Bachelor's Thesis

Formal Verification of Spilling Algorithms

Author:
Julian Tobias Rosemann

Supervisors:
Sigurd Schneider
Prof. Dr. Gert Smolka

Reviewers:
Prof. Dr. Gert Smolka
Prof. Dr. Sebastian Hack

submitted on
Wednesday 1st February, 2017

Saarland University
Faculty of Mathematics and Computer Science

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ____________________ ____________________

Date Signature

Abstract

Spilling is an important translation phase mandatory in every compiler back-end. Spilling translates a program with unbounded maximal live sets to a program where the variables are distributed into an unbounded set (memory) and a set bounded by an integer k (registers) at every program point. Spilling must ensure that the variables are in the register set whenever they are used. For any program there are different ways to achieve these criteria, and the choice greatly impacts performance.

In this thesis we develop a permissive correctness criterion designed to accommodate many spilling choices. If a spilling satisfies our criterion the register sets satisfy the register bound, all variables are in the registers when they are used and program equivalence is preserved.

As a case study we verify two spilling algorithms. The first is a trivial one where variables are loaded from the memory before any usage and the whole register set is spilled to the memory after every instruction. The second tries to minimize the spills and loads by remembering which variables are in the registers and in memory.

Acknowledgements

I would like to thank my family and friends for mocking me about how long it took to finish this thesis. I will just put it under the christmas tree this year.

I am much obliged for the corrections and suggestions for improvement by my proofreaders Nicklas Linz, Ferenc Beutel, Yannick Forster and my advisor Sigurd Schneider. I appreciate that Heiko Becker provided me with the LaTeX template for this thesis.

Contents

1 Introduction **1**
1.1 Related Work . . . 2
1.1.1 Global Register Allocation and Computational Complexity . . . 2
1.1.2 Algorithms for Register Allocation and Spilling . . . 3
1.1.3 Register Allocation in Verified Compilers . . . 5
1.1.4 Previous Work on Spilling in LVC . . . 6
1.2 Motivation . . . 6
1.3 Problem Statement . . . 7
1.4 Outline . . . 7

2 IL **8**
2.1 Basic Definitions . . . 8
2.2 Syntax . . . 8
2.3 Semantics . . . 8
2.4 Liveness . . . 9

3 Spilling **11**
3.1 Representation of Spilling . . . 11
3.2 Generation of the Spilled Program . . . 11

4 Correctness Criterion **14**
4.1 Configuration . . . 14
4.2 Structure of the Predicate . . . 14
4.3 Description of the Predicate . . . 15

5 Reconstruction of Liveness **18**

6 Soundness **20**
6.1 Variables in Registers . . . 20
6.2 Register Bound . . . 21
6.3 Simulation . . . 22
6.4 Soundness of the Correctness Criterion . . . 22

7 Verified Spilling Algorithms **23**
7.1 StupSpill . . . 23
7.2 SimplSpill . . . 23

8 Conclusion **26**
8.1 Future Work . . . 26

9 References **27**

1 Introduction

Spilling is an important translation phase mandatory in every compiler back-end. It deals with the problem that there is an unbounded number of variables in the source program, in contrast to only finitely many registers in any processor. At any program point the set of live variables has to be covered by the union of an unbounded set (memory) and a set bounded by an integer k (registers). We call this integer the *register bound*. Most instructions require their arguments to reside in the registers. Thus spilling must ensure that the variables used at a program point are in the registers at that program point. To guarantee that the register set satifies the register bound and that all variables are in the registers when they are used, store instructions (spills), which copy variables from the registers to the memory, and load instructions (loads), which copy variables from the memory to the registers, have to be generated.

For performance, it is crucial to execute as few load and spill instructions as possible because the processor is often at least an order of magnitude faster than the memory subsystem [1] [1]. Introducing spills and loads also increases the code size which is not desirable for performance.

As an example consider the source program given in Listing 1. On the left we have a program without spilling which needs at least three registers. The other two programs contain spills and loads to ensure execution requires only two registers. We use uppercase variable names for spill slots. Note that the decision whether x or y is spilled in the first line determines how many spills and loads are necessary in the continuation of the program.

```
                          let X := x in             let Y := y in
let z := x + y in         let z := x + y in         let z := x + y in
                                                    let X := x in
                                                    let y := Y in
if z ≥ y                  if z ≥ y                  if z ≥ y
  then                      then                      then
                              let x := X in             let x := X in
    x + z                     x + z                     x + z
  else                      else                      else
    z                         z                         z
```

Listing 1: Uppercase letters represent variables in the memory and lowercase variables represent variables in the registers

Spilling answers the question whether a variable is in the registers at a program point, but not in which register it is. The process of assigning the variables in the register set to specific registers is called *register assignment*. Spilling and register assignment together form the register allocation phase.

In the literature register allocation is often treated as a single problem, without phase separation between spilling and register assignment. *Global register allocation* is the approach to register allocation where every variable is either kept permanently in a fixed register or kept permanently in the memory [1]. In the latter case, a variable has to be loaded or spilled whenever it is used or defined, respectively. *Chaitin's graph coloring algorithm* for register allocation [10] is a global register allocation. The algorithm interleaves spilling and register allocation. Chaitin shows that in his setting register allocation is NP-complete [10].

[1] On Modern systems two or three orders of magnitude might be a more appropriate estimate (cf. http://stackoverflow.com/questions/10274355/cycles-cost-for-l1-cache-hit-vs-register-on-x86), but modern systems also include caches to avoid penalties for memory accesses.

The register allocation algorithm *Linear Scan* [18] differs from the global allocation approach and allows variables to sometimes reside in a register, and sometimes in a memory slot. This process is called *live-range splitting.* Linear Scan performs spilling and register assignment simultaneously. The use of an overapproximation of liveness makes the algorithm simple and efficient.

Both approaches are originally not using static single assignment form (SSA) [1], but in 2010, Wimmer and Franz published an SSA variant of Linear Scan [24]. We will discuss both algorithms in detail in Section 1.1.1.

In SSA-form spilling and register assignment can be decoupled. This is due to the result by Hack [13] that in SSA the maximal number of simultaneously live variables equals the maximal register pressure (i.e. the number of registers that are needed at the most critical point). This equality enables the spilling algorithm to effectively determine how many variables must be spilled at each program point. Furthermore Hack et al. give an algorithm for register assignment after spilling, that is quadratic in the number of variables and always yields an assignment [13]. We discuss the contrast in computational complexity to Chaitin's algorithm in Section 1.1.1.

It is difficult to formally state what an optimal spilling[2] is. A spilling with minimal loads and spills is not necessarily the most effective one, since it is much more important to reduce the loads and spills at frequently passed program points such as inner loops than anywhere else. The problem gets further complicated because of the different processor architectures and their properties.

In this thesis we develop a correctness criterion for spilling. In our setting, spilling algorithms annotate the source program with spilling information. The criterion is defined on this annotated program. If it satisfies the criterion, the spilling can be transformed to a program that meets the register bound, where variables are in the registers whenever they are used and where program equivalence is observed. For the verification of a spilling algorithm it suffices to prove that any possible produced spilling satisfies the criterion.

The correctness criterion is designed independent of any assumptions on optimal spilling. We try to restrict the possible applications as little as possible. Our criterion in particular supports arbitrary live range splitting, i.e., the choice where a variable resides can be made per program point. The criterion only uses set constraints which makes it convenient in application.

As a case study the predicate is used to verify two spilling algorithms. The first is a trivial one which loads before instructions and spills afterwards. The second tries to minimize the number of loads and spills by loading as late and as little as possible and only spilling variables that are overwritten and live in the program continuation.

The thesis is completely formalized in Coq. The Coq development[3] is integrated in the Linear Verified Compiler (LVC) developed by my advisor Sigurd Schneider [21]. During my work I made heavy use of the definitions and lemmas provided in LVC, in particular the liveness module.

1.1 Related Work

1.1.1 Global Register Allocation and Computational Complexity

Global Register Allocation Global register allocation can be quite effective if a good heuristic is used and sufficient registers are available. But if the register pressure is high and only a few registers are available, the produced spill code might perform poorly [7].

As an example consider Figure 1 in the use case where k registers are available and the self loops at block B and D are often traversed. No global register assignment can assign more than k

[2]We also use spilling as a term for the result of a spilling algorithm.
[3]It is available at https://www.ps.uni-saarland.de/~rosemann/lvc-spill/.

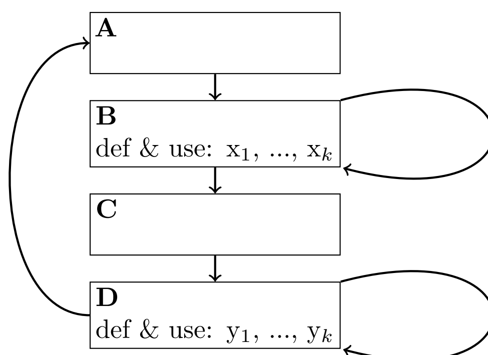

Figure 1: Pathological CFG for global register allocation

variables to registers. Consequently in block B and in block D together there will be at least k frequently executed spills and loads. By splitting the live ranges a much better spilling is possible: Load any x_i and spill any y_i in A, and load any y_i and spill any x_i in C. In this solution not a single spill or load occurs in the critical blocks B and D which greatly improves performance.

Computational Complexity Chaitin et al. proves the NP-completeness of global register allocation [10]. Bouchez and his colleagues find out that minimizing the spills and loads is NP-complete in SSA [5]. Hack shows that register assignment is in $\mathcal{O}(n^2)$ [13]. Thus in neither approach we are likely to find an optimal solution in a feasible way. But since there is an efficient algorithm for register assignment, it seems reasonable to focus on the spilling problem.

1.1.2 Algorithms for Register Allocation and Spilling

Classical Register Allocation via Graph Coloring In the following, we describe how Chaitin's algorithm [10] transforms a program such that it uses only k registers for execution.

1. create a new inference graph G
2. add a node to G for every variable in the program
3. add an edge for every pair of simultaneously live variables
4. for any copy-instruction (x `:=` y) where x and y are not connected:
   - remove the copy-instruction from the program
   - add all edges of the node y to the node x
   - remove x from the inference graph
5. if any nodes were coalesced in step 4 go to step 1
6. using a heuristic the graph is k-colored; if the coloring fails for node x:
   - insert a spill after every redefinition of x
   - insert a load before every usage of x
   - go to step 1

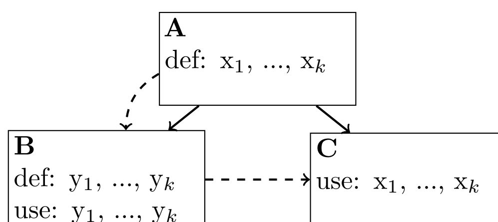

Figure 2: pathological CFG for linear scan

When the coloring finally succeeds the algorithm terminates and has modified the program in such a way that it uses only k registers. This algorithm is a global register allocation because a variable is either always in the registers or it is loaded before every use and spilled after every redefinition.

Linear Scan An alternative idea is the linear scan algorithm proposed by Poletto and Sarkar [18]. In this algorithm the basic blocks of the intermediate representation are assumed to be linearized with some ordering. The *live interval* of any variable x occurring in the linearisation is defined as the interval of program points from the first to the last position, where x is live. In particular, since live intervals argue on the basis of the linearisation which does not represent the, presumably non-linear, program flow accurately, the live interval of a variable x may contain large code fragments, where x is actually not live. The algorithm iterates over the linearisation and does the following steps at any position p:

1. Any register containing a variable whose live interval ends at p is freed.
2. If the live interval of a variable x starts at p, if possible, x is assigned to a free register. Otherwise the variable in the register, whose live interval ends the latest, is spilled.
3. If a spilled variable is used at p, it is loaded, and if necessary another variable is spilled.

This approach trades allocation quality for performance of the allocator and is therefore especially interesting for just-in-time compilers.

The linearisation tends to oversimplify the problem which in turn may result in a poor spilling. To this end consider Figure 2, where the dashed arrows indicate the linearisation of the program. Note that the problem does not arise from that specific ordering, in contrast, it is actually easy to obtain a analogous counter example for any ordering, which doesn't rely on the code inside the basic blocks.

We assume there to be k available registers. This linearisation leads to $2k$ parallel live intervals in block B, although only k variables are used there. In fact any x_i is only live in A and C, while any y_i is only live in B. Consequently, since there is no execution jumping from B to C, we could just overwrite any x_i with a y_i in B and wouldn't need to spill or load anything at all. But Linear Scan would introduce k spills and loads.

Improvements of Chaitin's Algorithm To address the disadvantages of global register allocation discussed above, Callahan and Koblenz introduce *Hierarchical Graph Coloring*. This is a variant of Chaitin's algorithm where spill choices are dependent on the local program flow and are not a global anymore [9].

The *Priority-Based Coloring* by Chow et al. considers the the problem that on some architectures with memory addressing modes Chaitin's algorithm runs into danger to overallocate registers and also develops a method to improve the locations of the spills and loads [11].

Coalescing *Coalescing* deals with the minimization of copy-instructions. In the setting of Chaitin's algorithm we have to distinguish between different definitions of coalescing because in this case coalescing influences spilling. *Aggressive Coalescing* is the problem where the number of copy-instructions are minimized, regardless of any effect this could have on the inference graph. Step 4 in Chaitin's algorithm approximates Aggressive Coalescing. It has the disadvantage that it sometimes increases the chromatic number of the inference graph, leading to overspilling (spilling unnecessarily many variables). In [8] Briggs and his colleagues introduce *Optimistic Coalescing* to stop this behaviour—which in turn has the disadvantage that it coalesces less copy-instructions. George and Appel revise it again to find some middleground between the danger of overspilling and coalescing too few copy-instructions [12]. This is called *Iterated Register Coalescing*.

Bouchez et al. prove the NP-completeness of different coalescing problems, among others Optimistic Coalescing and Chaitin's Aggressive Coalescing [4]. It is important to note that the above mentioned algorithms concerning coalescing only approximate these NP-complete problems.

In [13] Hack gives a coalescing algorithm. Since in his setting coalescing is done after spilling and register assignment, coalescing can not lead to overspilling. SSA-Maximize-Fixed-Points is an approximation for coalescing. Optimizing this problem does not necessarily imply that the number of copy-instructions is minimized. Since this approximation is still NP-complete [14] Hack uses a heuristic to approximate SSA-Maximize-Fixed-Points. An analysis with an Integer Linear Programming solver indicates that 95% of the possible optimization (of SSA-Maximize-Fixed-Points) is achieved by this heuristic [13].

Improvements of Linear Scan In [23] Traub et al. present a variant of linear scan called *Second-Chance Binpacking* and compare its efficiency and effectiveness with their own implementation of *Iterated Register Coalescing*: While the graph coloring algorithm is faster on small inputs, the linear scan algorithm is much faster on big inputs. Remarkably both algorithms yield nearly the same run-time of executables: For all tested benchmarks the ratio of the run-time (binpacking/graph coloring) is between 0.98 and 1.05 [23]. Wimmer and Franz modify linear scan for SSA-form which has simplifies and further accelerates the algorithm [24]. In this setting it is possible to integrate SSA-destruction into the resolution of the register allocator.

Spilling in SSA In SSA the maximal number of live variables is equal to the maximal register pressure [13], which allows to treat spilling and register assignment entirely separately. In this setting Braun and Hack provide an algorithm for spilling that is very sensible to the underlying program structure [6]. As already discussed above, because of the SSA-form, an optimal register assignment for the spilling can be found in quadratic time in the number of variables.

Braun's and Hack's spilling algorithm extends Belady's MIN-algorithm [2] from straight-line code to arbitrary programs. Variables are spilled less likely if the next possible usage is close. It also tries to keep spills and loads out of loops and would find the optimal spilling for the CFG depicted in Figure 1. Since it performs its computations on the program flow instead of just on a linearisation, it also delivers the perfect spilling for the second example in Figure 2. Benchmark tests show that this algorithm inserts 40% less spills and loads compared to the widely used C-compilers GCC and LLVM [6]. The correctness predicate developed in this thesis could principally be used to verify Braun's and Hack's algorithm.

1.1.3 Register Allocation in Verified Compilers

For any purpose where the correctness of the program is important enough to be formally verified, it is inevitable to have the same strong assurances on the corresponding compiler—otherwise

security ends at a *silent error* of the compiler[4].

By rigorous testing Yang et al. prove the existence of many bugs in ten commonly used C-compilers including *LLVM* and *GCC* [25]. The verified compiler *CompCert* was also tested. While their method has continuously discovered new semantic errors in any of the unverified compilers, not a single one was found in CompCert.

CompCert CompCert translates from C to machine code for PowerPC, ARM and IA32 [17] with considerable efforts of optimization: Its compiled code is about 40% faster than GCC's without optimization, 12% slower than GCC on optimization level 1 and 20% slower than GCC on optimization level 2 [15]. In the first version the graph coloring algorithm is implemented in untrusted Caml code and a verified translation validator is used to guarantee the correctness of the register assignment [17].

The spilling technique is verified and is a very simple one: Any variable that didn't get a register by the coloring is loaded before any use and spilled after any redefinition. The proof of this first approach involved about 4300 lines of Coq code [19]. Later also the register assignment was formally verified [3] which resulted in over 10000 lines of Coq code in total for register allocation [19].

Since CompCert supports IA32 as target machine from version 2 up and this architecture does not have enough registers to compensate for the verified naive spilling algorithm, the spilling algorithm had to be replaced. The new algorithm keeps track of the recently spilled or loaded variables and thus does not have to load again, if the variable is still in a temporary register. To avoid the, from an engineering point of view, expensive verification of the new spilling algorithm, translation validation was used again [19]. The verification of the validator consists of only 900 lines of Coq and is therefore a considerably concise solution.

CakeML Another verified compiler is the compiler for CakeML, whose verification is done in HOL4. This compiler shortens live ranges in a SSA-like manner (but doesn't have ϕs). Depending on the parameters given to the compiler by the user either a verified simple allocator or a verified Iterated Register Coalescing algorithm is used for register assignment [22]. This has the purpose to enable an efficient as well as an effective compile-mode. In the spirit of preventing re-engineering an inductive soundness predicate was developed and then used in both proofs. This predicate is considering the approach where the register assignment precedes the generation of spills and loads.

1.1.4 Previous Work on Spilling in LVC

In his Research Immersion Lab Patrick Klitzke began developing a correctness predicate for spilling in our setting [16], where the spilling phase precedes the register assignment phase. It has been used as a starting point and modified to the predicate stated in this thesis. Klitzke has developed examples that showed important corner cases which should be considered in the design of a permissive correctness predicate. Those examples were helpful in the construction of the predicate.

1.2 Motivation

Since an optimal solution of spilling is most likely infeasible in general and there is a research interest to find a good heuristic addressing the problem, it is convenient to have a correctness

[4] A silent error is a compilation error where code is produced that does not have the expected semantics. This leads to unexpected behaviour of the executables without a warning.

predicate to simplify the verification of any candidate algorithm. As a consequence of the disadvantages of global register allocation our correctness predicate will be permissive enough to allow for arbitrary live-range-splitting and uses precise liveness information.

Especially in a verified compiler it is useful to have such a predicate, as it is reasonable to assume that a given spilling algorithm will eventually be replaced by a better one and it would be inefficient engineering to do the whole verification proof again. Compared to the usage of verified translation validation, which also allows to exchange the spilling algorithm, this approach has the disadvantage that some new proofs are necessary. The advantages are that there are neither unjustified compiler errors nor is there compile-time overhead.

1.3 Problem Statement

We consider spilling algorithms only using insertions of spill and load calls, i.e. the output is always the same as the input aside from arbitrary spill and load calls between any two statements. The *spilled program* is the program where these spills and loads are inserted.

A spilled program s' of input program s is called *k-valid* if the following three conditions hold:

(a) all variables in s' are in a register when used

(b) at most k registers are used in s' at any point

(c) s and s' have the same behaviour

The spilled and loaded variables will be represented by sets linked to the statement before which they are to be inserted; thus the spilling algorithm can not enforce a specific ordering. The only guarantee on the ordering is that the spills will be inserted before the loads.

These are no severe restrictions: We can assume that statements containing more than k variables are split beforehand in the code generation, and the ordering of spill and load statements relatively to each other, respectively, does not really matter.

We develop an inductive correctness predicate, such that any spilling satisfying it is valid.

1.4 Outline

In Section 2 we recap the language IL and define liveness on IL. Section 3 treats the our representation of spilling and how the spilled program is generated from a spilling. The correctness predicate is presented in Section 4. In Section 5 an algorithm to compute the liveness of a spilled program is presented as well as the theorem verifying it. Our main result, the soundness of the correctness predicate, is introduced in Section 6, and the proof is sketched. In Section 7 two different spilling algorithms are presented and verified using the correctness predicate. Finally in Section 8 we conclude our results and give an outlook on possible future work.

2 IL

In our development we use the intermediate language IL as presented in [21]. The following definitions are taken from this paper.

2.1 Basic Definitions

Let $\mathbb{V}$ be a type of values and **exp** be a type of expressions. We use v as meta variable for values and e as a meta variable for expressions. We assume a type of variables $\mathcal{V}$. An environment has the type $\mathcal{V} \to \mathbb{V}_\perp$ where $\mathbb{V}_\perp$ includes $\mathbb{V}$ and $\perp$ in case there is no assignment available. Expression evaluation is a function $[\![\cdot]\!] : \mathbf{exp} \to (\mathcal{V} \to \mathbb{V}_\perp) \to \mathbb{V}_\perp$ taking an expression and an environment as arguments and returning a value or $\perp$ if the evaluation fails.

For lists we use the notation $\overline{x}$ and we lift $[\![\cdot]\!]$ to lists of expressions: $[\![e]\!]$ yields $\perp$ if at least one of the expressions in $\overline{e}$ has failed to evaluate and a list of the evaluated values otherwise.

Let $\mathrm{fv} : \mathbf{exp} \to \mathrm{set}\,\mathcal{V}$ be the usual function returning the *free variables* of an expression. Then we have for all environments V, V'

$$(\forall x \in \mathrm{fv}\, e, V\ x = V'\ x) \Rightarrow [\![e]\!]V = [\![e]\!]V'.$$

We assume a function $\beta : \mathbb{V} \to \{\mathrm{true}, \mathrm{false}\}$ to simplify the definition of semantics of the conditional.

2.2 Syntax

The syntax presented in Table 1 shows that IL is a first-order language with a tail-call restriction and mutual recursion.

We use a separate alphabet $\mathcal{F}$ for functions which enforces the first-order discipline. By convention f and g range over $\mathcal{F}$.

stmt ∋ s,t	::=	`let` x `:=` e `in` s	let statement
	|	`if` e `then` s `else` t	conditional
	|	e	return statement
	|	`fun` $\overline{f\ \overline{x}\ \texttt{:=}\ s}$ `in` t	mutual recursive function definition
	|	$f\ \overline{e}$	application

Table 1: Syntax of IL

2.3 Semantics

A *context* is a list of named definitions. A definition in a context may refer to previous definitions and itself. We use contexts like functions: Let C be a context and x be in the name space of the context. C_x returns the last definition in C corresponding to x and C^{-x} denotes the context C excluding all definitions after the last definition of x.

We call a tuple consisting of an environment, a list of variables and a statement, i.e. with type

$$(\mathcal{V} \to \mathbb{V}_\perp) * \overline{\mathcal{V}} * \mathbf{stmt}$$

a *closure*.

In a *semantic-context* function names from the alphabet $\mathcal{F}$ are mapped to closures. By convention L ranges over semantic-contexts.

$$\frac{[\![e]\!]V = v}{L \,|\, V \,|\, \texttt{let } e \texttt{ := } s \texttt{ in } t \longrightarrow L \,|\, V[x \mapsto v] \,|\, s} \text{ SemLet}$$

$$\frac{[\![e]\!]V = v \quad \beta(v) = b}{L \,|\, V \,|\, \texttt{if } e \texttt{ then } s_{\texttt{true}} \texttt{ else } s_{\texttt{false}} \longrightarrow L \,|\, V \,|\, s_b} \text{ SemIf}$$

$$\frac{}{L \,|\, V \,|\, \texttt{fun } \overline{f \ \overline{x} \texttt{ := } s} \texttt{ in } t \longrightarrow \overline{[f : (V, \overline{x}, s)]}; L \,|\, V \,|\, t} \text{ SemFun}$$

$$\frac{[\![\overline{e}]\!]V = \overline{v} \quad L_f = (V', \overline{x}, s)}{L \,|\, V \,|\, f \ \overline{e} \longrightarrow L^{-f} \,|\, V'[\overline{x} \mapsto \overline{v}] \,|\, s} \text{ SemApp}$$

Figure 3: Semantics of IL

Figure 3 shows the semantics of IL as a small-step relation $\longrightarrow$. The relation operates on *configurations* (L, V, s) where L is a semantic-context, V is an environment and $s \in \textbf{stmt}$. Often we write the configuration tuple $L \,|\, V \,|\, S$ to have the comma available as another separator. Since only tail-recursion is syntactically allowed in IL, no call stack is required.

The Coq development is done in this setting but for the sake of brevity we will only discuss single function definitions without mutual recursion in this thesis. For external operations as they are defined in [21], all results of the thesis also hold and are formally proven in Coq. Since they only have a minor effect on the proofs and results and further complicate the definitions, external operations are omitted in the written part of this thesis.

2.4 Liveness

A variable x is called *significant* in s if there is an environment V and values v, v' such that s has different behaviour in $V[x \mapsto v]$ and $V[x \mapsto v']$. Since insignificant variables don't affect the result, we do not want to keep them in registers.

As a semantic property significance is undecidable, so we use a decidable overapproximation called *liveness*. In almost all register allocation literature some variant of liveness is used, for example in register allocation by graph coloring [10] and linear scan [18]. In contrast to significance, liveness is defined solely on the syntax.

s' is a *substatement* of s if it is directly syntactically included in s. Thus let statements have one substatement, conditionals and function definition have two and return statements and function applications have none.

Intuitively speaking a variable x is live at statement s, if it is either used in an expression of s or live in any substatement of s, excluding the body in the function definition, and not overwritten in s. Additionally every defined variable is live directly succeeding the definition. The second requirement is necessary since we expect any variable assignment to be executed and thus it is written in some register, even if it is never used.

The live set corresponding to a program point contains exactly the variables that are live at that point. A statement where every program point is labeled with its live set is called a live-statement. For a let statement annotated with liveness we write

$$\texttt{let } x \texttt{ := } e \texttt{ in } s : X$$

and for any other statement the notation is analogous. Note that every substatement of a live-statement is a live-statement.

$$\frac{\mathrm{fv}\, e \subseteq X \quad X_s \setminus \{x\} \subseteq X \quad x \in X_s \quad Z \,|\, \Lambda \vdash \textbf{live}\ s : X_s}{Z \,|\, \Lambda \vdash \textbf{live}\ (\texttt{let}\ x\ \texttt{:=}\ e\ \texttt{in}\ s) : X}\ \textsc{LiveLet}$$

$$\frac{\mathrm{fv}\, e \subseteq X}{Z \,|\, \Lambda \vdash \textbf{live}\ e : X}\ \textsc{LiveReturn}$$

$$\frac{\mathrm{fv}\, e \cup X_{s_1} \cup X_{s_2} \subseteq X \quad Z \,|\, \Lambda \vdash \textbf{live}\ s_1 : X_{s_1} \quad Z \,|\, \Lambda \vdash \textbf{live}\ s_2 : X_{s_2}}{Z \,|\, \Lambda \vdash \textbf{live}\ (\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2) : X}\ \textsc{LiveIf}$$

$$\frac{\mathrm{fv}\, \overline{e} \subseteq X \quad \Lambda_f \setminus Z_f \subseteq X}{Z \,|\, \Lambda \vdash \textbf{live}\ f\ \overline{e} : X}\ \textsc{LiveApp}$$

$$\frac{X_{s_2} \subseteq X \quad \overline{x} \subseteq X_{s_1} \quad f : \overline{x}; Z \,|\, X_{s_1} :: \Lambda \vdash \textbf{live}\ s_1 : X_{s_1} \quad f : \overline{x}; Z \,|\, X_{s_2} :: \Lambda \vdash \textbf{live}\ s_2 : X_{s_2}}{Z \,|\, \Lambda \vdash \textbf{live}\ (\texttt{fun}\ f\ \overline{x}\ \texttt{:=}\ s_1\ \texttt{in}\ s_2) : X}\ \textsc{LiveFun}$$

Figure 4: Inductive definition of liveness

The variables live in the body of a function are called its *live-ins*. We define an inductive soundness predicate for a live-statement s using the notation

$$Z \,|\, \Lambda \vdash \textbf{live}\ s$$

where Z is the *parameter-context* containing for every defined function a list of its parameters. Λ represents the *live-in-context*, mapping every defined function to the set of variables that are live in the body.

Our definition of liveness depicted in Figure 4 is taken from [20] and has been adapted for use in this thesis.

For any statement containing expressions the free variables of these expressions are always in the live set. In the case of the return statement this is the only condition, and for function applications the only additional requirement is that the live-ins of the function, except the parameters, should be contained in the live set.

The live variables of a conditional include the live sets of the consequence and alternative as well as the free variables of the condition.

The live set X of a let statement contains the free variables of the expression and all variables that are live in the substatement, except the new defined one. Additionally the assigned variable is in the live set X_s of the substatement.

An interesting observation of this definition is that everything is defined using inclusion. This way it is possible to include unnecessary variables to the live set and remove them at any point. In practice we will use the *minimal* sets satisfying this rules which results in deleting (some) dead variables.

Finally in the inference rule for function definitions we require that the live set of the program continuation s_2 is contained in the live set of the function definition and that the parameters of the function are live in the body.

3 Spilling

3.1 Representation of Spilling

Spilling information always contains a set representing the spills, and a set representing the loads. By convention we use S as a meta variable for the set representing the spills and L as a meta variable for the set representing the loads. Spilling information is local, it always corresponds to a specific program point. We define spill-statements analogously to live-statements as statements annotated with spilling information at every program point.

For let statements, return statements and conditionals the corresponding spilling information is the pair (S, L) and we write:

$$\begin{aligned} \texttt{let } x \texttt{ := } e \texttt{ in } s &: (S, L) \\ \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 &: (S, L) \\ e &: (S, L) \end{aligned}$$

For a function definition, spilling information has an additional component, called the *live-in cover*. The live-in cover of a function f is a pair of sets (R_f, M_f). R_f contains all the variables the function expects to be in the registers when it is executed. The variables that are expected to be in memory are in M_f. The set of live-ins of f has to be covered by $R_f \cup M_f$. Note that R_f and M_f are not necessarily disjoint. This is important since it is sometimes convenient for a function if a variable is in the registers and in memory. To denote a spill-statement for a function definition we use the notation:

$$\texttt{fun } f \ \overline{x} \texttt{ := } s_1 \texttt{ in } s_2 : (S, L, (R_f, M_f))$$

Any non-trivial spilling algorithm has to compute the live-in cover anyway, thus it is no hard restriction to require it. The live-in cover is needed for an efficient reconstruction of liveness as it will be discussed in Section 5.

Spilling information at function application also has a third component: In our setting function applications only have variables as arguments. These variables do not have to be in the registers, instead they are also allowed to be passed through a memory slot. The main reason for this design is, that the register bound otherwise would also be a restriction on the number of arguments. As the requirements of the function definition on where the arguments are passed may differ from where the arguments are in the function application, spills and loads have to be introduced implicitly. The variables passed through a memory slot are called *slot variables*. Since the function computing the spilled program from the spill-statement needs to know which variables are slot variables, the third component of the spilling information at a function application is a set containing the slot variables. We denote a spill-statement for a function application with:

$$f \ \overline{y} : (S, L, Sl)$$

If the underlying statement of a spill-statement does not matter we write:

$$s : (S, L, _)$$

3.2 Generation of the Spilled Program

We partition the set of variables such that $\mathcal{V}_R \uplus \mathcal{V}_M = \mathcal{V}$ and require that the input program only contains variables in $\mathcal{V}_R$. Assume an injection $\text{slot} : \mathcal{V}_R \to \mathcal{V}_M$ that yields a *memory slot* for any

$$\text{doSpill}(s : (\underbrace{\{x_1, ..., x_n\}}_{\text{spills}}, \underbrace{\{y_1, ..., y_m\}}_{\text{loads}})) = \begin{array}{l}\texttt{let slot } x_1 = x_1 \texttt{ in ...} \\ \texttt{let } y_1 = \texttt{slot } Y_1 \texttt{ in ...} \\ \text{doSpillRec } s\end{array}$$

Listing 2: Specification of doSpill

variable of the input program. We use the respective uppercase letter to denote the memory slot of a (lowercase letter) variable, e.g. $X :=$ slot x.

The algorithm that computes the spilled program given a spill-statement is called doSpill and presented in Listing 2. For any variable in the spill set a spill and for any variable in the load set a load is inserted before doSpillRec is called.

doSpillRec works as follows:

- doSpill is recursively called on the substatements
- In case the argument is a let statement, a return statement or a conditional, the respective statement is returned where every substatement s' is replaced with doSpill s'.
- The parameters in a function definition have to be adjusted with respect to the live-in cover (R_f, M_f) of the function. For any parameter x:
  - if $x \in R_f \setminus M_f$ then it remains unaltered
  - if $x \in M_f \setminus R_f$ then x is replaced by slot x in the parameter list
  - if $x \in R_f \cap M_f$ then slot x is inserted after x in the parameter list

  The substatements of this modified function definition are replaced by doSpill s and this is what doSpillRec returns.
- In a function application doSpillRec replaces every slot variable y with slot y. Additionally, if the corresponding parameter in the respective function definition has been doubled, y or slot y is passed twice.

```
[a is in the registers,
 b is in memory]
fun f x y z :=     : (∅,∅,({a,x},{x,y,z}))
  if x ≥ 0         : (∅,∅)
    then

      let z' := z-1 in : (∅,{z})
      f x y z'         : (∅,∅,{x,y})
    else

      y + a            : (∅, {y})
  in f a a b           : (∅,∅,{b})
```

```
[a is in the registers,
 b is in memory]
fun f x X Y Z :=
  if x ≥ 0
    then
      let z  := Z in
      let z' := z-1 in
      f X X Y z'
    else
      let y  := Y in
      y + a
  in f a a a B
```

Listing 3: A spill-statement on the left and
the corresponding spilled program on the right

Consider Listing 3 as an example. No (explicit) spills are necessary. This is only possible because function applications are allowed to pass their arguments through memory slots. The loads of z and y are inserted before the corresponding statement.

For the function f we have $R_f = \{a, x\}$ and $M_f = \{x, y, z\}$. Thus x is passed in the registers as well as in its memory slot. For this reason it is doubled in every application of f. Note how we can pass the arguments of the same function in different ways (register or memory). The problem to connect function application and definition is treated in the next compilation step (lowering) after spilling.

We formulate our correctness predicate on spill-statements, which is convenient because spill-statements contain exactly the information we need. Validity of spilling algorithms is defined on spilled programs and thus this translation from spill-statements to the spilled program is used in the soundness of the correctness criterion.

4 Correctness Criterion

We develop a correctness predicate $\mathbf{spill_k}$ on spill-statements with the property

$$\mathbf{spill_k}\ s \Rightarrow \text{doSpill}\ s \text{ is a valid spilling,}$$

if the following preconditions hold:

- s is renamed apart, i.e. every variable is defined only once
- only variables are allowed as arguments of function calls
- all variables used or defined in s are contained in $\mathcal{V}_R$

Thus to verify a spilling algorithm it suffices to show that any produced spill-statement satisfies $\mathbf{spill_k}$. In the following k always represents the number of available registers.

4.1 Configuration

To argue about the correctness of a spilling regarding its corresponding statement we need to keep track of the state of the registers and the memory. Thus there is a set R containing the variables currently in the registers, and there is a set M representing the variables in the memory.

We will also need information about the functions that are already defined: The *parameter-context* is a context where function names are mapped to their lists of parameters. We use Z as a meta variable for parameter-contexts. In the *live-in-cover-context* functions are mapped to their live-in cover. By convention Σ ranges over live-in-cover-contexts.

Z, Σ, R and M together form the configuration. The notation for the whole judgement is:

$$Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ \ s$$

Note that the configuration is only needed to check the validity of the spillings undisclosed statements: Given a closed statement, (a statement without free variables or undefined functions) R, M, Σ and Z can be assumed to be empty. In this case we say the configuration is empty.

4.2 Structure of the Predicate

Our inductive correctness predicate is presented in Figure 6. The predicate consists of two generic rules that handle spilling and loading, and one rule for each statement type.[5] The structure of the rules assures that only SpillSpill can be used on annotations where the spill set is non-empty and only SpillLoad can be used on annotations where the spill set is empty and the load set is non-empty.

As an example consider partial derivation of a spill-statement in Figure 5, where the spilling annotation requires x to be spilled and y, z to be loaded. SpillSpill leaves us with a premise that reflects the effect of spilling the variables in the spill annotation. Note that only SpillSpill can be applied due to the non-empty spill annotation. Afterwards, SpillLoad must be used to reflect the effect of loading; again no other rule applies because only SpillLoad supports a non-empty load annotation. Finally, SpillLet, which requires spill and load annotation both to be empty and handles the let statement, can be used.

[5] In the Coq development there are actually only 5 rules: At first we stated our predicate with 5 rules where SpillSpill and SpillLoad are merged into any other rule. When we discovered that the predicate becomes clearer if they are separated, the proofs were almost finished and thus we decided to stick to the old predicate. Both formulations are equivalent.

$$\dfrac{\dots \quad \dfrac{\dots \quad \dfrac{\dots \quad C''' \vdash \mathbf{spill_k}\ s}{C'' \vdash \mathbf{spill_k}\ (\texttt{let}\ x\ \texttt{:=}\ y+z\ \texttt{in}\ s) : (\emptyset, \emptyset)}\ \textsc{SpillLet}}{C' \vdash \mathbf{spill_k}\ (\texttt{let}\ x\ \texttt{:=}\ y+z\ \texttt{in}\ s) : (\emptyset, \{y,z\})}\ \textsc{SpillLoad}}{C \vdash \mathbf{spill_k}\ (\texttt{let}\ x\ \texttt{:=}\ y+z\ \texttt{in}\ s) : (\{x\}, \{y,z\})}\ \textsc{SpillSpill}$$

Figure 5: A partial derivation

4.3 Description of the Predicate

Now consider Figure 6 where the correctness predicate is presented.

Spill and Load First examine the rule for a non-empty spill set SpillSpill: Since any variable that isn't in a register cannot be spilled, we require $S \subseteq R$. If we have a valid spilling of s without any further spills on the same configuration, except the addition of the spilled variables to the memory, then the predicate holds.

Analogously it is required that every variable in the load set is already in memory. When variables are loaded there has to be a register for each of them and since the number of registers is fixed we need to prevent loading too many variables. Thus the number of registers that are empty or can be safely overwritten (bear in mind that spills precede loads) is less or equal to the number of loads. The *kill set* K represents the variables to be overwritten and we get $R \setminus K \cup L$ as the new register state on which s should satisfy the predicate without spilling or loading anything.

Let, Return and Conditional In all three cases there is some expression in the statement and to make sure it can be evaluated, we require that any free variable in the expression is in the registers.

The Return-statement does not require anything else, i.e. it suffices if the free variables are in the registers.

In the case of the conditional the only additional demand is that the consequence and the alternative must fulfill the predicate on the same configuration.

Since the let statement introduces a new variable x which needs a register, the registers are modified in the evaluation. Similarly to SpillLoad we use a kill set to handle the case where x overwrites some other variable and, by checking the cardinality of the modified register set, the register bound is ensured. The substatement has to satisfy the predicate on this modified register set. Note that by our definition x can overwrite a variable used in the expression; this matches the ability of most architectures to use a register as an operand as well as the target of a computation.

Function Application and Definition In the case of the application we have three inclusions as conditions: To evaluate the function the live-ins of the function that are not parameters should be where they are expected by the function. This is guaranteed by the first two requirements. Finally all arguments have to be either in the registers or in memory.

Considering function definitions, we require that the function does not expect more variables to be in the registers than possible. The second condition ensures that the variables expected to be in the registers or in memory, together form a live set of the body. Finally we need to check whether the predicate holds for the function body on its register set R_f and memory set M_f, and whether the program continuation also satisfies it on unchanged register and memory sets. In both cases the parameters of the function and its live-in cover are added to the configuration.

$$\frac{\begin{array}{c} S \subseteq R \\ Z \,|\, \Sigma \,|\, R \,|\, M \cup S \vdash \mathbf{spill_k}\ \ s : (\emptyset, L, _) \end{array}}{Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ \ s : (S, L, _)}\ \textsc{SpillSpill}$$

$$\frac{\begin{array}{c} L \subseteq M \\ |R \setminus K \cup L| \leq k \\ Z \,|\, \Sigma \,|\, R \setminus K \cup L \,|\, M \vdash \mathbf{spill_k}\ \ s : (\emptyset, \emptyset, _) \end{array}}{Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ \ s : (\emptyset, L, _)}\ \textsc{SpillLoad}$$

$$\frac{\mathrm{fv}\, e \subseteq R}{Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ \ e : (\emptyset, \emptyset)}\ \textsc{SpillReturn}$$

$$\frac{\begin{array}{c} \mathrm{fv}\, e \subseteq R \\ Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ \ s_1 \\ Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ \ s_2 \end{array}}{Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ (\mathtt{if}\ \ e\ \ \mathtt{then}\ \ s_1\ \ \mathtt{else}\ \ s_2) : (\emptyset, \emptyset)}\ \textsc{SpillIf}$$

$$\frac{\begin{array}{c} \mathrm{fv}\, e \subseteq R \\ |R \setminus K_x \cup \{x\}| \leq k \\ Z \,|\, \Sigma \,|\, R \setminus K_x \cup \{x\} \,|\, M \vdash \mathbf{spill_k}\ \ s \end{array}}{Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ (\mathtt{let}\ \ x\ \ \mathtt{:=}\ \ e\ \ \mathtt{in}\ \ s) : (\emptyset, \emptyset)}\ \textsc{SpillLet}$$

$$\frac{\begin{array}{c} (R_f, M_f) := \Sigma_f \\ R_f \setminus Z_f \subseteq R \\ M_f \setminus Z_f \subseteq M \\ \mathrm{fv}\, \overline{y} \subseteq R \cup M \end{array}}{Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ (f\ \ \overline{y}) : (\emptyset, \emptyset, (R, M))}\ \textsc{SpillApp}$$

$$\frac{\begin{array}{c} |R_f| \leq k \\ Z \,|\, \mathrm{merge}\, \Sigma \vdash \mathbf{live}\ s_1 : R_f \cup M_f \\ f : \overline{x}; Z \,|\, f : (R_f, M_f); \Sigma \,|\, R_f \,|\, M_f \vdash \mathbf{spill_k}\ \ s_1 \\ f : \overline{x}; Z \,|\, f : (R_f, M_f); \Sigma \,|\, R\ \,|\, M\ \ \vdash \mathbf{spill_k}\ \ s_2 \end{array}}{Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ (\mathtt{fun}\ \ f\ \ \overline{x}\ \ \mathtt{:=}\ \ s_1\ \ \mathtt{in}\ \ s_2) : (\emptyset, \emptyset, (R_f, M_f))}\ \textsc{SpillFun}$$

$$\begin{aligned} \mathrm{merge}\ \ \mathrm{nil} &:= \mathrm{nil} \\ \mathrm{merge}\ (R_f, M_f); \Sigma &:= R_f \cup M_f; \mathrm{merge}\, \Sigma \end{aligned}$$

Figure 6: Inductive correctness predicate $\mathbf{spill_k}$

In the Coq development the liveness information is, in contrast to this definition, also included in the configuration and is required to satisfy the liveness predicate at any program point. This has the computational advantage that liveness has only to be computed only once.

5 Reconstruction of Liveness

Recall that we are considering two disjoint name spaces $\mathcal{V}_R$ and $\mathcal{V}_M$ for variables in the registers or in memory as introduced in Section 3.2. To ensure that at most k registers are used in the spilled program we need to know how many of the variables in $\mathcal{V}_R$ are live in the spilled program.

A possible approach to determine the liveness would be applying the fixpoint algorithm for arbitrary programs (which is already implemented in LVC [20]) on the spilled program. But since a proof on that fixpoint algorithms would need to deal with the fixed-point iteration, we use a simpler and more efficient algorithm called spilledLv that does not require fixed-point iteration. This is possible, because spilling only changes the live sets by renaming some of the variables to memory slots, and introduces new spill and load statements. Since spill and load statements are let statements their live sets can be determined in one iteration on the program.

spilledLv is a recursively descending algorithm with the following signature:

$$\begin{aligned} \text{spilledLv}\ &(Z : \text{parameter-context}) \\ &(\Sigma : \text{live-in-cover-context}) \\ &(G : \text{set}\ \mathcal{V}) \\ &(s : \text{spill-statement}) \\ :\ &\text{live-statement} \end{aligned}$$

A spill-statement s contains the statement itself as well as the spilling information and in particular the live-in cover if it is a function definition. The live-in cover is sufficient to reconstruct full liveness information, because it tells us the liveness information of the body of the function. Determine this information would require fixed-point iteration. The algorithm reconstructs liveness information between function definitions by a single backwards pass.

The argument G is always added to the live set. It is used to ensure the conditions $x \in X_s$ in LiveLet and $\overline{x} \subseteq X_s$ in LiveFun:

- if s is a let statement (`let` x `:=` e `in` s) then $G = \{x\}$ in the recursive subcall
- if s is a function definition (`fun` f $\overline{x}$ `:=` s_1 `in` s_2)
  - then $G = \overline{x}$ in the recursive subcall for the function body
  - and $G = \emptyset$ in the subcall for the program continuation.
- in any other case $G = \emptyset$

This guarantees that the live set of the live-statement directly succeeding a let statement or the body of a function definition contains the newly defined variable or all parameters, respectively.

The definition of the algorithm is technical, and we refer the interested reader to the Coq development for details. We only give an informal definition here. The algorithm proceeds recursively as follows: First doSpill is used to generate the spilled program of an input s. Then this spilled program is recursively annotated with its live sets:

- A let statement (`let` x `:=` e `in` s') is annotated with

  $$G \cup \text{fv}\, e \cup \text{spilledLv}\, Z\ \Sigma\ \{x\}\ s' \setminus \{x\}.$$

  Note that the loads and spills inserted by doSpill are let statements.
- A return-statement (e) is annotated with $G \cup \text{fv}\, e$.

- A conditional (`if` e `then` s_1 `else` s_2) is annotated with:

$$G \cup \text{fv}\, e \cup \text{spilledLv}\; Z\; \Sigma\; \emptyset\; s_1 \cup \text{spilledLv}\; Z\; \Sigma\; \emptyset\; s_2$$

- If s is a function application ($f\ \overline{y}$) we gain R_f and M_f from Σ and the parameters $\overline{z}$ from Z. Any variable in M_f has to be mapped to its slot to be distinguished from the case where it is in the registers. Let $\text{fv}\,\overline{y}$ be the free variables for each argument. s is annotated with

$$G \cup \text{fv}\,\overline{y'} \cup (R_f \cup \text{slot}\ M_f) \setminus \overline{z}$$

  where y is modified by doSpill to y' as discussed in Section 3.2.

- If s is a function definition (`fun` f $\overline{x}$ `:=` s_1 `in` s_2) it is annotated with the liveness of s_2 unified with G. In this case the live-in cover from the spilling information has to be added to Σ and the parameters $\overline{x}$ have to be added to Z in both recursive subcalls.

Theorem 1. *Let s be a spill-statement, Z a list of variables, Σ a live-in-cover-context and R, M and G sets of variables. We require:*

- $Z \cup R \cup M \subseteq \mathcal{V}_R$
- *all variables in s are in $\mathcal{V}_R$*
- *applications have only variables as arguments*
- *s is renamed apart*

Then

$$Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ \ s \Rightarrow Z \,|\, \text{merge}\, \Sigma \vdash \mathbf{live}\ \ \text{spilledLv}\; Z\; \Sigma\; G\; s.$$

Proof. By induction on $\mathbf{spill_k}\, s$. The formal proof in Coq is quite involved, requires several lemmas and many operations on sets. ☐

6 Soundness

We want to prove that the spilled program generated by translating a spill-statement s using doSpill is valid if s satisfies $\mathbf{spill_k}$. For this we need three lemmas. Each of them handles one of the three conditions of validity, respectively.

6.1 Variables in Registers

Figure 7 depicts the predicate guaranteeing that every variable is in a register whenever it is used. It consists of an inference rule for each statement type and an additional rule for load instructions from the memory. All substatements always have to fulfill the predicate.

The only admitted load instructions is a let statement where a register variable x is assigned to a variable from the memory. For every other let statement we require that the expression on the right hand side contains only variables from the registers. We have the same requirement in the case of the return statement and the conditional.

Function definitions don't need any additional conditions, because they don't operate on the registers. Since we allow arguments to reside in the memory any function application statement satisfies the predicate.

A program is called *renamed apart* if all variable and function definitions use different names, for further reference see [20].

Lemma 1. *Let s be a spill-statement, Z a list of variables, Σ a live-in-cover-context and R, M sets of variables. We require:*

- $Z \cup R \cup M \subseteq \mathcal{V}_R$
- *all variables in s are in $\mathcal{V}_R$*
- *s is renamed apart*

Then:

$$Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s \Rightarrow \mathbf{vir}\ \texttt{doSpill}\ s$$

Proof. By induction on $\mathbf{spill_k}\ s$. We have 7 cases:

- SpillLoad: Use the constructor ViRLoad and derive the conditions a from the definition of doSpill and the inductive hypothesis.
- SpillSpill: Use the constructor ViRLet, the definition of doSpill and the inductive hypothesis.
- SpillApp: There is nothing to prove.
- SpillLet, SpillIf, SpillReturn, SpillFun: Use the appropriate constructor, then the assumptions of the $\mathbf{spill_k}$-constructor and the inductive hypothesis(es) suffice to finish the proof.

□

$$\frac{x \in \mathcal{V}_R \quad y \in \mathcal{V}_M \quad \mathbf{vir}\, s}{\mathbf{vir}\, \texttt{let } x \texttt{ := } y \texttt{ in } s} \text{ VIRLOAD}$$

$$\frac{\text{fv}\, e \subseteq \mathcal{V}_R \quad \mathbf{vir}\, s}{\mathbf{vir}\, \texttt{let } x \texttt{ := } e \texttt{ in } s} \text{ VIRLET}$$

$$\frac{\text{fv}\, e \subseteq \mathcal{V}_R}{\mathbf{vir}\, e} \text{ VIRRETURN}$$

$$\frac{\text{fv}\, e \subseteq \mathcal{V}_R \quad \mathbf{vir}\, s \quad \mathbf{vir}\, t}{\mathbf{vir}\, \texttt{if } e \texttt{ then } s \texttt{ else } t} \text{ VIRIF}$$

$$\frac{}{\mathbf{vir}\, f\ \overline{y}} \text{ VIRAPP}$$

$$\frac{\mathbf{vir}\, s \quad \mathbf{vir}\, t}{\mathbf{vir}\, \texttt{fun } f\ \overline{x} \texttt{ := } s \texttt{ in } t} \text{ VIRFUN}$$

Figure 7: Predicate vir

6.2 Register Bound

To ensure that at most k registers are used in the spilled program we check at every program point whether the number of variables that are live and in the register name space $\mathcal{V}_R$ is less or equal to k.

Lemma 2. *Let s be a spill-statement, Z a list of variables, Σ a live-in-cover-context and R, M, G sets of variables. We require:*

- $|R| \leq k$
- $Z \cup R \cup M \subseteq \mathcal{V}_R$
- *all variables in s are in $\mathcal{V}_R$*
- *applications have only variables as arguments*
- *s is renamed apart*
- $\mathcal{V}_R \cap G \subseteq R$

Let $s' :=$ spilledLv Σ Z G s and let X be an arbitrary live set in s'. Then

$$Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ \ s \Rightarrow |\mathcal{V}_R \cap X| \leq k.$$

Proof. By induction on s. The proof is involving and requires several lemmas. In particular, it was difficult to show that the addition of G to every live set does not violate the register bound. For this a lemma was needed with an invariant on G, which was difficult to find. For further information consider the Coq development. ☐

6.3 Simulation

The next lemma states the program equivalence between original and spilled program.

Lemma 3. *Let s be a spill-statement where all variables are renamed apart and every argument of a function application is a variable.*

Then s and doSpill *s have the same behaviour.*

Proof. By induction on s. □

This result was not part of my work, it was proven by Sigurd Schneider. He made use of the definitions and lemmas I had developed for the proofs of the other claims. For completion it is included in this thesis.

6.4 Soundness of the Correctness Criterion

To state the main theorem we formalize the definition of *valid*:

Let s be a spill-statement and let s' be a live-statement with $Z \,|\, \Lambda \vdash \textbf{live}\ s'$ where Z and Λ are a parameter-context or a live-in-context, respectively. s' is a valid spilled program of s on Z and Λ if

(i) $\textbf{vir}\, s'$

(ii) $\forall\ X$ arbitrary live set in $s', |\mathcal{V}_R \cap X| \leq k$

(iii) s and s' have the same behaviour.

Now we can formulate the soundness of $\mathbf{spill_k}$:

Theorem 2. *Let s be a spill-statement, Z a list of variables, Σ a live-in-cover-context and R, M and G sets of variables. We require:*

- $|R| \leq k$
- $Z \cup R \cup M \subseteq \mathcal{V}_R$
- *all variables in s are in $\mathcal{V}_R$*
- *applications have only variables as arguments*
- *s is renamed apart*
- $\mathcal{V}_R \cap G \subseteq R$

Then

$$Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k} \ \Rightarrow \text{doSpill}\ s \textit{ is a valid spilled statement of } s.$$

Proof. For the second condition recall that Theorem 1 states that we can get the liveness of the spilled program using spilledLv. Then all conditions of validity can be directly derived from Lemma 1, Lemma 2 and Lemma 3. □

7 Verified Spilling Algorithms

In our setting spilling algorithms of closed programs have the signature:

$$\text{live-statement} \rightarrow \text{spill-statement}$$

The following two algorithms are both implemented and verified in Coq, using the correctness predicate. In the proof development they are formulated in a way that is not restricted to the case that the input is a closed program.

7.1 StupSpill

The naive spilling algorithm stupSpill is described by two simple rules applied recursively to every statement:

- Any variable used in the expression of the statement is loaded before the statement. Doing so overwrites arbitrary variables in the registers.
- Any variable that is in the registers after the execution of the statement is spilled afterwards.

Theorem 3. *Let s be a live-statement, Z a parameter-context, Σ a live-in-cover-context, R, M sets of variables. We require:*

- *all variables in s are in $\mathcal{V}_R$*
- *every expression in s contains at most k different variables*
- *for any live set X in s: $X \subseteq R \cup M$*
- *the first component in Σ_f is empty for any f*

Then:

$$Z \,|\, \text{merge}\, \Sigma \vdash \textbf{live}\;\; s \Rightarrow Z \,|\, \Sigma \,|\, R \,|\, M \vdash \textbf{spill}_\mathbf{k}\;\; \text{stupSpill}\, s.$$

Proof. Induction on s. Proving all the conditions of the corresponding constructor in each case is easy and the whole proof has less than 100 lines of Coq code. □

7.2 SimplSpill

The spilling algorithm simplSpill follows three key ideas:

- variables are loaded as late as possible
- arbitrary variables that are not used in the statement are chosen to be overwritten in the registers until there are enough free registers for loading
- of the overwritten variables only those are spilled, that are live in a substatement, which is not the function body

Let take : $\mathbb{N} \rightarrow \text{set}\, \mathcal{V}$ be a function that takes a natural number n and a set X as arguments and returns an arbitrary subset of X of size n, if possible. If $|X| \leq n$ then let take $X := X$. Given a live-statement the operator lv yields the topmost liveness annotation.

Listing 8 shows the algorithm simplSpill. The parameter-context Z and the live-in-cover-context Σ are represented as lists.

```
 1  Fixpoint simplSpill (k : ℕ)
 2                      (Z: list (list 𝒱)
 3                      (Σ : list ({𝒱} * {𝒱}))
 4                      (R M : {𝒱})
 5                      (s : live-statement) {struct s}
 6    : spill-statement :=
 7    match s with
 8    | lvLet x e s _
 9      => let L    := fv e \ R in
10         let K    := take |L| (R \ fv e) in
11         let Rₑ   := R \ K ∪ L in
12         let Kₓ   := if |Rₑ| ≥ k then take 1 Rₑ else ∅ in
13         let S    := lv s ∩ (K ∪ Kₓ) \ M in
14         let Rₛ   := {x; Rₑ \ Kₓ} in
15         spLet x e (simplSpill k Z Σ Rₛ (M ∪ S) s) (S,L)
16
17    | lvReturn e _
18      => spReturn e (∅, fv e \ R)
19
20    | lvIf e s₁ s₂ _
21      => let L  := fv e \ R in
22         let K  := take |L| (R \ fv e) in
23         let Rₑ := R \ K ∪ L in
24         let S  := (lv s₁ ∪ lv s₂) ∩ K in
25         spIf e (simplSpill k Z Σ Rₑ (M ∪ S) s₁)
26                (simplSpill k Z Σ Rₑ (M ∪ S) s₂) (S,L)
27
28    | lvApp f ȳ _
29      => let (R_f, M_f) := Σ_f in
30         let L        := R_f \ R \ Z_f in
31         let K        := take |L| (R \ R_f)
32         let S        := M_f \ M \ Z_f ∪ ((fv ȳ \ M) ∩ K) in
33         spApp f ȳ (S,L,M ∪ S)
34
35    | lvFun f x̄ s₁ s₂ _
36      => let R_f := take k (lv s₁) in
37         let M_f := lv s₁ \ R_f in
38         spFun f x̄ (simplSpill k (x̄::Z) ((R_f,M_f)::Σ) R_f M_f s₁)
39                   (simplSpill k (x̄::Z) ((R_f,M_f)::Σ) R M s₂)
40  .
```

Figure 8: simplSpill

Theorem 4. *Let s be a live-statement, Z a parameter-context, R, M sets of variables. Let Σ be a live-in-cover-context such that for all $(R_f, M_f) := \Sigma_f$ we have $|R_f| \leq k$. We require:*

- *all variables in s are in $\mathcal{V}_R$*
- *every expression in s contains at most k different variables*
- *for any live set X in s: $X \subseteq R \cup M$*
- *every function in s is defined in Σ and Z*

Then:

$$Z \,|\, \text{merge}\, \Sigma \vdash \textbf{live}\ s \Rightarrow \Sigma \,|\, Z \,|\, R \,|\, M \vdash \textbf{spill}_\mathbf{k}\ \text{simplSpill}\, s.$$

Proof. Induction on $Z \,|\, \text{merge}\, \Sigma \vdash \textbf{live}\ s$. There are 5 cases. We apply SpillSpill in every case except function definition and return statement, and then SpillLoad in every case except function definition. We have to show $S \subseteq R$, $L \subseteq M$ and $|R \setminus K \cup L| \leq k$. With some set operations and the prerequisites these conditions are easily derived in every case. It remains to show:

$$Z \,|\, \Sigma \,|\, R \setminus K \cup L \,|\, M \cup S \vdash \textbf{spill}_\mathbf{k}\ s : (\emptyset, \emptyset, _)$$

- if s is an assignment:
  - By definition of L, K and K_x we get $\text{fv}\, e \subseteq R \setminus K \cup L$.
  - Using case analysis on whether $|R \setminus K \cup L| \geq k$ we get

    $$|(R \setminus K \cup L) \setminus K_x \cup \{x\}| \leq k.$$

  - The last condition is derived by the inductive hypothesis, where we also have to verify our invariants.
- if s is a conditional: Analogously to the first case.
- if s is a return statement: Analogously to the first case.
- if s is a function application:
  - By definition of L and K we get $R_f \subseteq R \cup Z_f$.
  - By definition of S we get $M_f \subseteq S \subseteq M \cup S$.
  - $\text{fv}\, \overline{y} \subseteq R \setminus K \cup L \cup M \cup S$ is easily derived using the definition of M_f and S.
- if s is a function definition:
  - $|R_f| \leq k$ obviously holds by definition of take.
  - The induction yields $Z \,|\, \text{merge}\, \Sigma \vdash \textbf{live}\ s_1$ and since $R_f \cup M_f$ is the live set of s_1 we can conclude

    $$Z \,|\, \text{merge}\, \Sigma \vdash \textbf{live}\ s : R_f \cup M_f.$$

  - The last two conditions are derived by inductive hypothesis.

□

8 Conclusion

We have discussed different approaches in register allocation and argued why it is promising to follow the SSA-based register allocation approach of Hack et al. [13] to first generate the spill-code and then do the register assignment. In particular we presented simple situations where the arbitrary live-range-splitting combined with precise liveness information is necessary to obtain good results.

Motivated by this we developed and formally proved a correctness predicate for spilling algorithms that permits arbitrary live-range-splitting. To the best of our knowledge, it is the first formally proven correctness predicate for spilling that supports arbitrary live-range splitting.

It turns out that the conditions of the predicate are mostly set inclusions and bounds on set cardinalities. The predicate consists of seven simple inference rules. These are stated in a way that there is always only one applicable rule, which makes the proof easy and allows for automation.

Finally we used the predicate to verify two spilling algorithms. The first is a very naive algorithm, and the proof is easy, as it should be (less than 100 lines of Coq code). The second algorithm simplSpill is also simple: It tries to minimize loads and spills, but does not try to avoid spilling/loading within loops. The verification of simplSpill takes less than 500 lines of Coq code in total. Most of the proof script is required to show simple set inclusions which could be discharged by solvers like SPASS in fractions of a second. Once support for finite set theory arrives in Coq, proving our spilling predicate will be mostly automateable.

The Coq development has more than 5000 lines of code. Roughly 60% of it are proof scripts.

8.1 Future Work

The focus of this thesis was on the development and verification of a framework simplify the integration of new spilling algorithms. Naturally, the spilling algorithms we verified as examples are quite basic. The next step would be using the predicate in the verification of a more sophisticated spilling algorithm—for example the algorithm by Braun and Hack discussed in [6].

We are also interested in generalizing the result. Since our correctness predicate does not care which variables are spilled (as long as enough variables are spilled) we want to verify a parametric spilling algorithm. The parameter of the algorithm is an ordering of variables from an untrusted heuristic that suggests which variables are chosen to be spilled.

The correctness of this spilling algorithm does not rely on the heuristic, which means that the heuristic can be changed without changing the proofs. This has the advantage that one could implement different heuristics and test their effectiveness on benchmarks, and without having to verify any of them it is guaranteed that the spilling algorithm is correct.

Another idea would be to modify simplSpill in such a way that it takes a spill-statement as input and returns a valid spill-statement—independent of the validity of the input. It would only add/remove spill and load instructions if necessary and it would return the input unchanged if it is already valid. Executing this modified simplSpill after an output of an unproven spilling algorithm would again lead to a verified spilling algorithm.

This idea is similar to translation validation, but it has the advantage that if the untrusted algorithm produces an invalid spilling, instead of throwing a compiler error the spilling is repaired.

9 References

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] Laszlo A. Belady. “A study of replacement algorithms for a virtual-storage computer”. In: *IBM Systems journal* 5.2 (1966).

[3] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. “Formal Verification of Coalescing Graph-Coloring Register Allocation”. In: *Programming Languages and Systems, 19th European Symposium on Programming*. Vol. 6012. LNCS. Paphos, Cyprus, Mar. 20–28, 2010.

[4] Florent Bouchez, Alain Darte, and Fabrice Rastello. “On the complexity of register coalescing”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2007.

[5] Florent Bouchez, Alain Darte, and Fabrice Rastello. “On the Complexity of Spill Everywhere Under SSA Form”. In: *SIGPLAN Not.* 42.7 (June 2007). ISSN: 0362-1340.

[6] Matthias Braun and Sebastian Hack. “Register spilling and live-range splitting for SSA-form programs”. In: *International Conference on Compiler Construction*. Springer. 2009.

[7] Preston Briggs. “Register allocation via graph coloring”. PhD thesis. Rice University, 1992.

[8] Preston Briggs, Keith D Cooper, and Linda Torczon. “Improvements to graph coloring register allocation”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994).

[9] David Callahan and Brian Koblenz. “Register allocation via hierarchical graph coloring”. In: *ACM SIGPLAN Notices*. Vol. 26. 6. ACM. 1991.

[10] Gregory J Chaitin et al. “Register allocation via coloring”. In: *Computer languages* 6.1 (1981).

[11] Frederick Chow and John Hennessy. “Register allocation by priority-based coloring”. In: *ACM Sigplan Notices* 19.6 (1984).

[12] Lal George and Andrew W Appel. “Iterated register coalescing”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.3 (1996).

[13] Sebastian Hack, Daniel Grund, and Gerhard Goos. “Register Allocation for Programs in SSA-Form”. In: *Compiler Construction, 15th International Conference. Proceedings*. Vol. 3923. LNCS. Vienna, Austria, Mar. 30–31, 2006.

[14] Sebastian Hack, Daniel Grund, and Gerhard Goos. “Towards Register Allocation for Programs in SSA-Form”. In: 2005-27 (Sept. 2005).

[15] Daniel Kästner et al. “Closing the Gap – The Formally Verified Optimizing Compiler CompCert”. In: *SSS’17: Safety-critical Systems Symposium 2017*. Proceedings of the Twenty-fifth Safety-Critical Systems Symposium. Bristol, United Kingdom, Feb. 2017.

[16] Patrick Klitzke. “Verification of Spilling Algorithms in Coq”. 2015.

[17] Xavier Leroy. “A Formally Verified Compiler Back-end”. In: *JAR* 43.4 (2009).

[18] Massimiliano Poletto and Vivek Sarkar. “Linear scan register allocation”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.5 (1999).

[19] Silvain Rideau and Xavier Leroy. “Validating register allocation and spilling”. In: *Compiler Construction (CC 2010)*. Vol. 6011. LNCS. 2010.

[20] Sigurd Schneider, Gert Smolka, and Sebastian Hack. “A Linear First-Order Functional Intermediate Language for Verified Compilers”. In: *ITP*. Vol. 9236. LNCS. Nanjing, China, Aug. 24–27, 2015.

[21] Sigurd Schneider, Gert Smolka, and Sebastian Hack. “An Inductive Proof Method for Simulation-based Compiler Correctness”. In: *CoRR* abs/1611.09606 (2016).

[22] Yong Kiam Tan et al. “A New Verified Compiler Backend for CakeML”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan, 2016.

[23] Omri Traub, Glenn Holloway, and Michael D. Smith. “Quality and Speed in Linear-scan Register Allocation”. In: vol. 33. 5. New York, NY, USA, May 1998.

[24] Christian Wimmer and Michael Franz. “Linear scan register allocation on SSA form”. In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM. 2010.

[25] Xuejun Yang et al. “Finding and Understanding Bugs in C Compilers”. In: *SIGPLAN Not.* 46.6 (June 2011).