

It is easy to prove by induction that when *int_beg* and *int_beg_withlive* share a common key, then the values associated with this key are equal:

$$\begin{array}{l}\vdash (n_1,beg_1,end_1) = \mathsf{get\_intervals}\ lt\ n\ beg\ end \wedge \\ \quad (n_2,beg_2,end_2) = \mathsf{get\_intervals\_withlive}\ lt\ n\ beg'\ end\ live \wedge \\ \quad (\forall\, r\ v.\ \mathsf{lookup}\ r\ beg = \mathsf{Some}\ v \Rightarrow n \leq v) \wedge (\forall\, r\ v.\ \mathsf{lookup}\ r\ beg' = \mathsf{Some}\ v \Rightarrow n \leq v) \wedge \\ \quad (\forall\, r\ v_1\ v_2.\ \mathsf{lookup}\ r\ beg = \mathsf{Some}\ v_1 \wedge \mathsf{lookup}\ r\ beg' = \mathsf{Some}\ v_2 \Rightarrow v_1 = v_2) \Rightarrow \\ \quad\ \ \forall\, r\ v_1\ v_2.\ \mathsf{lookup}\ r\ beg_1 = \mathsf{Some}\ v_1 \wedge \mathsf{lookup}\ r\ beg_2 = \mathsf{Some}\ v_2 \Rightarrow v_1 = v_2\end{array}$$

Now we only have to prove that domain *int_beg* = domain *int_beg_withlive*

We define the set of registers in a live_tree like this:

$$\begin{array}{l}\mathsf{live\_tree\_registers\ (Writes}\ l) = \mathsf{set}\ l \\ \mathsf{live\_tree\_registers\ (Reads}\ l) = \mathsf{set}\ l \\ \mathsf{live\_tree\_registers\ (Branch}\ lt_1\ lt_2) = \mathsf{live\_tree\_registers}\ lt_1 \cup \mathsf{live\_tree\_registers}\ lt_2 \\ \mathsf{live\_tree\_registers\ (Seq}\ lt_1\ lt_2) = \mathsf{live\_tree\_registers}\ lt_1 \cup \mathsf{live\_tree\_registers}\ lt_2\end{array}$$

We will prove the inclusions:

live_tree_registers *lt* ⊂ domain *int_end* ⊂ domain *int_beg_withlive* ⊂ domain *int_beg* ⊂ live_tree_registers *lt*

Proving domain *int_beg* ⊂ live_tree_registers *lt* is easy by induction with the following theorem:

$$\begin{array}{l}\vdash (n\_out,beg\_out,end\_out) = \mathsf{get\_intervals}\ lt\ n\_in\ beg\_in\ end\_in \Rightarrow \\ \quad \mathsf{domain}\ beg\_out \subseteq \mathsf{domain}\ beg\_in \cup \mathsf{live\_tree\_registers}\ lt\end{array}$$

Proving domain *int_beg_withlive* ⊂ domain *int_beg* is easy by induction with the following theorem:

$$\begin{array}{l}\vdash (n_1,beg\_out_1,end_1) = \mathsf{get\_intervals\_withlive}\ lt\ n\ beg\_in_1\ end\ live \wedge \\ \quad (n_2,beg\_out_2,end_2) = \mathsf{get\_intervals}\ lt\ n\ beg\_in_2\ end \wedge \\ \quad \mathsf{domain}\ beg\_in_1 \subseteq \mathsf{domain}\ beg\_in_2 \Rightarrow \\ \quad\ \ \mathsf{domain}\ beg\_out_1 \subseteq \mathsf{domain}\ beg\_out_2\end{array}$$

Proving live_tree_registers *lt* ⊂ domain *int_end* is easy with the following theorem:

$$\begin{array}{l}\vdash (n\_out,beg\_out,end\_out) = \mathsf{get\_intervals\_withlive}\ lt\ n\_in\ beg\_in\ end\_in\ live\_in \Rightarrow \\ \quad \mathsf{domain}\ end\_in \cup \mathsf{live\_tree\_registers}\ lt \subseteq \mathsf{domain}\ end\_out\end{array}$$

The only proof left is live_tree_registers *lt* ⊂ domain *int_beg_withlive*. It turns out it was quite difficult.

Before proving this inclusion, we prove the following lemma which says that get_intervals_withlive is Lipschitz-continuous[1]

$$\begin{array}{l}\vdash (nout_1,begout_1,endout_1) = \mathsf{get\_intervals\_withlive}\ lt\ n_1\ beg_1\ end_1\ live \wedge \\ \quad (nout_2,begout_2,endout_2) = \mathsf{get\_intervals\_withlive}\ lt\ n_2\ beg_2\ end_2\ live \wedge \\ \quad \mathsf{domain}\ beg_2 \subseteq \mathsf{domain}\ beg_1 \cup \mathsf{domain}\ s \Rightarrow \\ \qquad \mathsf{domain}\ begout_2 \subseteq \mathsf{domain}\ begout_1 \cup \mathsf{domain}\ s\end{array}$$

It allows us to prove that domain $int\_end \subset$ domain $int\_beg\_withlive$:

$$\begin{array}{l}\vdash \mathsf{domain}\ end\_in \subseteq \mathsf{domain}\ beg\_in \cup \mathsf{domain}\ live\_in \wedge \\ \quad (n\_out,beg\_out,end\_out) = \mathsf{get\_intervals\_withlive}\ lt\ n\_in\ beg\_in\ end\_in\ live\_in \Rightarrow \\ \qquad \mathsf{domain}\ end\_out \subseteq \mathsf{domain}\ beg\_out \cup \mathsf{domain}\ (\mathsf{get\_live\_backward}\ lt\ live\_in)\end{array}$$

Proof sketch it is proved by induction on lt. The only hard case is $lt = \mathsf{Branch}\ lt_1\ lt_2$. Let's write
$(n_2,beg_2,end_2) = \mathsf{get\_intervals\_withlive}\ lt_2\ n\_in\ beg\_in\ end\_in\ live\_in$
and
$(n_1,beg_1,end_1) = \mathsf{get\_intervals\_withlive}\ lt_1\ n_2\ (\mathsf{difference}\ beg_2\ live\_in)\ end_2\ live\_in$.
By the induction hypothesis, we have
$\mathsf{domain}\ end_2 \subseteq \mathsf{domain}\ beg_2 \cup \mathsf{domain}\ (\mathsf{get\_live\_backward}\ lt_2\ live\_in)$.
Unfortunately, we can't use the induction hypothesis for lt_1 since we don't have the hypothesis
$\mathsf{domain}\ end_2 \subseteq \mathsf{domain}\ (\mathsf{difference}\ beg_2\ live\_in) \cup \mathsf{domain}\ live\_in$.
We can force the hypothesis: let's write
$(n_1,beg_1',end_1) =$
$\mathsf{get\_intervals\_withlive}\ lt_1\ n_2\ (\mathsf{difference}\ (\mathsf{union}\ beg_2\ (\mathsf{get\_live\_backward}\ lt_2\ live\_in))\ live\_in)\ end_2\ live\_in$.
We can see that
$\mathsf{domain}\ (\mathsf{difference}\ (\mathsf{union}\ beg_2\ (\mathsf{get\_live\_backward}\ lt_2\ live\_in))\ live\_in) \cup \mathsf{domain}\ live\_in =$
$\mathsf{domain}\ beg_2 \cup \mathsf{domain}\ (\mathsf{get\_live\_backward}\ lt_2\ live\_in) \cup \mathsf{domain}\ live\_in$
which contains end_2 as we saw at the beginning of the proof.
From this, we can deduce using the induction hypothesis that
$\mathsf{domain}\ end_1 \subseteq \mathsf{domain}\ beg_1' \cup \mathsf{domain}\ (\mathsf{get\_live\_backward}\ lt_1\ live\_in)$
and by the Lipschitz-continuity theorem, we have
$\mathsf{domain}\ beg_1' \subseteq \mathsf{domain}\ beg_1 \cup \mathsf{domain}\ (\mathsf{get\_live\_backward}\ lt_2\ live\_in)$.
Therefore,
$\mathsf{domain}\ end_1 \subseteq$
$\mathsf{domain}\ beg_1 \cup \mathsf{domain}\ (\mathsf{get\_live\_backward}\ lt_1\ live\_in) \cup \mathsf{domain}\ (\mathsf{get\_live\_backward}\ lt_2\ live\_in)$
which proves the theorem. □

Using all the previous lemmas, we can prove the following theorem:

$$\begin{array}{l}\vdash (n,beg,end) = \mathsf{get\_intervals\_withlive}\ (\mathsf{fix\_domination}\ lt)\ 0\ \mathsf{LN}\ \mathsf{LN}\ \mathsf{LN} \wedge \\ \quad (n',beg',end') = \mathsf{get\_intervals}\ (\mathsf{fix\_domination}\ lt)\ 0\ \mathsf{LN}\ \mathsf{LN} \Rightarrow \\ \qquad \forall\, r.\ \mathsf{lookup}\ r\ beg = \mathsf{lookup}\ r\ beg'\end{array}$$

4.3 Step 3: do the allocation

4.3.1 Modifications to the original algorithm

In section 3.4, we saw that CakeML's register allocation needs a few features not covered by the original linear scan register allocation algorithm [8].

[1]When f is an increasing function, the Lipschitz-continuity with a constant equal to 1 translates to $\forall x \forall y > 0, f(x+y) \leq f(x) + y$. Here the theorem says something like $f(T \cup S) \subset f(T) \cup S$

These features are:

- physical registers should be allocated themselves (e.g. for calling conventions)
- stack registers should be on the stack
- the *moves* list is a list of pair of registers that should be allocated to the same register if possible (to remove useless "move" instructions)
- the *forced* list is a list of pair of registers that can't be allocated to the same register
- registers that are spilled on the stack should be allocated in a "smart" way to minimise the stack-frame size

Small stack-frame size We used the classical solution to optimize the stack-frame size, which is to do the allocation in two passes. The first pass is the allocation as we saw previously, with a new fresh register on the stack each time we spill a register. During the second pass, we run the same algorithm only on the registers spilled on the stack to reallocate them in a smarter way.

The *moves* list The requirement that some pair of registers should be allocated to the same register if possible is easy to satisfy: before choosing a color from the color pool, check if a color of one of the registers in the *moves* list is available.

Stack registers The requirement that stack registers should be allocated on the stack is also easy to satisfy: when we encounter a such register, we can simply spill it.

The *forced* list The requirement that some pair of registers should not be allocated to the same register can be satisfied by adding some sanity check when we choose a color from the color-pool.

Physical registers The requirement that physical registers should be allocated to themselves is harder to satisfy. When we allocate the physical register *reg* which should have the color *col*, and that *col* is used by a register reg_2 in the active list, what should we do? It's not possible to give a smart color to reg_2 to free the color *col*: the only option is to spill reg_2. But that's a problem since it would produce very bad allocation.

I found a good solution to this, namely, to forget about this requirement, and fix the coloration afterward to fit this requirement. How do we fix the coloration? Afterward, we find a permutation of colors to ensure that each physical register is allocated to itself. A necessary and sufficient condition for a such permutation to exist, is that the physical registers must have different colors.

In summary: to ensure that physical registers are allocated to themselves, we first do the allocation and ensure that physical registers all have distinct colors, and we do a color permutation afterwards.

4.3.2 The state and invariants used in the linear scan algorithm

The internal state of the algorithm is represented by the two following records:

```
linear_scan_state = ⟪
  active : (int × num) list;
  colorpool : num list;
  phycols : num_set;
  colornum : num;
  colormax : num;
  stacknum : num
⟫
```

```
linear_scan_hidden_state = ⟪
  colors : num list
⟫
```

active is the list of active registers: the second component of the pair is the active register, the first component is the register's end of liveness interval. This list is sorted by increasing end of liveness interval. colorpool is the pool of available colors, along with the colors c such that colornum $\leq c <$ colormax. phycols is the set of colors used by the physical registers (to ensure the colors are all distinct). stacknum is the color of a fresh register to spill on the stack.

colors is an array such that its *reg*th element is the color of *reg*. In the state it is represented as a functional list, but when the HOL function is translated into a CakeML AST during the bootstrap process, it is represented as a static array with $O(1)$ element lookup. This is why colors is in a separate record. This record is said "hidden" because it is used in a state monad.

This state has a lot of invariants that are preserved during the algorithm. The invariants depend on several parameters: *int_beg* and *int_end* the beginning and end of liveness intervals, *st* and *sth* the state and the hidden state, the *forced* list, a list l of register that were processed by the algorithm, *pos* representing the position of the beginning of liveness interval of the last processed register.

The most important invariants are the one directly linked to the final correctness theorem:

- two different registers in l with intersecting liveness intervals must not have the same color
- two different registers in l and in the *forced* list must not have the same color

Some invariants describe the content of the active list:

- the active list is sorted by its first component
- all registers in the active list are also present in the l list
- the color of registers present in the active list are strictly lower that colornum
- for each element of active, the first component is equal to the end of liveness interval of the second component
- if a register of l is not spilled to the stack and its end of liveness interval is greater than *pos* then it is in the active list
- if a register r is in active then *pos* is less than its end of liveness interval incremented by one[2]

Some invariant describe the relationship between the colors manipulated by the algorithm:

- the colors of colorpool concatenated with the colors of the registers of active are all distinct[3]
- the color of physical registers are all distinct

[2] the increment is necessary to prove the correctness of `ExprireOldIntervals` by induction, in the case we are removing several intervals with the same endpoint

[3] this invariant also says that the colors in colorpool and the colors of registers in active are disjoint

And then there are miscellaneous invariants:

- the color of every register in l is strictly less than stacknum
- every color in colorpool is strictly less than colornum
- colornum is less than colormax which is less than stacknum
- if the color of a register of l is less than colormax then it is less than colornum
- the beginning of liveness interval of every register in l is less than pos
- phycols is the set of colors of registers reg in l such that reg is a physical register and reg is not spilled to the stack
- every register of l is strictly less than the length of colors

The invariants have an additional parameter $mincol$ which is the minimal color used in the algorithm. It is useful to prove that the second pass that reallocates spilled registers keeps registers on the stack. It follows the following invariants:

- $mincol$ is less than colornum
- $mincol$ is less than every color in colorpool
- $mincol$ is less than the color of every register in active

The invariants are checked by the predicate good_linear_scan_state int_beg int_end st sth l pos $forced$ $mincol$

4.3.3 Implementation of the linear scan algorithm

In this section, we present the different functions used to implement the linear scan algorithm, along with the correctness theorems. The correctness theorems usually says that the function doesn't fail (i.e. there is no array out-of-bounds), that the invariants are preserved, that length colors and colormax are not changed, and express which colors are changed.

The remove_inactive_intervals beg st sth function corresponds to the `ExpireOldIntervals` function presented in Section 3.2: it removes active intervals whose endpoint is before beg.

It has the following correctness theorem (notice that pos is replaced by beg in the conclusion):

$$
\begin{array}{l}
\vdash \mathsf{good\_linear\_scan\_state}\ int\_beg\ int\_end\ st\ sth\ l\ pos\ forced\ mincol \wedge \\
\quad pos \leq beg \Rightarrow \\
\quad\ \exists\, stout. \\
\qquad (\mathsf{Success}\ stout, sth) = \mathsf{remove\_inactive\_intervals}\ beg\ st\ sth \wedge \\
\qquad \mathsf{good\_linear\_scan\_state}\ int\_beg\ int\_end\ stout\ sth\ l\ beg\ forced\ mincol \wedge \\
\qquad stout.\mathsf{colormax} = st.\mathsf{colormax}
\end{array}
$$

find_color st $forbidden$ tries to find a color in the colorpool which is not in the set $forbidden$. When it succeeds, it returns ($stout$,Some col) where $stout$ is the output state. The output color is removed from the colorpool in the output state.

$$
\begin{array}{l}
\vdash \mathsf{good\_linear\_scan\_state}\ int\_beg\ int\_end\ st\ sth\ l\ pos\ forced\ mincol \wedge \\
\quad \mathsf{domain}\ forbidden \subseteq \{\ \mathsf{el}\ r\ sth.\mathsf{colors} \mid \mathsf{mem}\ r\ l\ \} \wedge \\
\quad \mathsf{find\_color}\ st\ forbidden = (stout, \mathsf{Some}\ col) \Rightarrow \\
\qquad \mathsf{good\_linear\_scan\_state}\ int\_beg\ int\_end \\
\qquad\ (stout\ \mathsf{with\ colorpool}\ :=\ col :: stout.\mathsf{colorpool})\ sth\ l\ pos\ forced\ mincol \wedge \\
\qquad col < stout.\mathsf{colornum} \wedge col \notin \mathsf{domain}\ forbidden \wedge \\
\qquad st = stout\ \mathsf{with}\ \langle\!|\mathsf{colorpool}\ :=\ st.\mathsf{colorpool};\ \mathsf{colornum}\ :=\ st.\mathsf{colornum}|\!\rangle
\end{array}
$$

color_register *st reg col rend sth* assign the color *col* to the register *reg*. It updates the active list with the end of interval *rend*. It has the following correctness theorem:

$$
\begin{array}{l}
\vdash \mathsf{good\_linear\_scan\_state}\ \mathit{int\_beg}\ \mathit{int\_end}\ \mathit{st}\ \mathit{sth}\ \mathit{l}\ \mathit{pos}\ \mathit{forced}\ \mathit{mincol} \wedge \\
\quad \mathsf{forbidden\_is\_from\_map\_color\_forced}\ \mathit{forced}\ \mathit{l}\ \mathit{sth}.\mathsf{colors}\ \mathit{reg}\ \mathit{forbidden} \wedge \mathit{col} \notin \mathsf{domain}\ \mathit{forbidden} \wedge \\
\quad (\mathsf{is\_phy\_var}\ \mathit{reg} \Rightarrow \mathsf{domain}\ \mathit{st}.\mathsf{phycols} \subseteq \mathsf{domain}\ \mathit{forbidden}) \wedge \\
\quad \neg\mathsf{mem}\ \mathit{col}\ (\mathit{st}.\mathsf{colorpool} +\!\!+ \mathsf{map}\ (\lambda\,(e,r).\ \mathsf{el}\ r\ \mathit{sth}.\mathsf{colors})\ \mathit{st}.\mathsf{active}) \wedge \\
\quad \mathsf{the}\ 0\ (\mathsf{lookup}\ \mathit{reg}\ \mathit{int\_beg}) = \mathit{pos} \wedge \mathsf{the}\ 0\ (\mathsf{lookup}\ \mathit{reg}\ \mathit{int\_beg}) \leq \mathsf{the}\ 0\ (\mathsf{lookup}\ \mathit{reg}\ \mathit{int\_end}) \wedge \\
\quad \mathit{col} < \mathit{st}.\mathsf{colornum} \wedge \mathit{mincol} \leq \mathit{col} \wedge \mathit{reg} < \mathsf{length}\ \mathit{sth}.\mathsf{colors} \wedge \neg\mathsf{mem}\ \mathit{reg}\ \mathit{l} \Rightarrow \\
\quad\ \exists\, \mathit{stout}\ \mathit{sthout}. \\
\qquad (\mathsf{Success}\ \mathit{stout},\mathit{sthout}) = \mathsf{color\_register}\ \mathit{st}\ \mathit{reg}\ \mathit{col}\ (\mathsf{the}\ 0\ (\mathsf{lookup}\ \mathit{reg}\ \mathit{int\_end}))\ \mathit{sth} \wedge \\
\qquad \mathsf{good\_linear\_scan\_state}\ \mathit{int\_beg}\ \mathit{int\_end}\ \mathit{stout}\ \mathit{sthout}\ (\mathit{reg}::\mathit{l})\ \mathit{pos}\ \mathit{forced}\ \mathit{mincol} \wedge \\
\qquad \mathsf{length}\ \mathit{sthout}.\mathsf{colors} = \mathsf{length}\ \mathit{sth}.\mathsf{colors} \wedge \\
\qquad (\forall\, r.\ r \neq \mathit{reg} \Rightarrow \mathsf{el}\ r\ \mathit{sth}.\mathsf{colors} = \mathsf{el}\ r\ \mathit{sthout}.\mathsf{colors}) \wedge \mathit{stout}.\mathsf{colormax} = \mathit{st}.\mathsf{colormax}
\end{array}
$$

The hypothesis of this theorem mostly says that *reg* and *col* are compatible with the invariants. An interesting part is the forbidden_is_from_map_color_forced which says that the *forbidden* set contains the color of registers that conflicts with *reg* in *forced*.

spill_register *st reg sth* spills the register *reg* on a fresh stack location. It has the following correctness theorem:

$$
\begin{array}{l}
\vdash (\neg\mathsf{is\_phy\_var}\ \mathit{reg} \vee \neg\mathsf{mem}\ \mathit{reg}\ \mathit{l}) \wedge \\
\quad \mathsf{good\_linear\_scan\_state}\ \mathit{int\_beg}\ \mathit{int\_end}\ \mathit{st}\ \mathit{sth}\ \mathit{l}\ \mathit{pos}\ \mathit{forced}\ \mathit{mincol} \wedge \\
\quad \mathit{reg} < \mathsf{length}\ \mathit{sth}.\mathsf{colors} \wedge \mathsf{the}\ 0\ (\mathsf{lookup}\ \mathit{reg}\ \mathit{int\_beg}) \leq \mathit{pos} \Rightarrow \\
\quad\ \exists\, \mathit{stout}\ \mathit{sthout}. \\
\qquad (\mathsf{Success}\ \mathit{stout},\mathit{sthout}) = \\
\qquad\ \mathsf{spill\_register}\ (\mathit{st}\ \mathsf{with\ active}\ :=\ \mathsf{filter}\ (\lambda\,(e,r).\ r \neq \mathit{reg})\ \mathit{st}.\mathsf{active})\ \mathit{reg}\ \mathit{sth} \wedge \\
\qquad \mathsf{good\_linear\_scan\_state}\ \mathit{int\_beg}\ \mathit{int\_end}\ \mathit{stout}\ \mathit{sthout}\ (\mathit{reg}::\mathit{l})\ \mathit{pos}\ \mathit{forced}\ \mathit{mincol} \wedge \\
\qquad \mathsf{length}\ \mathit{sthout}.\mathsf{colors} = \mathsf{length}\ \mathit{sth}.\mathsf{colors} \wedge \\
\qquad (\forall\, r.\ r \neq \mathit{reg} \Rightarrow \mathsf{el}\ r\ \mathit{sth}.\mathsf{colors} = \mathsf{el}\ r\ \mathit{sthout}.\mathsf{colors}) \wedge \\
\qquad \mathit{stout}.\mathsf{colormax} = \mathit{st}.\mathsf{colormax} \wedge \mathit{st}.\mathsf{colormax} \leq \mathsf{el}\ \mathit{reg}\ \mathit{sthout}.\mathsf{colors}
\end{array}
$$

This theorem is a bit more complex than the other functions, because every function except this one can have the hypothesis ¬mem *reg l*. However as we see in the `SpillInterval` function in Section 3.2 we can spill a register that is in the active list (and therefore in *l*).
If ¬mem *reg l* then *st*.active = filter $(\lambda\,(e,r).\ r \neq reg)$ *st*.active and then this theorem is similar to the usual ones. However, if mem *reg l* we need the requirement ¬is_phy_var *reg* because this function does not update phycols[4]. When we have mem *reg l* then *reg* needs to be removed to the active list: using the filter function is a brutal way to do this[5].

The find_spill *st forbidden reg rend force sth* function corresponds to the `SpillInterval` function in Section 3.2. *force* is a boolean which says that the function should not spill *reg* to the stack if possible (it is best-effort). If the function "steals" the color of an active register, the function makes sure that it is not in the *forbidden* set. It has the following correctness theorem, which is similar to color_register's correctness theorem.

[4] it could update phycols, but it is not done because in practice the hypothesis mem *reg l* ⇒ ¬is_phy_var *reg* is satisfied
[5] hopefully, the filter function only appears in the proof

$\vdash \neg$mem $reg\ l \wedge$
good_linear_scan_state $int\_beg\ int\_end\ st\ sth\ l$ (the 0 (lookup $reg\ int\_beg$)) $forced\ mincol \wedge$
$reg <$ length sth.colors $\wedge$ forbidden_is_from_map_color_forced $forced\ l\ sth$.colors $reg\ forbidden \wedge$
(is_phy_var $reg \Rightarrow$ domain st.phycols $\subseteq$ domain $forbidden$) $\wedge$
the 0 (lookup $reg\ int\_beg$) $\leq$ the 0 (lookup $reg\ int\_end$) $\Rightarrow$
$\exists\, stout\ sthout.$
(Success $stout$,$sthout$) = find_spill $st\ forbidden\ reg$ (the 0 (lookup $reg\ int\_end$)) $force\ sth \wedge$
good_linear_scan_state $int\_beg\ int\_end\ stout\ sthout\ (reg::l)$ (the 0 (lookup $reg\ int\_beg$))
$forced\ mincol \wedge$ length $sthout$.colors = length sth.colors $\wedge$
$(\forall\, r.$ $\neg$mem $r\ (reg::l) \Rightarrow$ el $r\ sthout$.colors = el $r\ sth$.colors) $\wedge$
$(\forall\, r.$ mem $r\ l \wedge$ is_phy_var $r \Rightarrow$ el $r\ sthout$.colors = el $r\ sth$.colors) $\wedge$
$stout$.colormax = st.colormax

apply_reg_exchange $l\ sth$ takes a list l of physical registers, and do the exchange described in Section 4.3.1. The following correctness theorem says that the registers in l are assigned to the right registers, and that the exchange preserves the fact that two registers have the same color.

$\vdash$ all_distinct (map $(\lambda\, r.$ el $r\ sth$.colors) $l) \wedge (\forall\, r.$ mem $r\ l \Rightarrow$ is_phy_var $r) \wedge$
$(\forall\, r.$ mem $r\ l \Rightarrow r <$ length sth.colors) $\Rightarrow$
$\exists\, sthout.$
(Success (),$sthout$) = apply_reg_exchange $l\ sth \wedge$ length $sthout$.colors = length sth.colors $\wedge$
$(\forall\, r_1\ r_2.$
$r_1 <$ length sth.colors $\wedge\ r_2 <$ length sth.colors $\Rightarrow$
el $r_1\ sthout$.colors = el $r_2\ sthout$.colors $\Rightarrow$ el $r_1\ sth$.colors = el $r_2\ sth$.colors) $\wedge$
$\forall\, r.$ mem $r\ l \Rightarrow$ el $r\ sthout$.colors = r div 2

In reality, the theorem has a more complex conclusion which gives information about how it preserves the fact that some registers are on the stack, but it was omitted here for simplicity.

linear_reg_alloc_intervals $int\_beg\ int\_end\ k\ forced\ moves\ reglist\_unsorted\ sth$ is the full algorithm. It does a first pass which corresponds to the original linear scan algorithm [8], then do the color exchange for physical registers that are not on the stack. Then, it runs a second pass for registers that were spilled to the stack and stack registers, then do the exchange for physical registers that are on the stack. This function has the following correctness theorem, which looks a lot like the correctness theorem in the original register allocator as seen in Section 3.4:

$\vdash$ every $(\lambda\,(r_1,r_2).$ mem $r_1\ reglist \wedge$ mem $r_2\ reglist)\ forced \wedge$
every $(\lambda\,(r_1,r_2).\ r_1 <$ length sth.colors $\wedge\ r_2 <$ length sth.colors) (map snd $moves) \wedge$
every $(\lambda\, r.\ r <$ length sth.colors) $reglist \wedge$
every $(\lambda\, r.$ the 0 (lookup $r\ int\_beg) \leq$ the 0 (lookup $r\ int\_end)) reglist \wedge$
all_distinct $reglist \wedge$ set $reglist$ = domain $int\_beg \wedge$ domain int_beg = domain $int\_end \Rightarrow$
$\exists\, sthout.$
(Success (),$sthout$) =
linear_reg_alloc_intervals $int\_beg\ int\_end\ k\ forced\ moves\ reglist\ sth \wedge$
check_intervals $(\lambda\, r.$ el $r\ sthout$.colors) $int\_beg\ int\_end \wedge$
every
$(\lambda\, r.$
if is_phy_var r then el $r\ sthout$.colors = r div 2
else if is_stack_var r then $k \leq$ el $r\ sthout$.colors
else T) $reglist \wedge$
every $(\lambda\,(r_1,r_2).$ el $r_1\ sthout$.colors = el $r_2\ sthout$.colors $\Rightarrow r_1 = r_2)\ forced \wedge$
length $sthout$.colors = length sth.colors

linear_scan_reg_alloc k *moves ct forced* is a function that combines the previous steps. It has exactly the same correctness theorem as the other register allocator seen in Section 3.4

5 Evaluation

The linear scan algorithm is compared against the iterated register coalescing (IRC) algorithm [1], and a simpler allocator which corresponds to the IRC algorithm without coalescing.

It is compared in term of compilation time, and in term of quality of the produced code on the standard benchmarks used for CakeML.

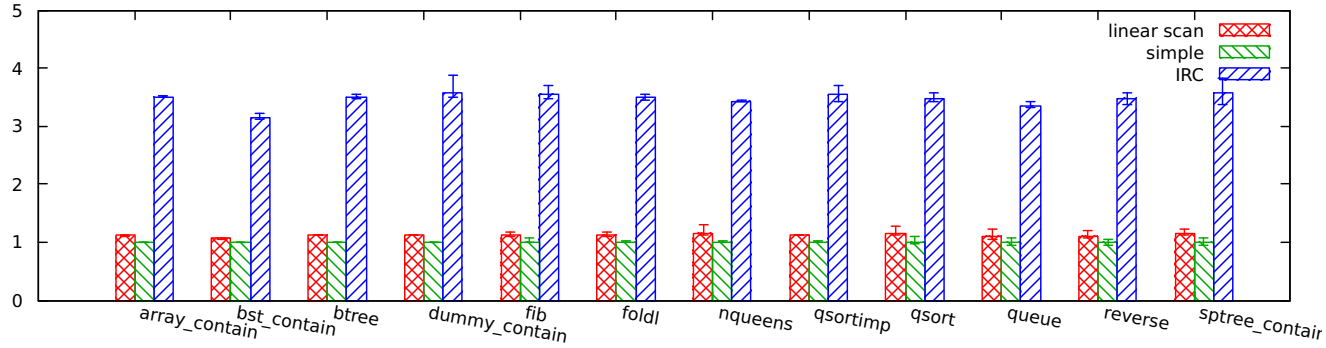

Figure 1: Compilation speed. The time in normalized on the time taken by the simple algorithm

We can see here that the linear scan algorithm is a bit slower than the simple allocator, but about three time faster than the IRC algorithm.

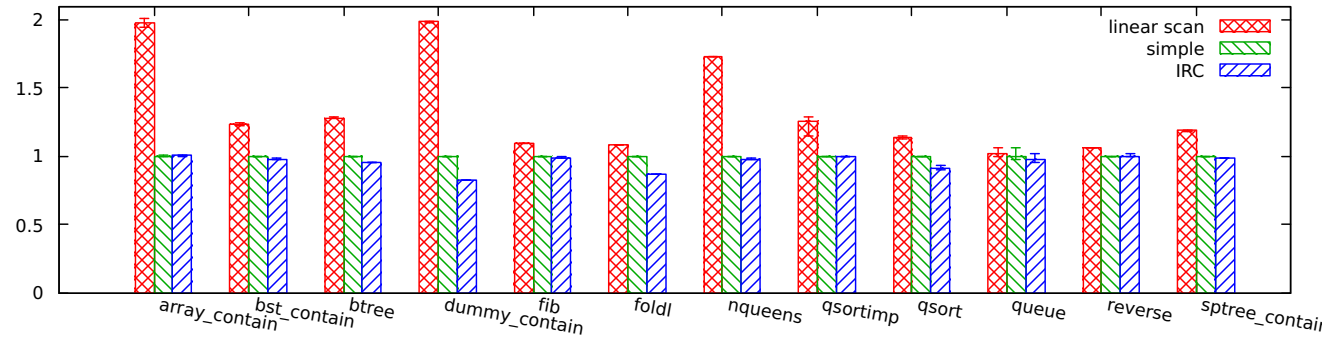

Figure 2: Produced code speed. The time in normalized on the time took by the simple algorithm

We can see here that the linear scan algorithm produces code of bad quality, even compared to the simple algorithm.

Why do we have such results, when the original paper announces that the code produced is about 10% slower than the code generated by the IRC algorithm [8, sec 5.3.2]? We found that the culprit were physical registers, more precisely calling conventions. The calling convention uses the first physical registers as the arguments of the function called. If in a program a function is called at the beginning, and a other function is called at the end, the liveness interval of the first physical register might be for example $\{1, 2, 3, 1001, 1002, 1003\} \subset [1, 1003]$. This means that the physical register might live for a very short time, but its liveness interval will be huge, and this physical register can't be used by another register in the middle of the program.

The solution to this problem is to do the allocation after the SSA pass, but before the calling convention is enforced. For this, we have to split a function in two parts since the SSA-form and the calling convention are enforced by the same function.

6 Future work

6.1 Split the SSA-form and the calling conventions

As we saw in Section 5, in order to produce code of decent quality we must split in two parts the function that transforms the program in SSA form and enforces calling conventions

6.2 Optimise the constant of the algorithm

There are a few things that can be done faster in the whole algorithm:

- we can compute liveness intervals directly on a clash_tree (the live_tree was useful for the proofs)
- we can use static arrays to store the endpoints of the liveness intervals (instead of a int num_map)
- the sorting function used is a quicksort on functional lists: we could write a faster version using static arrays

6.3 Remove the fix_domination function

Since proving the property that the set of live variables is empty at the beginning of the program is not easy, we force it using the function fix_domination. Proving the domination property would allow to remove this function call, which is in practice useless.

6.4 Use the fact that we work on a SSA-like AST

The WordLang AST in CakeML almost has an SSA form[6], and the live variables have nice properties in SSA-form.

It is possible to prove that in a program in SSA form, when two registers interfere, they interfere at the definition of one of the two registers [2, lemmas 11 & 12].

The reason we filled the liveness holes to have one big interval was to have the following property: two intervals intersects iff one of them contains the starting point of the other. It turns out that for programs in SSA form, this property is true even if the lifetime has holes.

Using this property, it is possible to produce better allocation [12]

7 Conclusion

Before this internship, I had a bit of experience with Coq, but I never really went beyond some toy examples.

During this internship, I learned to use HOL4 and I saw how verification works on a large program. I implemented and verified end-to-end a new register allocation algorithm, which will enable CakeML to have shorter compilation time.

I realised that often, the hard part was not to do the actual proofs, but to find the theorems I needed to prove: for most theorems, once I found the correct invariants, the proofs were mostly straightforward.

It was a very interesting and rewarding experience. It changed the way I view maths, because I had to be absolutely rigorous since the computer doesn't accept any hand-waving arguments.

[6] "almost" meaning that ϕ-functions are already resolved: some variable might be assigned twice : at the end of both branches of a condition

References

[1] Lal George and Andrew W. Appel. Iterated register coalescing. In *POPL*, pages 208–218. ACM Press, 1996.

[2] Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Inf. Process. Lett.*, 98(4):150–155, 2006.

[3] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In *IJCAR*, volume 10900 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2018.

[4] Gary A. Kildall. A unified approach to global program optimization. In *POPL*. ACM Press, 1973.

[5] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *POPL*, pages 179–192. ACM, 2014.

[6] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.

[7] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *ESOP*, volume 9632 of *LNCS*. Springer, 2016.

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

[9] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

[10] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *ICFP*, pages 60–73. ACM, 2016.

[11] Yong Kiam Tan, Scott Owens, and Ramana Kumar. A verified type system for CakeML. In *IFL*, pages 7:1–7:12. ACM, 2015.

[12] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE*, pages 132–141. ACM, 2005.