Validating Register Allocation and Spilling

Silvain Rideau[1] and Xavier Leroy[2]

[1] École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France
silvain.rideau@ens.fr
[2] INRIA Paris-Rocquencourt, BP 105, 78153 Le Chesnay, France
xavier.leroy@inria.fr

Abstract. Following the translation validation approach to high-assurance compilation, we describe a new algorithm for validating *a posteriori* the results of a run of register allocation. The algorithm is based on backward dataflow inference of equations between variables, registers and stack locations, and can cope with sophisticated forms of spilling and live range splitting, as well as many architectural irregularities such as overlapping registers. The soundness of the algorithm was mechanically proved using the Coq proof assistant.

1 Introduction

To generate fast and compact machine code, it is crucial to make effective use of the limited number of registers provided by hardware architectures. Register allocation and its accompanying code transformations (spilling, reloading, coalescing, live range splitting, rematerialization, etc) therefore play a prominent role in optimizing compilers.

As in the case of any advanced compiler pass, mistakes sometimes happen in the design or implementation of register allocators, possibly causing incorrect machine code to be generated from a correct source program. Such compiler-introduced bugs are uncommon but especially difficult to exhibit and track down. In the context of safety-critical software, they can also invalidate all the safety guarantees obtained by formal verification of the source code, which is a growing concern in the formal methods world.

There exist two major approaches to rule out incorrect compilations. *Compiler verification* proves, once and for all, the correctness of a compiler or compilation pass, preferably using mechanical assistance (proof assistants) to conduct the proof. *Translation validation* checks *a posteriori* the correctness of one run of compilation: a *validator*, conceptually distinct from the compiler itself, is given the intermediate code before and after a compilation pass, and verifies that they behave identically using static analysis or (specialized) theorem proving technology [1–4]. For additional confidence, the validator can itself be mechanically verified once and for all; this provides soundness guarantees as strong as compiler verification and reduces the amount of compiler code that needs to be proved correct, at the expense of weaker completeness guarantees [5].

This paper describes a new algorithm to validate (in one pass) register allocation plus splitting, reloading, coalescing, live range splitting, dead code elimination, and enforcement of calling conventions and architectural constraints on registers. This algorithm is based on a backward dataflow analysis that refines standard liveness analysis. It comes accompanied with a machine-checked proof of soundness, conducted using the Coq proof assistant [6, 7]. Our algorithm improves on an earlier algorithm by Huang, Childers and Soffa [8] because it is mechanically proved and because it can deal with overlapping registers. (See section 6 for a discussion.)

This work is part of the CompCert project, which aims at formally verifying a realistic optimizing compiler for the C language, usable in the context of critical embedded systems [9]. Currently, CompCert follows the compiler verification approach for its register allocation and spilling/reloading passes. While the verified register allocator is a state-of-the-art George-Appel graph coloring allocator [10], the spilling strategy that was proved correct is very naive: it inserts spills after every definition and reloads before every use of a temporary that could not be allocated to a register, reserving some registers specially for this purpose [11, section 11]. This strategy is adequate for a register-rich target architecture such as the PowerPC, but more sophisticated strategies are needed to retarget CompCert to a register-poor architecture like x86. Proving those sophisticated strategies is a daunting task. The verified validation algorithm presented in this paper offers an attractive alternative, reducing the amount of code that needs to be proved and enabling the use of advanced spilling strategies. Moreover, we can experiment with various register allocation algorithms and spilling strategies without having to re-do any proofs.

The remainder of this paper is organized as follows. Section 2 outlines the source and target languages for the untrusted register allocator and characterizes the code transformations it is allowed to make. Section 3 describes our validation algorithm. Section 4 sketches its proof of soundness. Section 5 discusses experience gained with a prototype implementation. Related work is reviewed in section 6, followed by concluding remarks in section 7.

2 A Bird's Eye View of Register Allocation and Spilling

2.1 Source Language

As input for register allocation, we consider the RTL intermediate language of the CompCert compiler [11, section 6]. This is a standard Register Transfer Language where control is represented by a control flow graph (CFG). Each node of a CFG carries an abstract instruction, corresponding roughly to one machine instruction but operating over variables x (also called temporaries) instead of hardware registers. Every function has an unlimited supply of variables and their values are preserved across function calls. Each variable has a machine type comprising a register class (typically, `int` or `float`) and a bit size (8, 16, 32, 64).

Control-flow graphs:

g	$::= p \mapsto I$		finite map

CFG nodes:

p, s	$\in N$		

RTL instructions:

I	$::=$	$\texttt{nop}(s)$	no operation
	$\mid$	$\texttt{op}(op, \vec{x}, x_d, s)$	arithmetic operation
	$\mid$	$\texttt{load}(\kappa, mode, \vec{x}, x_d, s)$	memory load
	$\mid$	$\texttt{store}(\kappa, mode, \vec{x}, x_s, s)$	memory store
	$\mid$	$\texttt{call}(\tau, id, \vec{x}, x_d, s)$	function call
	$\mid$	$\texttt{cond}(cond, \vec{x}, s_{true}, s_{false})$	conditional branch
	$\mid$	$\texttt{return}(x)$	function return

Each RTL instruction carries the list of its successors s in the CFG. For example, $\texttt{nop}(s)$ performs no computation and continues at node s, like an unconditional branch. $\texttt{op}(op, \vec{x}, x_d, s)$ applies the arithmetic operation op (taken from a machine-dependent set of operators) to the values of variables $\vec{x}$, stores the result in variable x_d, and continues at s. $\texttt{load}(\kappa, mode, \vec{x}, x_d, s)$ loads a memory quantity κ (e.g. "8-byte signed integer" or "64-bit float") from an address determined by applying addressing mode $mode$ to the values of registers $\vec{x}$, stores the result in x_d, and continues at s. $\texttt{store}(\kappa, mode, \vec{x}, x_s, s)$ is similar, except that the value of x_s is stored at the computed address instead. $\texttt{cond}(cond, \vec{x}, s_{true}, s_{false})$ evaluates the boolean condition $cond$ over the values of $\vec{x}$ and continues at s_{true} or s_{false} depending on the result. $\texttt{return}(x)$ terminates the current function, returning the value of x as the result. Finally, $\texttt{call}(\tau, id, \vec{x}, x_d, s)$ calls the function named id, giving it the values of $\vec{x}$ as arguments and storing the returned result in x_d. The τ parameter is the type signature of the call, specifying the number and types of arguments and results: this is used during register allocation to determine the calling conventions for the call. The full RTL language, described in [11], supports additional forms of function calls such as calls through a function pointer and tail calls, which we omit here for simplicity.

RTL functions:

$$f ::= \{\texttt{name} = id;\ \texttt{typesig} = \tau;\ \texttt{params} = \vec{x};\ \texttt{code} = g;\ \texttt{entrypoint} = p\}$$

An RTL function is defined by its name, its type signature, the list of parameter variables, a CFG, and a node in the CFG that corresponds to the function entry point.

2.2 Target Language

The purpose of register allocation is to transform RTL functions into LTL functions. LTL stands for "Location Transfer Language" and is a minor variation on RTL where variables are replaced by *locations*. A location is either a machine

register r or a slot $S(\delta, n)$ in the activation record of the function; δ is the byte offset and n the byte size of the slot.

Locations:

ℓ	$::= r$	machine register
	$\mid S(\delta, n)$	stack slot

Control-flow graphs:

$$g' ::= p \mapsto I'$$

LTL instructions:

I'	$::= \texttt{nop}(s)$	no operation
	$\mid \texttt{op}(op, \vec{\ell}, \ell, s)$	arithmetic operation
	$\mid \texttt{load}(\kappa, mode, \vec{\ell}, \ell_d, s)$	memory load
	$\mid \texttt{store}(\kappa, mode, \vec{\ell}, \ell_s, s)$	memory store
	$\mid \texttt{call}(\tau, id, s)$	function call
	$\mid \texttt{cond}(cond, \vec{\ell}, s_{true}, s_{false})$	conditional branch
	$\mid \texttt{return}$	function return

LTL functions:

$$f' ::= \{\texttt{name} = id;\ \texttt{typesig} = \tau;\ \texttt{code} = g';\ \texttt{entrypoint} = p\}$$

Most LTL instructions are identical to RTL instructions modulo the replacement of variables x by locations ℓ. However, function calls and returns are treated differently: the locations of arguments and results are not marked in the `call` and `return` instructions nor in the `params` field of functions, but are implicitly determined by the type signature of the call or the function, following the calling conventions of the target platform. We model calling conventions by the following three functions:

- `arguments`(τ): the list of locations for the arguments of a call to a function with signature τ. The LTL code is responsible for moving the values of the arguments to these locations (registers or stack slots) before the `call` instruction.
- `parameters`(τ): the list of locations for the parameters of a function with signature τ. On entrance, the LTL function expects to find the values of its arguments at these locations, and is responsible for moving them to other locations if desired. `parameters`(τ) is usually identical to `arguments`(τ) modulo relocation of stack slot offsets.
- `result`(τ): the location used to pass the return value for a function with signature τ.

2.3 The Effect of Register Allocation on the Code

The essence of register allocation is to replace variables by the locations that were assigned to it in each instruction of the source RTL code, leaving the rest of the instruction unchanged. For example, the RTL instruction $\texttt{op}(\texttt{add}, x.y, z, s)$

can become the LTL instruction $\texttt{op}(\texttt{add}, \texttt{EAX.EBX}, \texttt{EAX}, s)$ if the allocator decided to assign x and z to register `EAX` and y to register `EBX` at this program point. However, this is not the only effect of register allocation on the code: it can also insert or delete some instructions in the following cases.

- **Spilling:** a move from a register r to a stack slot is inserted at some point after an instruction that assigns r, to save the result value on the stack and free the register r for other uses.
- **Reloading:** symmetrically, a move from a stack slot to a register is inserted at some point before a use of r.
- **Coalescing:** some variable copies $\texttt{op}(\texttt{move}, x, y, s)$ present in the input RTL code may disappear if the register allocator assigned the same location to x and y. We model this deletion as replacing the $\texttt{op}(\texttt{move}, \ldots)$ instruction by a `nop` instruction.
- **Live range splitting:** if the allocator decided to split a live range of a variable x into several variables $x_1, \ldots, x_n$ connected by `move` instructions, some of these moves may remain in the generated LTL code as newly inserted instructions.
- **Enforcement of calling conventions:** additional moves may be inserted in the generated LTL code to deposit arguments to function calls and return values of functions in the locations dictated by the calling conventions, and fetch function parameters and return values from these locations.
- **Enforcement of architectural constraints:** the register allocator can also introduce move instructions to work around irregularities of the target architecture: two-address instructions, special registers, etc.
- **Dead code elimination:** the register allocator can also eliminate side effect-free instructions such as `op` and `load` whose result variables are never used. Dead code elimination can be performed in a separate pass prior to register allocation, but the availability of liveness information during register allocation makes it convenient to perform dead code elimination at the same time.

The validation algorithm we present next is able to cope with all these modifications of the code performed during register allocation. Other code transformations that sometimes accompany register allocation, such as rematerialization, are discussed in section 7.

3 The Validation Algorithm

Like intraprocedural register allocation itself, the validator proceeds function per function. It takes as input an RTL function f, the corresponding LTL function f' produced by the untrusted register allocator, and a partial map φ from the CFG nodes of f' to those of f.

The purpose of φ is to connect the computational instructions of the LTL code back to the corresponding instructions in the original RTL. Since deleted

instructions are not actually removed but simply turned into LTL `nop` instructions, φ also maps these `nop` instructions back to the corresponding deleted RTL instruction. Finally, LTL `move` instructions that were inserted during register allocation are not in the domain of φ, indicating that they are new. (All these properties of φ are checked during validation.) We assume that the register allocator has been lightly instrumented to produce this mapping φ and give it as additional argument to our validator.

The validation algorithm proceeds in two steps:

- A set of structural checks (section 3.1) verifies that the computational instructions in the two CFGs match properly, that their successors agree, and that the φ mapping is consistent.
- A backward dataflow analysis (section 3.2) establishes that the same values flow in both CFGs.

The combination of these two steps suffices to ensure that the two functions f and f' behave identically at run-time (as proved in section 4).

3.1 Structural Checks

The main structural check is performed on each pair of RTL instructions and LTL instructions that match according to the φ mapping. For each mapping $p' \mapsto p$ in φ, the validator calls the following `check_instr` predicate:

$\texttt{check\_instr}(f, f', \varphi, p, p') =$
  let $I = f.\texttt{code}(p)$ and $I' = f'.\texttt{code}(p')$ in
  let $s_1, \ldots, s_n$ be the successors of I
  and $s'_1, \ldots, s'_m$ be the successors of I' in
  I and I' are structurally similar
  and $\texttt{path}(f', \varphi, s_i, s'_i)$ for $i = 1, \ldots, n$

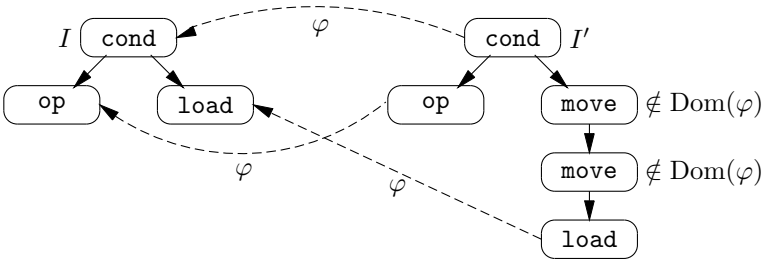
An RTL instruction I is *structurally similar* to an LTL instruction I' if they are identical modulo changes of successors and replacement of registers by locations, or if I is an `op` or `load` and I is a `nop` (dead code elimination). Table 1

Instruction I	Instruction I'	Condition
$\texttt{nop}(s)$	$\texttt{nop}(s')$	
$\texttt{op}(op, \vec{x}, x, s)$	$\texttt{op}(op, \vec{\ell}, \ell, s')$	if $\vec{x} \sim \vec{\ell}$ and $x \sim \ell$
$\texttt{op}(op, \vec{x}, x, s)$	$\texttt{nop}(s')$	
$\texttt{load}(\kappa, mode, \vec{x}, x, s)$	$\texttt{load}(\kappa, mode, \vec{\ell}, \ell, s')$	if $\vec{x} \sim \vec{\ell}$ and $x \sim \ell$
$\texttt{load}(\kappa, mode, \vec{x}, x, s)$	$\texttt{nop}(s')$	
$\texttt{store}(\kappa, mode, \vec{x}, x, s)$	$\texttt{store}(\kappa, mode, \vec{\ell}, \ell, s')$	if $\vec{x} \sim \vec{\ell}$ and $x \sim \ell$
$\texttt{call}(\tau, id, \vec{x}, x, s)$	$\texttt{call}(\tau, id)$	
$\texttt{cond}(cond, \vec{x}, s_1, s_2)$	$\texttt{cond}(cond, \vec{\ell}, s'_1, s'_2)$	if $\vec{x} \sim \vec{\ell}$
$\texttt{return}(x)$	`return`	

Table 1. Structural similarity between RTL and LTL instructions

gives a more precise definition. The $\sim$ relation (pronounced "agree") between a variable x and a location ℓ means that x and ℓ agree in register class and in size. For example, a variable of class `int` and size 32 bits agrees with the x86 register `EAX` and the stack slot $S(0,4)$, but not with the register `AX` (wrong size) nor with the register `XMM0` (wrong class) nor with the stack slot $S(0,8)$ (wrong size).

Besides structural similarity, `check_instr` also verifies the consistency of the successors of the two instructions I and I'. Naively, if φ maps the program point of I' to the program point of I, one could expect that the i-th successor of I' is mapped to the i-th successor of I. In the example below, this is the case for the left successors of the `cond` instructions.



This is not always the case because of the fresh `move` instructions that can be inserted during register allocation. However, there must exist a (possibly empty) path from the i-th successor of I' to a CFG node that is mapped to the i-th successor of I. This path must consist of `move` instructions that are not in the domain of φ. (See example above, right successors of the `cond` instructions.) This condition is checked by the auxiliary predicate `path`:

$\texttt{path}(f', \varphi, p, p') =$
  `false` if the node p' was previously visited;
  `true` if $\varphi(p') = p$;
  $\texttt{path}(f', \varphi, p, s')$ if $p' \notin \mathrm{Dom}(\varphi)$ and $f'.\texttt{code}(p') = \texttt{op}(\texttt{move}, \_, \_, s')$;
  `false`, otherwise.

Besides calling `check_instr` on each pair (p, p') of matching program points, the structural check pass also verifies that the two functions f, f' agree in name and type signature, and that there exists a valid path (in the sense above) from the entry point of f' to a point that maps to the entry point of f. (Typically, this path corresponds to `move` instructions that shuffle the parameters of the function.)

$\texttt{check\_structure}(f, f', \varphi) =$
  $f.\texttt{name} = f'.\texttt{name}$ and $f.\texttt{typesig} = f'.\texttt{typesig}$
  and $\texttt{path}(f', \varphi, f.\texttt{entrypoint}, f'.\texttt{entrypoint})$
  and for each p, p' such that $\varphi(p) = p'$,
    $\texttt{check\_instr}(f, f', \varphi, p, p')$

There is one last family of structural checks that we omitted here: enforcement of architectural constraints on the uses of locations. In the case of a RISC load-store architecture, for instance, argument and result locations must be hardware registers for all instructions except `move` operations, for which one of the source and destination can be a stack slot, but not both. CISC architectures like the x86 tolerate stack slots as arguments or results of some operations, but impose other constraints such as the result location being identical to the first argument location in the case of two-address instructions. These checks can be performed either during validation or as part of a later compiler pass; we omit them for simplicity.

3.2 Dataflow Analysis

To show that the original RTL function f and the register-allocated LTL function f' compute the same results and have the same effects on memory, we use a dataflow analysis that associates to each program point p' of f' a set $E(p')$ of equations between variables and locations:

$$E(p') = \{x_1 = \ell_1; \ldots; x_n = \ell_n\}$$

The semantic meaning of these equations is that in every execution of the code, the value of x_i at point $\varphi(p')$ in f is equal to the value of ℓ_i at point p' in f'.

There are two ways to build and exploit these sets of instructions: the forward way and the backward way. For concreteness, assume that we have structurally-similar `op` instructions at points p' in f' and $p = \varphi(p')$ in f:

$$f.\texttt{code}(p) = \texttt{op}(op, \vec{x}, x, s) \qquad f'.\texttt{code}(p') = \texttt{op}(op, \vec{\ell}, \ell, s')$$

These instructions use $\vec{x}$ and $\vec{\ell}$ and define x and ℓ, respectively.

In the forward approach, we assume given a set E of variable-location equations that hold "before" points p, p'. We can then check that $\{\vec{x} = \vec{\ell}\} \subseteq E$. If so, we know that both `op` instructions are applied to the same argument values, and since the operator op is the same in both instructions, they will compute the same result value and store it in x and ℓ. To obtain the equations that hold "after" these instructions, we remove from E all equations invalidated by the parallel assignment to x and ℓ (see below for a discussion), then add the equation $x = \ell$.

In the backward approach, we are given a set E of equations that must hold "after" points p, p' for the rest of the executions of f, f' to produce identical results and effects. We first check that the assignment to x and ℓ performed by the two op instructions does not render unsatisfiable any of the equations in E. If this check succeeds, we can remove the equation $x = \ell$ from E, since it is being satisfied by the parallel execution of the two `op` instructions, then add the equations $\{\vec{x} = \vec{\ell}\}$, since these are necessary for the two `op` instructions to produce the same result value. This gives us the set of equations that must hold "before" points p, p'.

In this work, we adopt the backward approach, as it tends to produce smaller sets of equations than the forward approach, and therefore runs faster. (To build an intuition, consider a long, straight-line, single-assignment sequence of instructions: the forward approach produces sets whose cardinal grows linearly in the number of instructions, while the backward approach produces sets whose cardinal is only proportional to the length of the live ranges.)

Unsatisfiability and Overlap. We mentioned the need to check that assigning in parallel to x and ℓ does not render unsatisfiable any equation in a set E. An example of this situation is $E = \{y = \ell\}$ where $x \neq y$. The LTL-side *op* instruction overwrites ℓ with a statically-unknown value, while the RTL-side *op* instruction leaves y unchanged. Therefore, there is no way to statically ensure that $y = \ell$ after executing these two instructions. In register allocation terms, this situation typically occurs if the allocator wrongly assigned ℓ to both x and y, despite x and y being simultaneously live and not being copies of one another.

The determination of unsatisfiable equations is made more complicated by the fact that LTL locations can *overlap*, i.e. share some bits of storage. Two overlapping locations contain a priori different values, yet assigning to one changes the value of the other. Overlap naturally occurs with stack slots: for instance, the slots $S(0, 8)$ (eight bytes at offset 0) and $S(4, 4)$ (four bytes at offset 4) clearly overlap. Some processor architectures also exhibit overlap between registers. For example, on the x86 architecture, the 64-bit register `RAX` contains a 32-bit sub-register `EAX`, a 16-bit sub-register `AX`, and two 8-bit sub-registers `AL` and `AH`. All these registers overlap pairwise except `AL` and `AH`. In summary, for two locations ℓ_1 and ℓ_2 there are three mutually-exclusive possibilities:

- Equality (written $\ell_1 = \ell_2$): both locations always contain the same value.
- Disjointness (written $\ell_1 \perp \ell_2$): assigning to one location does not change the value of the other.
- Partial overlap (written $\ell_1 \# \ell_2$): the values of the locations are a priori different, yet assigning to one affects the value of the other.

For stack slots, we have the following definitions:

$$S(\delta_1, n_1) = S(\delta_2, n_2) \iff \delta_1 = \delta_2 \wedge n_1 = n_2$$
$$S(\delta_1, n_1) \perp S(\delta_2, n_2) \iff [\delta_1, \delta_1 + n_1) \cap [\delta_2, \delta_2 + n_2) = \emptyset$$

For registers, the precise definitions of $\perp$ depends on the target architecture.

Armed with these notions of overlap and disjointness, we can formally define the compatibility between a pair x, ℓ of destinations and a set of equations E, written $(x, \ell) \perp E$:

$$(x, \ell) \perp E \stackrel{\text{def}}{=} \forall (x' = \ell') \in E, \quad (x' = x \wedge \ell' = \ell) \vee (x' \neq x \wedge \ell' \perp \ell)$$

Note that if $(x, \ell) \perp E$ holds, assigning in parallel the same value to x and ℓ will satisfy the equation $x = \ell$ and preserve the satisfiability of all other equations appearing in E. (See lemma 2 in section 4.)

$\texttt{transfer}(f, f', \varphi, p', E) =$
  if $E = \top$ then $\top$ (1)
  else if $\varphi(p') = p$ then:
    if $f.\texttt{code}(p) = \texttt{nop}(\_)$ and $f'.\texttt{code}(p') = \texttt{nop}(\_)$: (2)
      E
    if $f.\texttt{code}(p) = \texttt{op}(\texttt{move}, x_s, x_d, \_)$ and $f'.\texttt{code}(p') = \texttt{nop}(\_)$: (3)
      $E[x_d \leftarrow x_s]$
    if $f.\texttt{code}(p) = \texttt{op}(\_, \vec{x}, x, \_)$ or $\texttt{load}(\_, \_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{nop}(\_)$: (4)
      if $(x = \_) \in E$ then $\top$ else E
    if $f.\texttt{code}(p) = \texttt{op}(\_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{op}(\_, \vec{\ell}, \ell, \_)$ (5)
    or $f.\texttt{code}(p) = \texttt{load}(\_, \_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{load}(\_, \_, \vec{\ell}, \ell, \_)$:
      if $(x, \ell) \perp E$ then $(E \setminus \{x = \ell\}) \cup \{\vec{x} = \vec{\ell}\}$ else $\top$
    if $f.\texttt{code}(p) = \texttt{store}(\_, \_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{store}(\_, \_, \vec{\ell}, \ell, \_)$: (6)
      $E \cup \{x = \ell\} \cup \{\vec{x} = \vec{\ell}\}$
    if $f.\texttt{code}(p) = \texttt{call}(\_, \_, \vec{x}, x, \_)$ and $f'.\texttt{code}(p') = \texttt{call}(\tau, \_)$: (7)
      if $(x, \texttt{result}(\tau)) \perp E$ and E does not mention caller-save locations
      then $(E \setminus \{x = \texttt{result}(\tau)\}) \cup \{\vec{x} = \texttt{arguments}(\tau)\}$
      else $\top$
    if $f.\texttt{code}(p) = \texttt{cond}(\_, \vec{x}, \_, \_)$ and $f'.\texttt{code}(p') = \texttt{cond}(\_, \vec{\ell}, \_, \_)$: (8)
      $E \cup \{\vec{x} = \vec{\ell}\}$
    if $f.\texttt{code}(p) = \texttt{return}(x)$ and $f'.\texttt{code}(p') = \texttt{return}$: (9)
      $\{x = \texttt{result}(f'.\texttt{typesig})\}$
  else if $p' \notin \mathrm{Dom}(\varphi)$ then:
    if $f'.\texttt{code}(p') = \texttt{op}(\texttt{move}, \ell_s, \ell_d, \_)$: (10)
      $E[\ell_d \leftarrow \ell_s]$

Fig. 1. The transfer function for backward dataflow analysis

The Transfer Function. In preparation for a backward dataflow analysis, we now define the transfer function $\texttt{transfer}(f, f', \varphi, p', E)$ that computes the set E' of equations that must hold "before" program point p' in order for the equations E to hold "after" point p'. Here, E and E' range over sets of equations plus the symbolic constant $\top$ denoting inconsistency, or in other words the fact that the analysis failed to validate the flow of data.

The transfer function is defined in figure 1. We now explain its various cases. First, inconsistency propagates up, therefore $E' = \top$ if $E = \top$ (case 1). Then, we discuss whether the instruction at p' in f' is mapped to a structurally-similar instruction at p in f (i.e. $\varphi(p') = p$) or is new (i.e. $p' \notin \mathrm{Dom}(\varphi)$).

If $\varphi(p') = p$, we examine the shape of the two similar instructions. For instructions that perform no definitions, such as `store` and `cond`, we simply add equations $\{x_i = \ell_i\}$ to E, where $x_1, \ldots, x_n$ are the uses of the RTL instruction and $\ell_1, \ldots, \ell_n$ those of the LTL instruction (cases 6 and 8). These equations must be satisfied "before" for the two instructions to behave the same.

For instructions that define a variable x or a location ℓ, such as `op` and `load`, we first check compatibility between (x, ℓ) and E, and return $\top$ if false; for in this case there is no way to ensure that the equations E will be satisfied after

the assignments to x and ℓ. Otherwise, we remove the equation $x = \ell$ because the execution of the two instructions will satisfy it, then add equations $\{x_i = \ell_i\}$ before the uses as in the case of `store` or `cond` instructions.

The cases of `call` and `return` instructions are similar, except that the uses and defs of these LTL instructions are not marked in the instructions (as in RTL), but are implicitly determined from a type signature. Therefore, the uses and defs of an LTL $\texttt{call}(\tau, \ldots)$ are respectively $\texttt{arguments}(\tau)$ and $\texttt{result}(\tau)$ (case 7), and the uses of an LTL `return` are $\{\texttt{result}(f'.\texttt{typesig})\}$ (case 9). Moreover, not all registers and stack slots are preserved across an LTL function call, but only those marked as callee-save by the application binary interface used. The `call` case therefore returns $\top$ if the set E of equations "after" contains any equation $x = \ell$ where ℓ is caller-save: since the value of ℓ after the call is unpredictable, this equation cannot be satisfied.

Two cases remain that correspond to RTL instructions that were eliminated (turned into `nop`) during register allocation. Case 3 corresponds to one step of coalescing: a `move` instruction from x_s to x_d was eliminated because x_s and x_d were assigned the same location. In this case, any equation $x_d = \ell$ holds "after" provided that $x_s = \ell$ holds "before"; and any equation $x = \ell$ with $x \neq x_d$ holds after if only if it holds before. Therefore, the set E' of equations "before" is

$$E[x_d \leftarrow x_s] \stackrel{\text{def}}{=} \{(x_s = \ell) \mid (x_d = \ell) \in E\} \cup \{(x = \ell) \mid (x = \ell) \in E \wedge x \neq x_d\}$$

Case 4 corresponds to dead code elimination: an `op` or `load` instruction was removed because its destination variable x is not used later. We check that this is the case by making sure that no equation $x = \ell$ for some ℓ occurs in E, returning E if so and $\top$ if not.

Finally, let us consider the case $p' \notin \mathrm{Dom}(\varphi)$, indicating that the instruction at p' was inserted during register allocation. By our assumptions on what an allocator is allowed to do, this new LTL instruction must be a move (case 10). Let ℓ_s be its source and ℓ_d its destination. By a similar reasoning as in case 3, an equation $x = \ell_d$ is satisfied after the move if $x = \ell_s$ is satisfied before. Moreover, the move preserves satisfiability of any equation $x = \ell$ such that $\ell \perp \ell_d$. However, equations $x = \ell$ where $\ell \# \ell_d$ are not satisfiable because of overlap. The set E' of equations before point p' is, therefore:

$$\begin{aligned} E[\ell_d \leftarrow \ell_s] &= \top \text{ if there exists } (x = \ell) \in E \text{ such that } \ell \# \ell_d \\ E[\ell_d \leftarrow \ell_s] &= \{(x = \ell_s) \mid (x = \ell_d) \in E\} \cup \{(x = \ell) \mid (x = \ell) \in E \wedge \ell \perp \ell_d\} \\ &\qquad \text{otherwise} \end{aligned}$$

The Dataflow Analysis and Its Uses. Armed with the transfer function of figure 1, we set up backward dataflow equations of the form

$$E(p') = \bigcup \{\texttt{transfer}(f, f', \varphi, s', E(s') \mid s' \text{ successor of } p' \text{ in } f'\}$$

The unknowns are $E(p')$, the set of equations that must hold after each program point p' of the transformed function f'. By convention on $\top$, we take $\top \cup E =$

$E \cup \top = \top$. We then solve those equations by standard fixpoint iteration, starting with $E(p') = \emptyset$ for all points p'. (In our case, we reused a generic implementation of Kildall's algorithm provided by the CompCert compiler.)

Interestingly, this dataflow analysis generalizes liveness analysis, in the following sense: if $\{x_1 = \ell_1; \ldots; x_n = \ell_n\}$ are the equations "after" inferred at a program point p' mapped to p by φ, then the first projection $\{x_1, \ldots, x_n\}$ is the set of variables live in the original function f after point p and the second projection $\{\ell_1, \ldots, \ell_n\}$ is the set of locations live in the transformed function f' after point p'.

The validator then considers the set E_0 of equations "before " the function entry point:

$$E_0 \stackrel{\text{def}}{=} \texttt{transfer}(f, f', \varphi, f'.\texttt{entrypoint}, E(f'.\texttt{entrypoint}))$$

If $E_0 = \top$, an unprovable equation was encountered at some reachable instruction; validation therefore fails. Otherwise, we need to make sure that the equations in E_0 always hold. The only variable-location equations that hold with certainty are those between the RTL function parameters and the corresponding LTL locations:

$$E_{params} \stackrel{\text{def}}{=} \{f.\texttt{params} = \texttt{parameters}(f'.\texttt{typesig})\}$$

The validator could, therefore, check that $E_0 \subseteq E_{params}$ and signal an error otherwise. However, this check is too strong for C programs: it amounts to imposing Java's "definite assignment" rule. Indeed, $E_0 \subseteq E_{params}$ implies that all variables live at the beginning of the RTL function are parameters of this function. This is not always the case in RTL code generated from valid C functions such as:

```
int f(int x) {
  int y;
  if (x != 0) y = 100 / x;
  if (x != 0) return y; else return -1;
}
```

Here, the local variable `y` is live at the beginning of `f`, yet the function is semantically well-defined. Performed on the corresponding RTL code and a correct LTL register allocation of this code, the dataflow analysis of our validator produces an E_0 containing the equation $\texttt{y} = \ell$ for some ℓ. (This equation arises from the use of `y` in the "then" branch of the second "if", combined with the lack of a definition of `y` in the "else" branch of the first "if".)

How, then, can we avoid rejecting such correct codes at validation time? We take advantage of two very reasonable assumptions:

1. The semantics of RTL, like that of C, states that a program has undefined behavior if at run-time it uses the value of an undefined variable.
2. When establishing the correctness of a run of register allocation via validation, we are only interested in RTL programs that have well-defined behavior. For source programs with undefined behaviors, the register allocator can

produce arbitrary code. (Most compilers take this "garbage in, garbage out" view of optimization.)

Now, an equation $x = \ell$ at a program point where x is guaranteed to be uninitialized can safely be considered as always satisfied: since the RTL program has well-defined semantics, it is not going to use the value of x before defining it, therefore the actual value of x does not matter, and we might just as well assume that it matches the value of ℓ in the LTL code. The check performed by the validator on the initial equations E_0 is, therefore,

$$E_0 \cap f.\texttt{params} \subseteq E_{params}$$

where the intersection $E \cap X$ between a set of equations E and a set of RTL variables X is defined as

$$E \cap X \stackrel{\text{def}}{=} \{(x = \ell) \mid (x = \ell) \in E \wedge x \in X\}$$

3.3 The Validation Algorithm

Combining the definitions of sections 3.1 and 3.2, we obtain the main validation function:

$\texttt{check\_function}(f, f', \varphi) =$
  if $\texttt{check\_structure}(f, f', \varphi) = \texttt{false}$, return `false`
  compute the solutions $E(p')$ of the dataflow equations
    $E(p') = \bigcup\{\texttt{transfer}(f, f', \varphi, s', E(s') \mid s' \text{ successor of } p' \text{ in } f'\}$
  let $E_0 = \texttt{transfer}(f, f', \varphi, f'.\texttt{entrypoint}, E(f'.\texttt{entrypoint}))$
  check $E_0 \neq \top$ and $E_0 \cap f.\texttt{params} \subseteq \{f.\texttt{params} = \texttt{parameters}(f'.\texttt{typesig})\}$

Typically, this validator is combined with an untrusted implementation of a register allocator `regalloc`, as follows:

$\texttt{validated\_regalloc}(f) =$
  let $(f', \varphi) = \texttt{regalloc}(f)$ in
  if $\texttt{check\_function}(f, f', \varphi)$ then return f' else abort compilation

4 Soundness Proof

There are two properties of interest for a translation validator. One is *soundness*: if the validator says "yes", the transformed code behaves identically to the source code. The other is *relative completeness*: the validator never raises a false alarm; in other words, it accepts all valid instances of the code transformation considered. The completeness property is, necessarily, relative to a limited class of program transformations such as those listed in section 2.3: otherwise, validation would boil down to checking semantic equivalence between two arbitrary programs, which is undecidable.

We have formally proved the soundness of the validation algorithm presented in section 3. The proof was mechanized using the Coq proof assistant, bringing near-absolute confidence. This section gives a simplified sketch of this soundness proof. Relative completeness is difficult to even state formally, so we did not attempt to prove it. Testing shows no false alarms (see section 5). We conjecture that our validator is complete for all program transformations that can only rename variables, insert move operations, and delete operations and loads, but treat as uninterpreted (and therefore preserve) all other computations.

4.1 Dynamic Semantics

In preparation for stating and proving soundness, we need to give formal semantics to the RTL and LTL languages. The full semantics of RTL is described in [11, section 6]. Here, for simplicity, we outline the semantics of the fragment of RTL that excludes function calls and returns, and therefore is given relative to a single function f.

The semantics is presented in small-step style as a transition relation $\rightarrow$ between execution states. States are triples (p, e, m) where p is the current program point (a CFG node), e is a partial map from variables to values, and m is the memory state: a partial map from (pointer, memory quantity) pairs to values. Values are the discriminated union of integers, floating-point numbers, and pointers. (In the full semantics of RTL, the state contains additional components such as the function currently executing and an abstract call stack.)

$$\frac{f.\texttt{code}(p) = \texttt{nop}(s)}{(p, e, m) \rightarrow (s, e, m)} \qquad \frac{f.\texttt{code}(p) = \texttt{op}(op, \vec{x}, x, s) \quad \overline{op}(e(\vec{x})) = v}{(p, e, m) \rightarrow (s, e[x \leftarrow v], m)}$$

$$\frac{f.\texttt{code}(p) = \texttt{load}(\kappa, mode, \vec{x}, x, s) \quad \overline{mode}(e(\vec{x})) = v_{ad} \quad m(v_{ad}, \kappa) = v}{(p, e, m) \rightarrow (s, e[x \leftarrow v], m)}$$

$$\frac{f.\texttt{code}(p) = \texttt{store}(\kappa, mode, \vec{x}, x, s) \quad \overline{mode}(e(\vec{x})) = v_{ad} \quad m[(v_{ad}, \kappa) \leftarrow e(x)] = m'}{(p, e, m) \rightarrow (s, e, m')}$$

$$\frac{f.\texttt{code}(p) = \texttt{cond}(cond, \vec{x}, s_1, s_2) \quad s = \begin{cases} s_1 & \text{if } \overline{cond}(e(\vec{x})) = \texttt{true} \\ s_2 & \text{if } \overline{cond}(e(\vec{x})) = \texttt{false} \end{cases}}{(p, e, m) \rightarrow (s, e, m)}$$

Fig. 2. Transition rules for the simplified semantics of RTL

The transition relation $\rightarrow$ between states is defined by the rules of figure 2. The rules discriminate on the instruction at the current program point p, then update the three components of the state accordingly. The partial functions $\overline{op}$, $\overline{mode}$ and $\overline{cond}$ are the semantic interpretations of operators, addressing modes

and conditions as functions over values. We make no assumptions about these interpretations, except that the `move` operation is the identity: $\overline{\texttt{move}}(v) = v$. The notation $e[x \leftarrow v]$ stands for the variable environment mapping x to v and all other variables y to $e(y)$. The initial state is $(f.\texttt{entrypoint}, [f.\texttt{params} \mapsto \vec{v}_{args}], m_{init})$ where $\vec{v}_{args}$ are the values of the arguments given to function f. The final state is (p, e, m) where p points to a `return` instruction.

The semantics of LTL is essentially isomorphic to that of RTL, at least for the fragment considered here. (The full LTL treats function calls somewhat differently from RTL, to reflect the passing of function arguments and results through conventional locations.) LTL states are triples (p', e', m') of a program point p' in f', an environment e' mapping locations to values, and a memory state m'. The main difference between RTL and LTL is the update $e'[\ell \leftarrow v]$ of a location ℓ by a value v: it sets ℓ to v, but as collateral damage is also sets overlapping locations $\ell' \# \ell$ to unspecified values:

$$\begin{aligned} e'[\ell \leftarrow v](\ell) &= v \\ e'[\ell \leftarrow v](\ell') &= e'(\ell') \text{ if } \ell' \perp \ell \\ e'[\ell \leftarrow v](\ell') &\quad \text{is unspecified if } \ell' \# \ell \end{aligned}$$

Note that the values of stack locations $S(\delta, n)$ are stored in the location environment e' and not in the memory state m'. This simplifies the proof. A separate proof, detailed in [11, section 12], shows that accesses to stack locations can later be reinterpreted as memory loads and stores within the activation record.

4.2 Equation Satisfaction

The crucial invariant of the soundness proof is the following: whenever control reaches point p' in the LTL function f' and matching point $\varphi(p')$ in the RTL function f, the corresponding environments e and e' satisfy the equations E "before" point p' inferred by the validator. Equation satisfaction is written $e, e' \models E$ and defined as

$$e, e' \models E \stackrel{\text{def}}{=} \forall (x = \ell) \in E,\ x \in \mathrm{Dom}(e) \Longrightarrow e(x) = e'(\ell)$$

This predicate enjoys nice properties that are keys to the soundness proof. First, satisfaction implies that the argument values to matching RTL and LTL operations are identical. (This lemma is used in the parts of the soundness proof that corresponds to cases 3, 6, 7, 8 and 9 of the transfer function.)

Lemma 1. *If $e, e' \models E \cup \{\vec{x} = \vec{\ell}\}$ and $e(\vec{x})$ is defined, then $e'(\vec{\ell}) = e(\vec{x})$.*

Second, satisfaction is preserved by several kinds of parallel or unilateral assignments. (For each lemma we indicate the corresponding cases of the transfer function.)

Lemma 2 (Parallel assignment – cases 5 and 7). *If $e, e' \models E \setminus (x = \ell)$ and $(x, \ell) \perp E$ then $e[x \leftarrow v], e'[\ell \leftarrow v] \models E$*

Lemma 3 (RTL assignment to a dead variable – case 4). *If* $e, e' \models E$ *and* $(x = \_) \notin E$ *then* $e[x \leftarrow v], e' \models E$

Lemma 4 (Coalesced RTL move – case 3). *If* $e, e' \models E[x_d \leftarrow x_s]$ *then* $e[x_d \leftarrow e(x_s)], e' \models E$

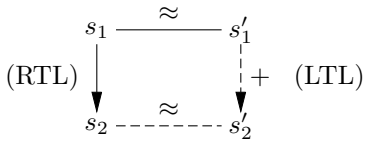
Lemma 5 (Inserted LTL move – case 10). *If* $E[\ell_d \leftarrow \ell_s] \neq \top$ *and* $e, e' \models E[\ell_d \leftarrow \ell_s]$ *then* $e, e'[\ell_d \leftarrow e'(\ell_s)] \models E$

Finally, satisfaction holds in the initial states, taking $\vec{x} = f.\texttt{params}$ and $\vec{\ell} = \texttt{parameters}(f'.\texttt{typesig})$ and $\vec{v}$ to be the values of the function parameters.

Lemma 6. *If* $E \cap \vec{x} \subseteq \{\vec{x} = \vec{\ell}\}$*, then for any* e' *such that* $e'(\vec{\ell}) = \vec{v}$*, we have* $[\vec{x} \mapsto \vec{v}], e' \models E$*.*

4.3 Forward Simulation

The soundness proof takes the form of a forward simulation diagram relating one transition in the RTL code to one or several transitions in the LTL code, starting and ending in matching states. (The "or several" part corresponds to the execution of move instructions inserted during register allocation.)



The relation $\approx$ between RTL and LTL states is defined as follows:

$$(p, e, m) \approx (p', e', m') \stackrel{\text{def}}{=} \varphi(p') = p \ \wedge \ e, e' \models \texttt{transfer}(f, f', \varphi, p', E(p')) \ \wedge \ m = m'$$

That is, the program points must match according to the φ mapping; the variable and location environments must satisfy the dataflow equations "before" point p'; and the memory states are identical.

Theorem 1 (Forward simulation). *Assume that* $\texttt{check\_function}(f, f', \varphi) = \texttt{true}$*. Let* $E(p')$ *be the solutions to the dataflow equations. If* $s_1 \rightarrow s_2$ *and* $s_1 \approx s'_1$*, there exists* s'_2 *such that* $s'_1 \stackrel{+}{\rightarrow} s'_2$ *and* $s_2 \approx s'_2$*.*

The proof of this theorem proceeds in two steps. First, we show that the LTL code can make one transition from s'_1 to some state (p', e', m') that does not necessarily match s_2 (because $\varphi(p')$ can be undefined) but is such that $e, e' \models \texttt{transfer}(f, f', \varphi, p', E(p'))$. This part of the proof proceeds by case analysis on the RTL and LTL instructions pointed to by s_1 and s'_1, and exercises all cases of the structural checks and the transfer function except the `path` check and case 10. Then, the following lemma shows that we can extend this LTL transition with zero, one or several transitions (corresponding to executions of inserted move instructions) to reach a state matching s_2.

Lemma 7 (Execution of inserted moves). *Assume* $\texttt{path}(f', \varphi, p, p') = \texttt{true}$ *and* $e, e' \models \texttt{transfer}(f, f', \varphi, p', E(p'))$. *Then, there exists* p'' *and* e'' *such that* $(p', e', m) \xrightarrow{*} (p'', e'', m)$ *and* $\varphi(p'') = p$ *and* $e, e'' \models \texttt{transfer}(f, f', \varphi, p'', E(p''))$.

From the forward simulation theorem 1, semantic preservation for whole programs (that is, agreement between the observable behaviors of the source RTL code and transformed LTL code) follows easily using the general results of [11, section 3.7].

5 Implementation and Experimental Results

We implemented the validation algorithm and a prototype register allocator within the CompCert verified compiler [9]. Like all other verified parts of this compiler, the validator is written directly in the Gallina specification language of the Coq proof assistant, in pure functional style. Sets of equations are implemented as persistent AVL trees, using the `FSet` standard library of Coq. This implementation supports insertion and removal of equations in $O(\log n)$ time, but the compatibility check $(x, \ell) \perp E$ requires an $O(n)$ traversal of the set E. Whether a better data structure could support compatibility check in logarithmic time is an open question.

The Gallina implementation of the validator lends itself immediately to program proof within Coq. Efficient Caml code is automatically generated from the Gallina code using Coq's program extraction facility. The generated Caml code is then linked with a register allocator hand-written in Caml.

The prototype register allocator we experimented with is a standard Chaitin-style graph coloring allocator, using George and Appel's iterated register coalescing algorithm to color the interference graph [10]. If some variables x were assigned stack slots and are used by instructions that demand a hardware register, spill and reload instructions to/from fresh temporary variables are introduced and register allocation is repeated. Two spilling strategies were experimented. The first simply inserts a reload before every use of a spilled variable and a spill after every definition. The second splits the live ranges of a spilled variable at every definition and every use, in the hope that reloaded values can stay in a register across several reloads in parts of the code where register pressure is low. (This is a less aggressive form of splitting than that considered by Appel and George [12].) Since we are targeting a register-rich architecture (the PowerPC), spilling occurs rarely. To stress the validator, we reduced the number of callee-save registers, forcing considerable spilling across function calls.

On the CompCert test suite, the validator performed as expected: it did not raise any false alarms, but found several mistakes in our implementation of the second spilling strategy. The compile-time overhead of the validator is very reasonable: validation adds 20% to the time taken by register allocation and 6% to the whole compilation time.

From a proof engineering viewpoint, the validator is a success. Its mechanized proof of correctness is only 900 lines of Coq, which is quite small for a 350-line

piece of code. (The typical ratio for Coq program proofs is 6 to 8 lines of proof per line of code.) In contrast, 4300 lines of Coq proof were needed to verify the register allocation and spilling passes of the original CompCert compiler. Even this earlier development used translation validation on a sub-problem: the George-Appel graph coloring algorithm was implemented directly in untrusted Caml code, then followed by a verified validator to check that the resulting assignment is a valid coloring of the interference graph. Later, Blazy, Robillard and Appel conducted a Coq proof of the graph coloring algorithm [13]. This is a fairly large proof: in total, more than 10000 lines of proof are needed to completely verify the original CompCert register allocation and spilling passes. In summary, the translation validation approach delivers a ten-fold reduction in the proof effort compared with the compiler verification approach, while providing soundness guarantees that are just as strong. Of course, the compiler verification approach offers additional formal guarantees: not just soundness, but also completeness (register allocation never fails at compile-time). In contrast, the verified validator approach cannot rule out the possibility of a spurious compile-time error.

6 Related Work

The idea of translation validation goes back at least to Samet's 1975 Ph.D. thesis [14]. It was rediscovered and popularized by Pnueli *et al.* ten years ago [1]. Some translation validators proceed by generation of verification conditions followed by model checking or automatic theorem proving [1, 15–17]; others rely on less powerful but cheaper and more predictable approaches based on symbolic evaluation and static analyses [2–5, 8, 18]. For another dividing line, some validators are general-purpose and apply to several compilation passes [2] or even to a whole compiler [3], while others are specialized to particular families of optimizations, such as software pipelining [15, 19, 18], instruction scheduling [5], partial redundancy elimination [4], or register allocation [8]. The present work falls squarely in the cheap, specialized, static analysis-based camp.

The earlier work most closely related to ours is that of Huang, Childers and Soffa [8]: a validator for register allocation that was prototyped within SUIF. Their validator proceeds by forward dataflow analysis and global value numbering. A nice feature of their validator, which ours lacks, is the production of meaningful explanations when an error is detected. On the other hand, their validation algorithm was not proved sound. Such a proof appears delicate because the semantic interpretation of global value numbers is difficult.

The general-purpose validators of Necula [2] and Rival [3] can also validate register allocation among other program transformations. They proceed by symbolic evaluation: variables and locations in the source and transformed code are associated symbolic expressions characterizing their values, and these expressions are compared modulo algebraic identities to establish semantic equivalence. Symbolic evaluation is a very versatile approach, able to validate many program transformations. On the particular case of register allocation and spilling, it ap-

pears no more powerful, but more costly, than the specialized techniques used by Huang *et al.* and by us.

7 Conclusions and Future Work

The validation algorithm for register allocation and spilling presented in this paper is simple enough to be integrated in production compilers and efficient enough to be invoked on every compilation run. At the same time, the mechanically-checked proof of soundness brings considerable confidence in its results.

Our validator can be improved in several directions. One is to design a more efficient data structure to represent sets of equations. As mentioned in section 5, the simple representation of equation sets as AVL trees performs compatibility checks in linear time. A more sophisticated data structure might support logarithmic-time operations over equation sets.

Another direction is to introduce additional forms of equations to enable the validation of even more code transformations related to register allocation. For example, rematerialization of constants [20] could probably be validated if we were able to keep track of equations of the form $x = \mathit{constant}$. Likewise, the parts of the function prologue and epilogue that save and restore used callee-save registers to/from stack slots (currently treated in CompCert by a separate, verified pass) could be validated along with register allocation if we had equations of the form $\texttt{init}(r) = \ell$, where $\texttt{init}(r)$ is a symbolic constant denoting the value of callee-save register r on entrance to the function.

References

1. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorithms for Construction and Analysis of Systems, TACAS '98. Volume 1384 of Lecture Notes in Computer Science., Springer (1998) 151–166
2. Necula, G.C.: Translation validation for an optimizing compiler. In: Programming Language Design and Implementation 2000, ACM Press (2000) 83–95
3. Rival, X.: Symbolic transfer function-based approaches to certified compilation. In: 31st symposium Principles of Programming Languages, ACM Press (2004) 1–13
4. Tristan, J.B., Leroy, X.: Verified validation of Lazy Code Motion. In: Programming Language Design and Implementation 2009, ACM Press (2009) 316–326
5. Tristan, J.B., Leroy, X.: Formal verification of translation validators: A case study on instruction scheduling optimizations. In: 35th symposium Principles of Programming Languages, ACM Press (2008) 17–27
6. Coq development team: The Coq proof assistant. Software and documentation available at `http://coq.inria.fr/` (1989–2010)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer (2004)
8. Huang, Y., Childers, B.R., Soffa, M.L.: Catching and identifying bugs in register allocation. In: Static Analysis, 13th Int. Symp., SAS 2006. Volume 4134 of Lecture Notes in Computer Science., Springer (2006) 281–300

9. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009) 107–115
10. George, L., Appel, A.W.: Iterated register coalescing. ACM Transactions on Programming Languages and Systems **18**(3) (1996) 300–324
11. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4) (2009) 363–446
12. Appel, A.W., George, L.: Optimal spilling for CISC machines with few registers. In: Programming Language Design and Implementation 2001, ACM Press (2001) 243–253
13. Blazy, S., Robillard, B., Appel, A.W.: Formal verification of coalescing graph-coloring register allocation. In: European Symposium on Programming (ESOP 2010). Lecture Notes in Computer Science, Springer (2010) To appear.
14. Samet, H.: Automatically Proving the Correctness of Translations Involving Optimized Code. PhD thesis, Stanford University (1975)
15. Leviathan, R., Pnueli, A.: Validating software pipelining optimizations. In: Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2002), ACM Press (2006) 280–287
16. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: VOC: A methodology for translation validation of optimizing compilers. Journal of Universal Computer Science **9**(3) (2003) 223–247
17. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A translation validator for optimizing compilers. In: Computer Aided Verification, 17th International Conference, CAV 2005. Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 291–295
18. Tristan, J.B., Leroy, X.: A simple, verified validator for software pipelining. In: 37th symposium Principles of Programming Languages, ACM Press (2010) Accepted for publication, to appear.
19. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Programming Language Design and Implementation 2009, ACM Press (2009) 327–337
20. Briggs, P., Cooper, K.D., Torczon, L.: Rematerialization. In: Programming Language Design and Implementation 1992, ACM Press (1992) 311–321