Verified Spilling and Translation Validation with Repair

Julian Rosemann✉, Sigurd Schneider(orcid.org/0000-0003-1948-0596), Sebastian Hack

Saarland University, Saarland Informatics Campus
✉ rosemann@stud.uni-saarland.de

Abstract. Spilling is a mandatory translation phase in every compiler back-end. It decides whether and where a value is stored in a register or in memory and has therefore a significant impact on performance. In this paper, we study spilling in the setting of a verified compiler with a term-based intermediate representation that provides an alternative way to realize SSA. We devise a permissive correctness criterion to accommodate many SSA-based spilling algorithms and prove the criterion sound. As case study, we verify two basic spilling algorithms. Finally, we show that our criterion is decidable by deriving a translation validator that repairs spilling information if necessary. We show that the validator always produces a valid spilling, and that the validator does not alter valid spilling information. Our results are formalized in Coq as part of the LVC compiler project.

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, but only finitely many registers in any processor. After successful spilling, the set of live variables at every program point is covered by the union of an unbounded set (the memory) and a set bounded by an integer k (the registers). We call k the *register bound*. Spilling must ensure the value of a variable resides in a register whenever an instruction uses it. For this purpose, spilling inserts store instructions (spills), which copy the values of variables from the registers to the memory, and load instructions (loads), which copy the values of variables from the memory to the registers. For performance, it is crucial that few load and spill instructions are executed, because register access is at least an order of magnitude faster than memory access. 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. The program on the left needs at least three registers. The middle and right programs are different spilled forms of the left program, and each requires only two registers. 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.

Spilling determines whether a variable resides in a register at a program point, but does not determine the register. *Register assignment* assigns variables

```
                        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. A program (left) and two spilled forms of the same program. Lowercase variables denote registers, uppercase variables denote spill slots.

to specific registers. 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. In this work, we leverage that the number of simultaneously live variables equals the register pressure to decouple spilling from register assignment. This is possible for static single assignment (SSA) programs [7], and programs in the intermediate language IL [11] used in the verified compiler LVC[1]. IL realizes SSA in a term-based setting by interpreting variable definition as binding with scope.

We develop a small framework for verification of spilling based on an inductively defined correctness criterion. The criterion is formulated relative to spilling information (i.e. which variable is spilled/loaded where) and liveness information. If spilling information satisfies the criterion, it can be used to obtain a program that meets the register bound, and in which variables are in the registers whenever they are used, and which is equivalent to the original program. To verify a spilling algorithm, it suffices to prove that every produced spilling information satisfies the criterion.

It is difficult to formally state what optimal spilling is. Minimizing loads and spills is not necessarily the most effective approach, because reducing the loads and spills at frequently passed program points is more important than anywhere else. Properties of different processor architectures further complicate the problem. Our correctness criterion is independent of assumptions about optimality. We restrict the spilling choices as little as possible. Our criterion in particular supports *arbitrary live range splitting*, i.e., the choice whether a variable should reside in memory or registers is made per program point. This is mandatory to produce spillings with acceptable performance [4]. A value may also reside in a register and in memory simultaneously.

As a case study, we use the predicate to verify three 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

[1] https://www.ps.uni-saarland.de/~sdschn/LVC

as little as possible, and only spilling variables that are overwritten and live in the program continuation.

The third spilling algorithm is similar to a translation validator that takes spilling information from an untrusted source as input. Instead of only validating the spilling information, our algorithm corrects mistakes in the untrusted spilling. We formally show that our algorithm transforms any spilling information (correct or not) to an ultimately correct spilling, and that spilling information that already satisfies our criterion remains unchanged. Interestingly, the algorithm is not much more complicated than a translation validator. To our knowledge, this is the first algorithm of its kind. This approach unites the flexibility of translation validation with the guarantees of full verification.

Our results are formalized in Coq and part of the Linear Verified Compiler (LVC). The development is available online[2]. In summary, this paper makes the following contributions:

- A modular framework for correctness of spilling for term-based SSA
- Verification of two simple spilling algorithms for term-based SSA
- A translation validator for spilling that not only accepts valid spillings, but also repairs incorrect spilling from an external untrusted source.

Outline. The paper is organized as follows. Section 3 contains the semantics of the language IL, and Section 4 discusses liveness information. Section 5 discusses the representation of spilling information and the generation of the spilled program. In Section 6, we define the correctness criterion for spilling and in Section 7 we prove its soundness. Section 8 contains two case studies. Section 9 describes our translation validator with repair. Section 10 concludes.

2 Related Work

Global register allocation was pioneered by Chaitin [5]. Since Chaitin's initial work, there have been several improvements to graph coloring that mostly concentrated on coalescing, i.e. the removal of copy instructions. Most graph coloring approaches decide for every variable globally whether it resides in a register (and if so, in which) or a spill slot. Especially, graph coloring allocators do not attempt to split live ranges sophistically but rather transfer spilled variables from/to memory upon each access. This gives a simple spilling scheme that is also amenable to formal verification (see below). However, in practice the spilling quality of these algorithms is not sufficient to achieve acceptable performance [4].
Linear Scan by Poletto and Sarkar [9] is the basis for many practically popular approaches to register allocation. Linear scan splits live ranges, i.e. it allows a variable to be in a register at one program point and in memory at another. For performance reasons, linear scan over-approximates the live ranges of variables by linearizing control flow, hence the name. Linear scan intertwines spilling and register assignment.

[2] https://www.ps.uni-saarland.de/~rosemann/lvc-spill

Static Single Assignment (SSA) allows to decouple spilling and register assignment. In SSA the number of simultaneously live variables equals the register pressure [7]. SSA-based spilling algorithms can hence effectively determine how many variables must be spilled at each program point without knowing the register assignment. Braun and Hack [4] provide an SSA-based spilling algorithm that is very sensitive to the underlying program structure.

Computational Complexity Chaitin proves NP-completeness of global register allocation [5]. Bouchez et al. show that minimizing spills and loads is NP-complete in SSA [3]. Bouchez also shows NP-completeness of different coalescing problems, i.e. minimizing the number of copies/swaps required to implement SSA's ϕ-functions after the register allocation phase.

CompCert Register allocation in the first version of CompCert used a translation validated graph coloring algorithm implemented in OCaml [8]. Spilling is verified and very simple: Variables not in a register are loaded before use and spilled after redefinition. Later Blazy et al. [2] fully verified Appel's [6] iterated register coalescing (IRC) approach, which includes spilling. Being a graph coloring technique, this algorithm suffers from the same drawbacks concerning spilling that we discussed above. Hence, especially for machines with few registers (such as IA32), the code quality is hardly acceptable. Instead of changing the fully verified spiller, which would have been a tremendous effort, Rideau et al. [10] developed a new translation validated algorithm for register allocation and spilling. The new spilling algorithm tracks recently spilled and loaded variables and thus avoids loading if the variable is still in a temporary register.

In contrast to the verified register allocation by Blazy et al., the second spilling algorithm we verify as case study splits live ranges. The algorithm follows a strategy similar to the translation validated algorithm of Rideau et al, is verified, but does not support overlapping registers yet. There is a project that aims to bring SSA to CompCert [1], but SSA-based register allocation for CompCert has not been explored yet.

CakeML The compiler for CakeML [13] is verified in HOL4. The compiler represents loops as recursive functions and forces all variables a function uses to be parameters through closure conversion. This breaks all live ranges at loop headers. The CakeML compiler assumes all function parameters are live, hence register pressure may increase if closure conversion introduces dead parameters. CakeML does not use SSA with ϕ-functions and delegates register allocation to a non-SSA-based, verified IRC algorithm [6] that performs spilling and register assignment together. In contrast to the CakeML approach, our approach is SSA-based, separates spilling from register assignment, and allows fine-grained control over live range splitting. Our approach does not require closure conversion, but allows functions to refer to variables that are not parameters.

3 Syntax and Semantics of IL

The formal development in LVC uses the intermediate language IL with mutually recursive function definitions and external events (system calls) [12]. For

the presentation of spilling in this paper, we omit mutually recursive function definitions and system calls for the sake of simplicity. IL as used in LVC has a functional and an imperative semantic interpretation [11]. We verify spilling with respect to the imperative semantics, as it simplifies the treatment of the new definitions introduced by spills and loads.

3.1 Expressions

Let $\mathbb{V}$ be the type of values and **exp** be the type of expressions. By convention, v ranges over values and e over expressions. The type of variables $\mathcal{V}$ is isomorphic to the natural numbers $\mathbb{N}$. 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 $[\![\cdot]\!] : \mathbf{exp} \to (\mathcal{V} \to \mathbb{V}_\perp) \to \mathbb{V}_\perp$ that takes an expression and an environment and returns a value or $\perp$ if the evaluation fails. For lists, we use the notation $\overline{x}$ and we lift $[\![\cdot]\!]$ accordingly: $[\![\overline{e}]\!]$ yields $\perp$ if at least one of the expressions in $\overline{e}$ failed to evaluate and the list of the evaluated values otherwise. We use the usual function $\mathsf{fv} : \mathbf{exp} \to \operatorname{set} \mathcal{V}$ that yields the *free variables* of an expression. If V and V' agree on $\mathsf{fv}\, e$, then $[\![e]\!]V = [\![e]\!]V'$. There is a function $\beta : \mathbb{V} \to \{\mathbf{t}, \mathbf{f}\}$ that simplifies the definition of the semantics of the conditional.

3.2 Syntax

IL is a first-order language with a tail-call restriction, which ensures that every IL program corresponds can be implemented without a call stack. The syntax of IL is given in Table 1. We use a separate alphabet $\mathcal{F}$ for function names to enforce a first-order discipline. By convention, f ranges over $\mathcal{F}$.

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

Table 1. Syntax of IL

3.3 Semantics

A **context** is a list of named definitions. By convention, L ranges over contexts. A definition in a context may refer to previous definitions and itself. Notationally, we use contexts like functions and write L_f to access the first element with name f. We have $L_f = \perp$ if no such element exists. We write L^{-f} for the context obtained from L by dropping all definitions before the first definition of f. We write ; for context concatenation and $\emptyset$ for the empty context.

Figure 1 shows the small-step transition relation $\longrightarrow$ of IL. The relation is defined on *configurations* (L, V, s) where L is a context containing tuples of type $\overline{\mathcal{V}} * \mathbf{stmt}$, V is an environment and s is an IL term. Often we write the configuration tuple $L \mid V \mid S$ to have the comma available as another separator. Since only tail recursion is syntactically allowed in IL, no call stack is required. Function application in IL is hence similar to a "goto" with a parallel copy on the variables resulting from parameter passing, and very different from a function call in a language with a call stack.

$$\frac{[\![e]\!]V = v}{L \mid V \mid \texttt{let}\ x\ \texttt{:=}\ e\ \texttt{in}\ s \longrightarrow L \mid V[x \mapsto v] \mid s}\ \textsc{SemLet}$$

$$\frac{[\![e]\!]V = v \quad \beta(v) = b}{L \mid V \mid \texttt{if}\ e\ \texttt{then}\ s_{\mathtt{t}}\ \texttt{else}\ s_{\mathtt{f}} \longrightarrow L \mid V \mid s_b}\ \textsc{SemIf}$$

$$\frac{}{L \mid V \mid \texttt{fun}\ f\ \overline{x}\ \texttt{:=}\ s\ \texttt{in}\ t \longrightarrow f : (\overline{x}, s)\,; L \mid V \mid t}\ \textsc{SemFun}$$

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

Fig. 1. Semantics of IL

3.4 Renaming Apart

Many results in this paper require the input programs to be **renamed apart**, that is, every variable must be assigned at most once and defined before used. In general, renaming apart an imperative program requires SSA with ϕ-functions. As presented in our previous work [11], IL realizes SSA by interpreting variables as binders and emulates ϕ-functions through function applications. We previously established that renamed-apart IL programs are coherent, i.e. they behave equivalently under a semantic interpretation with binders and a semantics with imperative assignables. For this reason, our theorems require programs to be renamed apart, and at the same time rely on the imperative interpretation.

4 Liveness

Liveness over-approximates the semantic (and hence undecidable) notion that a variable is still used *later on*. The notion of liveness used in a register allocation approach greatly impacts the algorithm and its effectiveness. Consider, for example, the different notions of liveness used by graph-coloring register allocation [5] and linear scan [9].

We inductively define a soundness predicate $Z \mid \Lambda \vdash \mathbf{live}\ s : X$ that associates a set of variables X called **live set** with a program s. The **parameter context**

Z maps every defined function to its parameters. The **live-in context** Λ maps every function to a set of variables that contains the variables live in the function body and the parameters, which we call the **live-in set** of the function. We embed the live-in sets in the IL syntax of function definitions, which from now on take the syntactic form $\texttt{fun}\ f\ \overline{x}\ \texttt{:=}\ s_1\ \{X_1\}\ \texttt{in}\ s_2$, in which the function body s_1 is syntactically annotated with its live-in set X_1. We call these sets embedded in the syntax at function bodies **live-in annotations**. In contrast to these annotations, the live set X that appears in the judgment $Z \mid \Lambda \vdash \textbf{live}\ s : X$ is not part of the syntax of IL. The inductive definition of liveness is given in Figure 2 and similar to our previous definition [11]. The liveness predicate allows X to over-approximate the live set, that is, X may contain variables that are not used later on.

LiveLet

$$\frac{X_s \setminus \{x\} \subseteq X \qquad \mathsf{fv}\, e \subseteq X \qquad x \in X_s \qquad Z \mid \Lambda \vdash \textbf{live}\ s : X_s}{Z \mid \Lambda \vdash \textbf{live}\ (\texttt{let}\ x\ \texttt{:=}\ e\ \texttt{in}\ s) : X}$$

LiveApp

$$\frac{\mathsf{fv}\, \overline{e} \subseteq X \qquad \Lambda_f \setminus Z_f \subseteq X}{Z \mid \Lambda \vdash \textbf{live}\ f\ \overline{e} : X}$$

LiveReturn

$$\frac{\mathsf{fv}\, e \subseteq X}{Z \mid \Lambda \vdash \textbf{live}\ e : X}$$

LiveIf

$$\frac{\mathsf{fv}\, e \cup X_1 \cup X_2 \subseteq X \qquad Z \mid \Lambda \vdash \textbf{live}\ s_1 : X_1 \qquad Z \mid \Lambda \vdash \textbf{live}\ s_2 : X_2}{Z \mid \Lambda \vdash \textbf{live}\ (\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2) : X}$$

LiveFun

$$\frac{\overline{x} \subseteq X_1 \qquad f : \overline{x}, Z \mid f : X_1, \Lambda \vdash \textbf{live}\ s_1 : X_1 \qquad X_2 \subseteq X \qquad f : \overline{x}, Z \mid f : X_1, \Lambda \vdash \textbf{live}\ s_2 : X_2}{Z \mid \Lambda \vdash \textbf{live}\ (\texttt{fun}\ f\ \overline{x}\ \texttt{:=}\ s_1\ \{X_1\}\ \texttt{in}\ s_2) : X}$$

Fig. 2. Inductive definition of liveness

4.1 Description of the Rules of the Inductive Predicate

LiveLet requires the live set X of the let statement to contain the free variables of the expression e, and the variables live in the continuation s, except the newly defined variable x. We also require x to be in the live set X_s of the continuation. This reflects that x must be considered live during the let-statement even if x is not used afterwards, because x is overwritten and hence cannot hold a value that is still used later on. LiveReturn requires the free variables of the expression to be live. LiveApp requires the live-ins of the function that are not parameters to be live. LiveIf requires the live variables of the consequence, the alternative, and the free variables of the condition to be live. LiveFun requires that variables live in continuation s_2 are live. The parameters are recorded in the context Z, and the live-ins X_1 are recorded in the context Λ. The live-ins X_1 contain all

variables live in the function body and all parameters, regardless of whether a parameter is used: $\overline{x} \subseteq X_1$. This reflects that unused parameters are overwritten during function application, and hence occupy a register or a spill slot.

4.2 Minimal Live Sets and Live Set Annotations

Live-in annotations uniquely determine the *minimal* live set for every program point; those live sets can be computed by a bottom-up traversal. Since liveness annotations are part of the syntax, every algorithm or judgment formulated on the syntax can easily refer to and compute with the live-ins at function definitions. This allows us to concisely describe how the live sets change during spilling, we can explain changes to live sets by explaining them just for the live-ins at function definitions. The effect on the other live sets in the program is then uniquely determined. We write $Z \mid \Lambda \vdash \textbf{live}\ s$ for $\exists X.\, Z \mid \Lambda \vdash \textbf{live}\ s : X$ and use this notation whenever we want to hide the precise form of the live set.

5 Spilling

Spilling transforms a program into an equivalent program by inserting spills and loads such that the number of registers in the maximal live set is afterwards bounded by a given integer k. In our framework, spilling consists of two steps: First, a spilling algorithm inserts spilling annotations into the program that describe where spills and loads should be placed. Second, the spills and loads are inserted into the program as prescribed by these spilling annotations, which yields the **spilled program**. The spilled program also contains live set annotations at function definitions, and we describe in Section 7.2 how those are recomputed according to the spilling annotations.

Spilling annotations are tree-tuples embedded in the syntax at every subterm. A statement with spilling annotation has the form $s : (S, L, \_)$, where S is the set of variables to be spilled (**spill set**) and L is the set of variables to be loaded (**load set**). The third component is only required if s is a function applications or a function definition, and we discuss its purpose below. We call a statement that contains such annotations a **spill statement**.

A spill statement can be turned into a spilled program via the recursive function doSpill, which we now informally describe. We assume that the variables are partitioned into two countably-infinite sets $\mathcal{V} = \mathcal{V}_R \uplus \mathcal{V}_M$, and require that the spill statement only contains variables from $\mathcal{V}_R$. We further assume an injection $\mathsf{slot} : \mathcal{V}_R \to \mathcal{V}_M$ which we use to generate names for spill slots (cf. CompCert [8]).

$$\mathsf{doSpillLocal}(s : (\underbrace{\{x_1, ..., x_n\}}_{\text{spills}}, \underbrace{\{y_1, ..., y_m\}}_{\text{loads}}, \_)) \quad = \quad \begin{array}{l} \texttt{let slot } \mathtt{x_1} = \mathtt{x_1} \texttt{ in } \ldots \\ \texttt{let slot } \mathtt{x_n} = \mathtt{x_n} \texttt{ in} \\ \texttt{let } \mathtt{y_1} = \texttt{slot } \mathtt{y_1} \texttt{ in } \ldots \\ \texttt{let } \mathtt{y_n} = \texttt{slot } \mathtt{y_n} \texttt{ in } s \end{array}$$

Listing 2. Definition of doSpillLocal

```
1 fun f x y z :=                    R_f = {y, z}, M_f = {c, x, z}
2  if y > 0 then
3   let a := y+z in
4   f x a z                         R_app = {a, z}, M_app = {x, z}
5  else if y = 0 then
6
7
8   x + c                           L = {c, x}
9  else
10  let w := y*y in
11  let a := y+w in
12  f x a z                         R_app = {a}, M_app = {x, z}
```

```
fun f X y z Z :=
 if y > 0 then
  let a := y+z in
  f X a z Z
 else if y = 0 then
  let x := X in
  let c := C in
  x + c
 else
  let w := y*y in
  let a := y+w in
  f X a Z Z
```

Listing 3. A spill statement on the left (non-empty sets in spilling annotations are indicated by equations) and the resulting spilled program on the right. The live-ins of f are $\{x, y, z, c\}$. The variable c is free in f. Lowercase variables denote registers, uppercase variables denote spill slots. In line 4, z is passed in register and memory to avoid loading z in line 3. The application in line 12 implicitly stores x (1st parameter) and loads z (3rd parameter).

To generate the spilled program for $s : (S, L, \_)$, doSpill first prepends the statement s with spills for each variable in S, followed by the loads for each variable in L as depicted in Listing 2. For let statements, conditionals, and return statements this is all that needs to be done. Function definitions and applications require some additional work, which we describe next.

Function definitions take a pair of sets (R_f, M_f) as third component of the spilling annotation: `fun` $f\ x_1, \ldots, x_n$ `:=` s_1 `in` $s_2 : (S, L, (R_f, M_f))$. We call the pair (R_f, M_f) the **live-in cover** and require it to cover the live-ins X_f of f, i.e. $R_f \cup M_f = X_f$. The set R_f specifies the variables the function expects to reside in registers, and the set M_f specifies the variables the function expects to reside in memory. The sets R_f and M_f are not necessarily disjoint, as a function may want a variable to reside both in register and in memory when it is applied (see Listing 3). Besides inserting spills and loads according to S and L as already described, the function parameters must be modified to account for parameters that are passed in spill slots. For this purpose, every parameter $x_i \in M_f \setminus R_f$ is replaced by the name $\mathsf{slot}\, x_i$ in $\overline{x}$. Furthermore, for any parameter $x_i \in M_f \cap R_f$ an additional parameter with name $\mathsf{slot}\, x_i$ is inserted directly after x_i.

Function applications have a pair of sets (R_{app}, M_{app}) as third component of spilling information and take the form $f\, y_1, \ldots, y_n : (S, L, (R_{app}, M_{app}))$. We require all function arguments y_i to be variables, and that $R_{app} \cup M_{app} = \{y_1, \ldots, y_n\}$. The sets R_{app} and M_{app} indicate the availability of argument variables at the function application. If an argument variable y_i is available in a register, then the spilling algorithm sets $y_i \in R_{app}$, if it is in memory, then $y_i \in M_{app}$. Besides inserting spills and loads according to S and L as already described, doSpill modifies the argument vector $y_1, \ldots, y_n$. For every parameter $x_i \in R_f$ such that the corresponding argument variable y_i is not in R_{app} (i.e.

not available in a register), the variable y_i is replaced by the name $\mathsf{slot}\, y_i$ in the argument vector. For every parameter $x_i \in M_f \setminus R_f$ such that the corresponding argument variable y_i is in M_{app} (i.e. available in memory), the variable y_i is replaced by the name $\mathsf{slot}\, y_i$ in the argument vector. Furthermore, for every parameter $x_i \in M_f \cap R_f$ an additional argument is inserted directly after the corresponding argument variable (y_i or $\mathsf{slot}\, y_i$) in $y_1, \ldots, y_n$, and the name of the additional argument is $\mathsf{slot}\, y_i$ if $y_i \in M_{app}$ and y_i otherwise. In this way, R_f and M_f are used to avoid implicit loads and stores at function application if availability, as indicated in R_{app} and M_{app}, permits. Since spill slots are just a partition of the variables, parameter passing can copy between spill slots and registers if the argument variable y_i for a register parameter x_i is only available in memory, or vice versa. This fits nicely in our setting, as we handle the generation of these implicit spills and loads later on, when parameter passing is lowered to parallel moves. In line 12 of Listing 3, for example, the application implicitly loads z. In contrast, availability of z in both register and memory at the application in line 4 allows avoiding any implicit loads and stores. Assuming $y > 0$ holds for most executions, this is beneficial for performance.

6 A Correctness Criterion for Spilling

We define a correctness predicate for spilling on spill statements of the form $Z \,|\, \Sigma \,|\, R \,|\, M \vdash \mathbf{spill_k}\ \ s : (S, L, \_)$. Note that as described in Section 5, the spilling annotation $(S, L, \_)$ is embedded in the syntax. The correctness predicate is defined relative to sets R and M, which contain the variables currently in registers, and in memory, respectively. Additionally, the *parameter context* Z maps function names to their parameter list, and the *live-in cover context* Σ maps functions to their live-in cover. The parameter k is the register bound. The rules defining the predicate are given in Figure 3.

6.1 Description of the Rules of the Inductive Predicate

The predicate consists of two generic rules that handle spilling and loading, and one rule for each statement. Rules for statements only apply once spills and loads have been handled. This is achieved by requiring empty spill and load sets in statement rules, and requiring an empty spill set in the load rule. SpillSpill requires $S \subseteq R$ to ensure only variables currently in registers are spilled. The new memory state is $M \cup S$. SpillLoad requires the spill set to be empty. Its second premise ensures there are enough free registers to load all values. The *kill set* K represents the variables that may be overwritten because they are not used anymore or are already spilled. $R \setminus K \cup L$ is the new register state after loading. Clearly, K is most useful if $K \subseteq R$, but our proofs do not require this restriction. We also do not include K in the spilling annotation, as the spilling algorithm would have to compute liveness information to provide it. Simple spilling algorithms, such as the one we verify in Section 8.1, never need to compute liveness information.

$$\textsc{SpillSpill}\quad \frac{S \subseteq R}{Z \mid \Sigma \mid R \mid M \cup S \vdash \mathbf{spill_k}\ \ s : (\emptyset, L, \_)} \qquad \textsc{SpillLoad}\quad \frac{L \subseteq M \qquad |R \setminus K \cup L| \leq k}{Z \mid \Sigma \mid R \setminus K \cup L \mid M \vdash \mathbf{spill_k}\ \ s : (\emptyset, \emptyset, \_)}$$

$$\textsc{SpillReturn}\quad \frac{\mathsf{fv}\, e \subseteq R}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ e : (\emptyset, \emptyset)} \qquad \textsc{SpillApp}\quad \frac{\Sigma_f = (R_f, M_f) \qquad R_f \setminus Z_f \subseteq R \qquad M_f \setminus Z_f \subseteq M \qquad \overline{y} = R_{app} \cup M_{app} \qquad M_{app} \subseteq M \qquad R_{app} \subseteq R}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ (f\ \ \overline{y}) : (\emptyset, \emptyset, (R_{app}, M_{app}))}$$

$$\textsc{SpillIf}\quad \frac{\mathsf{fv}\, e \subseteq R \qquad Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s_1 \qquad Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s_2}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ (\texttt{if}\ \ e\ \ \texttt{then}\ \ s_1\ \ \texttt{else}\ \ s_2) : (\emptyset, \emptyset)}$$

$$\textsc{SpillLet}\quad \frac{\mathsf{fv}\, e \subseteq R \qquad |R \setminus K \cup \{x\}| \leq k \qquad Z \mid \Sigma \mid R \setminus K \cup \{x\} \mid M \vdash \mathbf{spill_k}\ \ s}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ (\texttt{let}\ \ x\ \ \texttt{:=}\ \ e\ \ \texttt{in}\ \ s) : (\emptyset, \emptyset)}$$

$$\textsc{SpillFun}\quad \frac{|R_f| \leq k \qquad f : \overline{x}; Z \mid f : (R_f, M_f); \Sigma \mid R_f \mid M_f \vdash \mathbf{spill_k}\ \ s_1 \qquad R_f \cup M_f = X_f \qquad f : \overline{x}; Z \mid f : (R_f, M_f); \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \ s_2}{Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ (\texttt{fun}\ \ f\ \ \overline{x}\ \ \texttt{:=}\ \ s_1\ \ \{X_f\}\ \ \texttt{in}\ \ s_2) : (\emptyset, \emptyset, (R_f, M_f))}$$

Fig. 3. Inductive correctness predicate $\mathbf{spill_k}$

SpillReturn requires that the free variables are in the registers. SpillIf requires the consequence and the alternative to fulfill the predicate on the same configuration, and that the variables used in the condition are in registers. SpillLet deals with the new variable x, which needs a register. The resulting register state is $R \setminus K \cup \{x\}$, the size of which must be bounded by the register bound k. This imposes a lower bound on k. The kill set K reflects that there might be a variable y holding the value of a variable required to evaluate the expression e, that is then overwritten to store the value of x. In this case $K = \{y\}$.

SpillApp uses the sets R_f and M_f from the corresponding function definition. The premises $R_f \setminus Z_f \subseteq R$ and $M_f \setminus Z_f \subseteq M$ require that all live-ins of the function except parameters are available in registers and memory at the application. The remaining premises require that all argument variables are available either in the registers (R_{app}) or in the memory (M_{app}), as discussed in Section 5. Note that the argument vector $\overline{y}$ is variables only, i.e. applications can only have variables as arguments. SpillFun refers to the live-in set X_f embedded in the syntax to require that the live-in cover (R_f, M_f) covers the live-ins X_f of the program: $R_f \cup M_f = X_f$. The rule also requires the function to expect at most k variables in registers: $|R_f| \leq k$. The parameters and the live-in cover are recorded in the context. The condition for the function body s_1 uses R_f and M_f as register and memory sets, respectively.

6.2 Formalization of the Spill Predicate in Coq

The predicate $\mathbf{spill_k}$ is realized with five rules in the Coq development instead of the seven rules presented here. Each of the five rules corresponds to a consecutive application of SpillSpill, SpillLoad and one of the statement-specific rules. The five-rule system behaves better under inversion and induction in Coq, but we think the formulation with seven rules provides more insight. The Coq development contains a formal proof of the equivalence of the two systems.

7 Soundness of the Correctness Predicate

In this section we show that our spilling predicate is sound. We show that if s is renamed apart and all variables in s are in $\mathcal{V}_R$, and the spilling and live-in annotations in s are sound, the following holds for the spilled program s':

(Section 7.1) all variables in s' are in a register when used
(Section 7.2) at most k registers are used in s'
(Section 7.3) s and s' have the same behavior

7.1 Variables in Registers

Figure 4 defines a predicate that ensures every variable is in a register when used. The inference rules are straightforward. The predicate also ensures that let-statements that assign to memory have a single register on the right-hand side. We define $\mathsf{merge}\,(R, M) = R \cup M$ and $\mathsf{slotMerge}\,(R, M) = R \cup \mathsf{slot}\ M$ and analogously their pointwise liftings.

$$\textsc{ViRLoad}\ \frac{x \in \mathcal{V}_R \qquad y \in \mathcal{V}_M \qquad \mathbf{vir}\ s}{\mathbf{vir}\ \texttt{let}\ x\ \texttt{:=}\ y\ \texttt{in}\ s} \qquad \textsc{ViRLet}\ \frac{\mathsf{fv}\ e \subseteq \mathcal{V}_R \qquad \mathbf{vir}\ s}{\mathbf{vir}\ \texttt{let}\ x\ \texttt{:=}\ e\ \texttt{in}\ s} \qquad \textsc{ViRReturn}\ \frac{\mathsf{fv}\ e \subseteq \mathcal{V}_R}{\mathbf{vir}\ e}$$

$$\textsc{ViRIf}\ \frac{\mathsf{fv}\ e \subseteq \mathcal{V}_R \qquad \mathbf{vir}\ s \qquad \mathbf{vir}\ t}{\mathbf{vir}\ \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ t} \qquad \textsc{ViRApp}\ \frac{}{\mathbf{vir}\ f\ \overline{y}} \qquad \textsc{ViRFun}\ \frac{\mathbf{vir}\ s \qquad \mathbf{vir}\ t}{\mathbf{vir}\ \texttt{fun}\ f\ \overline{x}\ \texttt{:=}\ s\ \texttt{in}\ t}$$

Fig. 4. Predicate vir

Lemma 1. *Let* $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ s$ *and* $Z \mid \Lambda \vdash \mathbf{live}\ s$ *and let* s *be renamed apart and let all variables in* s *be in* $\mathcal{V}_R$*. If* $R \cup M \cup \bigcup Z \subseteq \mathcal{V}_R$ *then* $\mathbf{vir}\,(\mathsf{doSpill}\ Z\ \Lambda\ s)$.

Proof. The conditions follow directly by induction on $\mathbf{spill_k}\ s$.

7.2 Register Bound

After the spilling phase, the liveness information in the program changed tremendously. Spills and loads introduce new live ranges, and shorten live ranges of already defined variables. To prove correctness of the spilling predicate, we must show that after spilling the register pressure is lowered to k. To formally establish the bound, we show that the number of variables from $\mathcal{V}_R$ in each live set in the spilled program is bounded by k. The following observation is key to this proof: The live-ins of a function after spilling can be obtained from the live-ins of the function before spilling by keeping the variables passed in registers, and adding the slots of the variables passed in memory. This property can be seen in the rule SpillFun, where we require $R_f \cup M_f = X_f$.

In the Coq development, the statements of the following lemmas involve the algorithm that reconstructs minimal liveness information we informally described in Section 4.2, but omitted in this presentation for the sake of simplicity.

Lemma 2. *Let* $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ s$ *and* $Z \mid \mathsf{merge}\,\Sigma \vdash \mathbf{live}\ s$ *and let* s *be renamed apart and let all variables in* s *be in* $\mathcal{V}_R$*. If* $R \cup M \cup \bigcup Z \subseteq \mathcal{V}_R$ *then* $Z \mid \mathsf{slotMerge}\,\Sigma \vdash \mathbf{live}\ \mathsf{doSpill}\ Z\,\Sigma\,s$*.*

Proof. By induction on $\mathbf{spill_k}\ s$; mostly simple but tedious set constraints.

Lemma 3. *Let* $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ s$ *and let* s *be renamed apart and let all variables in* s *be in* $\mathcal{V}_R$*. If* $|R| \leq k$ *and* $R \cup M \cup \bigcup Z \subseteq \mathcal{V}_R$ *then for live set* X *in the minimal liveness derivation* $Z \mid \mathsf{slotMerge}\,\Sigma \vdash \mathbf{live}\ \mathsf{doSpill}\ Z\,\Sigma\,s$ *the bound* $|\mathcal{V}_R \cap X| \leq k$ *holds.*

Proof. By induction on s. The proof uses a technical lemma about the way the liveness reconstruction deals with forward-propagation that was difficult to find.

7.3 Semantic Equivalence

In this section we show that the spilled program is semantically equivalent to the original program. Semantic equivalence means trace-equivalence à la CompCert. As proof tool we use a co-inductively defined simulation relation. See our previous work [11, 12] for details on simulation and proof technique. The verification is done with respect to the imperative semantics of IL. This allows for a simple treatment of the new variables that each spill and each load introduces. A typical spill and load looks as follows:

```
let x = 5 in                let x = 5 in
fun f () = x in             fun f () = x in
                            let X = x in        // spill
...                         ...
                            let x = X in        // load
f()                         f()
```

Note that in a semantics with binding, serious effort would be required to introduce additional function parameters after spilling and loading. In the above example, f would need to take x as a parameter. We postpone the introduction of additional parameters to a phase after spilling, where we switch to the functional semantics again to do register allocation. Changing the semantics from imperative to functional corresponds to SSA construction and is in line with practical implementations of SSA-based register allocation [4] that break the SSA invariant during spilling, and then perform some form of SSA (re-)construction.

Lemma 4. *Let s be a spill statement where all variables are renamed apart and in $\mathcal{V}_R$. Let $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ s$ and $Z \mid \mathsf{merge}\,\Sigma \vdash \mathbf{live}\ s : X$ and $V =_R V'$ and $V =_M (\lambda x.V'(\mathsf{slot}\ x))$. If V' is defined on $R \cup \mathsf{slot}\ M$ and $R \cup M \subseteq \mathcal{V}_R$ and L and L' are suitably related then (L, V, s) and $(L', V', \mathsf{doSpill}\ Z\ \Sigma\ s)$ are in simulation.*

8 Case Study: Verified Spilling Algorithms

A spilling algorithm translates a statement with live-in annotations to a spill statement, that is, it inserts spilling annotations. The following algorithms are implemented in Coq and verified using the correctness predicate.

8.1 SimpleSpill

The naive spilling algorithm simpleSpill loads the required values before each statement, without considering that some values might still be available in a register. After a variable is assigned, the algorithm immediatelly spills the variable. This is a very simple algorithm, and it corresponds to the spilling strategy used in the very first version of CompCert [8].

Theorem 1. *Let $Z \mid \mathsf{merge}\,\Sigma \vdash \mathbf{live}\ s$ and let s be renamed apart and let all variables in s be in $\mathcal{V}_R$ and let every expression in s contain at most k different variables. If every live set X in s is bounded by $R \cup M$ and the first component in Σ_f is empty for every f then $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ \mathsf{simpleSpill}\ s$.*

Proof. By induction on s in less than 100 lines.

8.2 SplitSpill

The spilling algorithm splitSpill follows three key ideas: Variables are loaded as late as possible, but in contrast to simpleSpill, only values not already available in registers. If a register must be freed for a load, the algorithm lets an oracle choose the variable to be spilled from the list of variables live and currently in a register. The correctness requirement for the oracle is trivial. The oracle enables live range splitting based on an external heuristic, similar to the approach of Braun [4]. In contrast to Braun's algorithm, splitSpill cannot hoist loads from their uses.

Theorem 2. *Let* $Z \mid \mathsf{merge}\, \Sigma \vdash \mathbf{live}\ s$ *and let* s *be renamed apart and let all variables in* s *be in* $\mathcal{V}_R$ *and let every expression in* s *contain at most* k *different variables. If every live set* X *in* s *is bounded by* $R \cup M$ *and and for every* f *such that* $(R_f, M_f) = \Sigma_f$ *we have* $|R_f| \leq k$ *then* $\Sigma \mid Z \mid R \mid M \vdash \mathbf{spill_k}\ \mathsf{splitSpill}\, s$.

Proof. By induction on s in less than 500 lines.

9 Translation Validation with Repair

We devise a translation validator repairSpill for our correctness predicate. The translation validator repairSpill operates on a statement with liveness and spilling annotations, and assumes the liveness to be sound. Besides deciding whether the spilling annotation is sound, repairSpill also repairs the spilling annotation if necessary. The output of repairSpill always contains sound spilling annotations. Furthermore, we show that repairSpill leaves the spilling annotations unchanged if they are already sound with respect to the provided live-in annotation and our correctness predicate.

To explain the principle behind repairSpill, it is instructive to understand how the algorithm recomputes a live-in cover (R_f, M_f) from the (possibly unsound) spilling annotation using the corresponding live-ins X_f from the sound live-in annotation. Let $\mathsf{take}\ k\, X$ be a function that yields a k-sized subset from X or X if $|X| \leq k$. The new live-in cover (R'_f, M'_f) is obtained as follows:

$$R'_f = \mathsf{take}\ k\, (R_f \cap X_f)$$
$$M'_f = (X_f \setminus R'_f) \cup (M_f \cap X_f)$$

These equations have two important properties: First, it holds $R'_f \cup M'_f = X_f$, so the equations produce a correct live-in cover independent of the input sets R_f and M_f. Second, if $R_f \cup M_f = X_f$ and $|R_f| \leq k$ then $R'_f = R_f$ and hence $M'_f = M_f$, i.e. the original live-in cover is retained, if it is valid. repairSpill transforms every spill and load set in the spilling annotation in a similar way such that these two properties hold.

The kill sets K appearing in the derivation of $\mathbf{spill_k}$ are not recorded in the spilling annotation, because we did not want to require the spilling algorithm to compute them. To check whether a spilling annotation is correct, repairSpill must reconstruct kill sets. Maximal kill sets can be reconstructed in a backwards fashion from spilling annotation similar to how minimal liveness information can be reconstructed (Section 4.2). A maximal kill set upper-bounds the variables that can be soundly killed. The correctness of the kill sets repairSpill reconstructs depends on the correctness of the spilling annotation. For this reason, we designed repairSpill in such a way that that the correctness of its output does not depend on the correctness of the kill sets. This is similar to the fact that the correctness of R'_f and M'_f in the equations above does not depend on the correctness of R_f and M_f. If the spilling annoation is correct, however, the kill sets are correct and ensure that the algorithm does not change the spilling annotation.

Consider, for example, a conditional `if` e `then` s `else` $t : (S, L, \_)$ where S and L are the potentially unsound spill and load sets. From the memory state (R, M) and the assumption $\mathsf{fv}\, e \subseteq R \cup M$ the algorithm produces sound spill and load sets S' and L' that agree with S and L if those are already sound. For correctness, we only require $\mathsf{fv}\, e \subseteq R$. We use the following definitions:

$$\begin{aligned} \mathsf{pick}\, k\, s\, t &= s \cup \mathsf{take}\, (k - |s|)\, (t \setminus s) \\ \mathsf{pickload}\, k\, R\, M\, S\, L\, e &= (\mathsf{fv}\, e \cap R \cap Q) \cup P \\ \text{where } Q &= L \cap ((S \cap R) \cup M) \\ \text{and where } P &= \mathsf{pick}\, (k - |\mathsf{fv}\, e \cap R|)\, (\mathsf{fv}\, e \setminus R)\, (Q \setminus (\mathsf{fv}\, e \cap R)) \end{aligned}$$

We can now pick the new load set $L' = \mathsf{pickload}\, k\, R\, M\, S\, L\, e$. Lemma 5 and Lemma 6 establish that L' satisfies the register bound and loads the variables necessary to evaluate e. Lemma 7 shows that $L' = L$ if L was already correct.

Lemma 5. *If* $|\mathsf{fv}\, e| \leq k$ *then* $|\mathsf{pickload}\, k\, R\, M\, S\, L\, e| \leq k - |\mathsf{fv}\, e \cap R \setminus (L \cap (S \cup M))|$.

Lemma 6. $(\mathsf{fv}\, e \cap R \setminus (L \cap (S \cup M))) \cup (\mathsf{fv}\, e \setminus R) \subseteq \mathsf{pickload}\, k\, R\, M\, S\, L\, e$.

Lemma 7. *Let* $S \subseteq R$ *and* $L \subseteq S \cup M$ *and* $\mathsf{fv}\, e \setminus R \subseteq L$ *and* $|\mathsf{fv}\, e \cup L| \leq k$ *then* $\mathsf{pickload}\, k\, R\, M\, S\, L\, e = L$.

We developed definitions similar to pickload that allow repairSpill to transform every spill and load set in the spilling annotation. Correct sets are retained, and incorrect sets are repaired. NP-completeness of the spilling problem makes it unlikely that quality guarantees hold for a polynomial-time repair algorithm.

Theorem 3. (Correctness) *Let* $Z \mid \mathsf{merge}\, \Sigma \vdash$ **live** $s : X$ *and let* R, M *be sets of variables such that* $X \subseteq R \cup M$ *and let every expression in* s *contain at most* k *different variables. If for every* f *such that* $(R_f, M_f) = \Sigma_f$ *we have* $|R_f| \leq k$ *then* $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ (\mathsf{repairSpill}\ k\, Z\, \Sigma\, R\, M\, s)$.

Theorem 4. (Idempotence) *Let* s *be renamed apart and let* $Z \mid \mathsf{merge}\, \Sigma \vdash$ **live** $s : X$ *and let* $Z \mid \Sigma \mid R \mid M \vdash \mathbf{spill_k}\ s$. *If for every* f *such that* $(R_f, M_f) = \Sigma_f$ *we have* $|R_f| \leq k$ *then* $\mathsf{repairSpill}\ k\, Z\, \Sigma\, R\, M\, s = s$.

10 Conclusion

We presented a correctness predicate for spilling algorithms that permits arbitrary live range splitting. To our knowledge, it is the first formally proven correctness predicate for spilling on term-based SSA and the first to support arbitrary live range splitting. The conditions of our correctness predicate are mainly set constraints, and our case studies show that the predicate simplifies correctness proofs of spilling algorithms.

Based on the correctness predicate, we defined a translation validator for spilling algorithms with repair. The algorithm takes any spilling annotation and

repairs it if necessary. Our algorithm combines the flexibility of translation validation with the correctness guarantees of verification.

This work is part of the verified compiler LVC. LVC has about 50k LoC and extracts to an executable verified compiler. The spilling framework presented in this paper is about 8k LoC. A considerable difference between the paper and the formal proofs is the presentation of liveness information: In the formal development, liveness reconstruction must, of course, be handled by a Coq function, and we must prove that is function is correct and yields minimal live sets.

References

[1] Gilles Barthe, Delphine Demange, and David Pichardie. "A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert". In: *ESOP*. Vol. 7211. LNCS. Tallinn, Estonia, Mar. 24–Apr. 1, 2012.

[2] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. "Formal Verification of Coalescing Graph-Coloring Register Allocation". In: *ESOP*. Vol. 6012. LNCS. Paphos, Cyprus, Mar. 20–28, 2010.

[3] Florent Bouchez, Alain Darte, and Fabrice Rastello. "On the complexity of spill everywhere under SSA form". In: *LCTES*. San Diego, California, USA, June 13–15, 2007.

[4] Matthias Braun and Sebastian Hack. "Register Spilling and Live-Range Splitting for SSA-Form Programs". In: *CC*. Vol. 5501. LNCS. York, UK, Mar. 22–29, 2009.

[5] Gregory J. Chaitin. "Register Allocation & Spilling via Graph Coloring". In: *PLDI*. Boston, Massachusetts, USA, June 23–25, 1982.

[6] Lal George and Andrew W. Appel. "Iterated Register Coalescing". In: *ACM Trans. Program. Lang. Syst.* 18.3 (1996).

[7] Sebastian Hack, Daniel Grund, and Gerhard Goos. "Register Allocation for Programs in SSA-Form". In: *CC*. Vol. 3923. LNCS. Vienna, Austria, Mar. 30–31, 2006.

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

[9] Massimiliano Poletto and Vivek Sarkar. "Linear scan register allocation". In: *TOPLAS* 21.5 (1999).

[10] Silvain Rideau and Xavier Leroy. "Validating Register Allocation and Spilling". In: *CC*. Vol. 6011. LNCS. Paphos, Cyprus, Mar. 20–28, 2010.

[11] 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.

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

[13] Yong Kiam Tan et al. "A new verified compiler backend for CakeML". In: *ICFP*. Nara, Japan, Sept. 18–22, 2016.