Static Validation of Register Allocation

Fernando Magno Quintão Pereira

UCLA
University of California, Los Angeles

Abstract. Testing the implementation of register allocation algorithms is a difficult and time-consuming activity. Errors in the code generated by the register allocator tend to manifest sporadically, often many instructions past the problematic point. In this paper we formalize the concept of semantic consistency of register allocation directives. Also, we present a collection of data flow algorithms that statically validates the translation between two representations of a program: the first has an unbounded number of temporary variables, whereas the second can only use a limited number of registers. The need for this validation infrastructure emerged during the development of actual register allocators for the Strong ARM architecture, and its availability greatly improved our ability to trace errors in register allocation settings.

1 Introduction

The *core register allocation problem* [18] consists in mapping a program m, that can use a virtually unbounded number of temporary variables, to a program f that contains a finite, and possibly small number of registers. We call this mapping a set of *register allocation directives*. Informally, we say that the allocation directives are semantically consistent if m and f always produce the same observable results when interpreted with equal inputs.

Register allocation is a complex and error prone activity. The validation of a register allocation algorithm is a especially difficult task. Errors can surface in non-trivial ways, possibly many instructions after the erroneous instruction. Moreover, the low-level nature of the machine code, and also its big size, makes a visual inspection of the register allocator's output a tedious, and most often, non-effective task. Therefore, when implementing register allocation algorithms, developers can greatly benefit from tools that help them to find errors in the code been produced.

In this paper we present a collection of data flow algorithms that statically validate the translation between two intermediate representations of a program. The first has access to an unbounded number of variables. The second can only use a finite, and generally small, number of registers. Our translation validation infrastructure certifies that this mapping is semantically consistent, or, if this is not the case, it lists the sources of inconsistencies. The proposed verification framework differs from previous works because it does not depend on a particular implementation of a register allocator, e.g. graph coloring, ILP, linear scan, etc.

The algorithms and proofs presented in this paper verify the correctness of the output of a register allocation algorithm, not the algorithm itself. Moreover, our algorithms do not target any particular computer architecture.

The remainder of this paper is organized as follows: Section 2 presents other approaches for verifying the correctness of register allocation directives. Section 3 formally introduces the concept of semantic consistency of register allocation directives. Section 4 describes the algorithms that we have used to verify semantic consistency. Finally, Section 5 concludes the paper.

1.1 Motivations

Our interest for the proposed validation infrastructure arouse during the implementation of three register allocation algorithms: the usage based count [6], the iterated register coalescing [8], and the non-iterative chordal coloring algorithm [17]. Such algorithms have been plugged into `gcc` by means of a register allocation framework known as Ralf (Register ALlocation Framework) [12]. A comparison between these algorithms, and many others, can be found in [12].

`Gcc` has a modular structure that allows to plug and test different register allocation algorithm; however, it does not check if the mapping between temporaries and registers is correct. Ralf is a framework specifically designed to facilitate the implementation and testing of register allocation algorithms. It is essentially a new layer built on top of `gcc`. Ralf hides all the internal complexity of the `gcc` framework, because its input and output data are ASCII files.

In addition of enhancing `gcc` with a more amiable interface, Ralf implements a static verification phase that checks the output produced by the register allocator. Examples of properties verified by Ralf are: (a) every temporary must have a register allocated to it; (b) if a store instruction i is produced to save the value of a spilled register r, then r must be alive at i; (c) two different temporaries used at the same instruction cannot be allocated to the same machine register. Although the static verification performed by this framework is an improvement on `gcc`, it is not complete. For instance, Ralf does not report as an error the possibility of two live ranges of the same temporary to reach a joint point allocated to different machine registers.

2 Related Works

Although translation-validation is a well know topic in the compiler literature, little has been said about the verification of register allocators. To the best of our knowledge, the first explicit description of correctness of register allocation was given by Leroy [10]. However, correctness proofs for specific algorithms have been described before. For example, Naik and Palsberg [11] have proved the correctness of the ILP-based register allocator of Appel and George [3]. As another example, Ohori [16] has designed a register allocation algorithm as a series of proof transformations. These transformations specify how different representations of a program can be sequentially reached, until machine code is generated.

Necula [14] presents a translation validation infrastructure for the `gcc` compiler that includes register allocation. One of the improvements of Necula's scheme over other approaches is to treat the memory address of spilled registers as variables. However, this technique relies on specific characteristics of the `gcc` compiler, such as the addressing modes.

Leroy [10] formally describes a technique to validate the output of graph coloring based register allocation algorithms. Basically, if the interference graph contains a pair of adjacent temporaries allocated to the same register, the verifier emits an error, otherwise it assumes that the code generated is correct. Similar approaches have been adopted by Andersson [2] and Pereira et al [17]. This technique only serves the graph-coloring paradigm, and, at least in [17], it was not able to detect situations in which a dead definition overwrite a live register.

Another way to guarantee that register allocation maintain certain properties is to augment the machine code been produced with annotations. Such annotations describe invariant properties that must always be true during the program execution (e.g.: the value of some variable is always greater than 0). In general the annotations are propagated across the intermediate representations used by the compiler, until the machine code. Examples of this approach are the type annotations used by Agat [1]. Types increase the readability of the assembly code, and ensure that data are not used in improper ways, e.g. a register holding an integer value is always used as an integer. The Touchstone [15] compiler goes a step further, and accommodates more general annotations, such as memory bounds. The designers of Touchstone argue that if the properties specified by the annotations are certified by a theorem prover for the intermediate representation, than they are also true for the machine code. Although these annotations reduce the possibility of errors at the assembly level, they do not guarantee that the output of the register allocator is completely correct. It is still possible that live temporaries be overwritten, or that live ranges representing the same temporary reach a joint point assigned to different registers.

3 Semantic Consistency of Register Allocation

In this section we (i) define the types and functions used in our formal descriptions, (ii) introduce the intermediate representations used during the register allocation process. (iii) define the concept of semantic consistency of register allocation directives, and (iv) show how semantic consistency can be broken by incorrect allocation directives.

3.1 Instructions, Registers and Temporary Variables

In order to formally describe the properties and algorithms presented in this paper we define types for registers, temporary variables and instructions, plus a set of functions that manipulate these data.

Registers – $R : \{\bot, r_0, r_1, \ldots, r_k\}$ – This data type represents machine registers. It contains a finite number of elements, because the number of machine registers is limited. For instance, for the Strong ARM processor, $k = 12$, and for the MIPS instruction set $k = 32$. The unit value $\bot$ is used to designate the location of an undefined temporary.

- **isCallerSave**: $R \to \{true, false\}$ – This boolean function will return *true* if the register given as a parameter is *caller save*. A caller save register may be overwritten during the execution of a function call, and, thus, must be saved by the caller.

Temporary variables – $T : \{t_0, t_1, \ldots\} \cup R$ – This data type represent temporary variables. This type contains an infinite number of values, given that the core register allocation problem does not imposes limits on the number of variables in the pre-allocation code. Because pre-colored registers can appear in this code, we let the type of machine registers be a subtype of temporary registers.

- **loc**: $T \to R$ – this function determines the register to which a temporary has been allocated.
- **temp**: $R \to T$ – this function determines the temporary that a register is currently holding.

Instructions – $I : \{\mathbf{error}, i_0, i_1, \ldots\}$ – an instruction $i = (o, d, u, s, p, in, out)$ is a 7-tuple where d is the set of registers defined at i, u is the set of registers used at i, in is the set of registers alive before i is executed, and out is the set of registers alive after i's execution. The predecessor instructions are given by p, and s contains the successor instructions. Finally, the first field o represents the instruction opcode: **cl** for calls, and **op** for all the other types of instructions.

- **def**: $I \to$ set of T – If $i = (o, d, u, s, p, in, out)$, then $\mathbf{def}(i) = d$.
- **use**: $I \to$ set of T – If $i = (o, d, u, s, p, in, out)$, then $\mathbf{use}(i) = u$.
- **in**: $I \to$ set of T – If $i = (o, d, u, s, p, in, out)$, then $\mathbf{in}(i) = in$.
- **out**: $I \to$ set of T – If $i = (o, d, u, s, p, in, out)$, then $\mathbf{out}(i) = out$.
- **pred**: $I \to$ set of I – If $i = (o, d, u, s, p, in, out)$, then $\mathbf{pred}(i) = p$.
- **succ**: $I \to$ set of I – If $i = (o, d, u, s, p, in, out)$, then $\mathbf{succ}(i) = s$.
- **isCall**: $I \to \{true, false\}$ – If $i = (\mathbf{cl}, d, u, s, p, in, out)$, then $\mathbf{isCall}(i) = true$; otherwise, $\mathbf{isCall}(i) = false$.

We define the following additional functions:

- **pc**: I – This nullary function represents the program counter: a register used to keep track of the instruction that is been currently executed.
- **any**: set of $\top \to \top$ – The generic function $\mathbf{any}(s)$ chooses non-deterministically an element among the elements of s.
- **input**: $\mathcal{F} \to$ set of T – given a program f, let i be the first instruction of f. $\mathbf{input}(f) = \mathbf{in}(i)$. We define the type of f in the next section.

3.2 Intermediate Representation and Machine Code

The register allocation process consists in translating a program m, that uses a unbounded number of variables, into a program f, that can use only a finite number of registers. Let f be an instance of a language $\mathcal{F}$, and let m be an instance of $\mathcal{M}$. Following the traditional compiler's nomenclature [10], we represent $\mathcal{M}$ as a *RTL* like language, and we describe $\mathcal{F}$ as a *LTL* like language [1]. Figures 1 (a) and (b) outline the abstract syntax of both languages. The only syntactical difference between these representations is that in $\mathcal{F}$ variables are represented by pairs (t, r), where t is a temporary variable, and r is the register assigned to it by the register allocator. Figure 2 (a) shows a typical MIPS procedure, with pseudo-registers instead of machine registers, and Figure 2 (b) outlines its RTL representation. We chose the SSA representation because it allows us to illustrate all the register allocation directives. However, the SSA-form is not a requirement for the algorithms presented in this paper.

a)

$\mathcal{M} ::= i*$
$i ::= \langle op, \mathbf{def}, \mathbf{use}, \mathbf{in}, \mathbf{out}, \mathbf{succ}, \mathbf{pred} \rangle$
$op ::= \mathbf{op} \mid \mathbf{cl}$
$\mathbf{def}, \mathbf{use}, \mathbf{in}, \mathbf{out} ::= t*$
$\mathbf{succ}, \mathbf{pred} ::= i*$

b)

$\mathcal{F} ::= i*$
$i ::= \langle op, \mathbf{def}, \mathbf{use}, \mathbf{in}, \mathbf{out}, \mathbf{succ}, \mathbf{pred} \rangle$
$op ::= \mathbf{op} \mid \mathbf{cl}$
$\mathbf{def}, \mathbf{use}, \mathbf{in}, \mathbf{out} ::= a*$
$\mathbf{succ}, \mathbf{pred} ::= i*$
$a ::= (t, r)$

c)

$\mathcal{D} ::= \mathbf{defReg\ useReg\ storeSpill\ loadSpill\ parallelCopy}$
$\mathbf{defReg}, \mathbf{useReg}, \mathbf{storeSpill}, \mathbf{loadSpill} ::= ra*$
$\mathbf{parallelCopy} ::= (i_{origin}\ i_{destiny}\ t_{destiny}\ r_{destiny}\ t_{origin}\ r_{origin})*$
$ra ::= i\ t\ r$

t ranges over T, r ranges over R, i ranges over I

Fig. 1. (a) $\mathcal{M}$: the register transfer language. (b) $\mathcal{F}$: the location transfer language. (d) $\mathcal{D}$: the language for the specification of register allocation directives.

We define a register allocator as a function $\mathcal{A}$ such that, given a RTL program m, and a number of register K, $\mathcal{A}(m, K) = d$. The output d represents a set of *register allocation directives*. Let $\mathcal{D}$ be the language that describes these directives. The grammar in Figure 1 (c) gives the abstract syntax of $\mathcal{D}$. Except for parallel copies, our representation is a subset of the one used in the Ralf framework [12]. It is independent of any particular target architecture. This independence enhances the reusability of the proposed translation validation scheme; however, it prevents us of handling particular implementations of register allocators. Although simple, our representation is expressive enough to accommodate a wide variety of register allocation algorithms. For example, it can describe the

[1] RTL stands for Register Transfer Language, and LTL stands for Location Transfer Language

output of six of the seven register allocators described in [12]. The only omission is due to one register allocator that relies on the stack layout in order to minimize the traffic between the register bank and the memory [13].

Figure 2 (c) exhibits a possible set of register allocation directives for the program given in Figure 2 (b), assuming that the target architecture has only 2 registers available. $\mathcal{D}$ does not contain references to pre-colored registers, because they are already bound to a machine register. Allocation directives consist of five types of clauses. `DefReg` describes, for each instruction, to which register a defined temporary should be mapped. For instance, temporary t_{14b} should be allocated to register r_0 when defined in instruction (6). `UseReg` specifies, for each instruction, the locations of the used temporaries. `StoreSpill` defines after which instructions the values of spilled temporaries should be stored. For example, temporary t_{14a}, located in register r_1 must be sent to memory immediately after instruction (3) is executed. `LoadSpill` determines before which instructions the spilled temporaries must be loaded. Again in Figure 2 (c), temporary t_{14a} must be loaded into register r_1 immediately before instruction (5) is executed. Finally, the last clause permits to specify parallel copies. An entry such as i j (t_d, r_d) (t_o, r_o) would cause the value of r_o to be moved to r_d at the edge between instructions i and j. All the copies inserted in the same edge are executed in parallel, after any possible store, and before any load directive.

A parallel copy, such as $(t_1, \ldots, t_n) = (u_1, \ldots, u_n)$ performs all the assignments $t_i = u_i$ simultaneously; hence, it avoids interferences between the used and defined variables. Parallel copies are used by some recent register allocation algorithms that rely on the SSA [2] transformation in order to obtain better results [17, 5, 9]. The core register allocation problem can be solved in polynomial time for programs in strict SSA-form [9]. However, this computational time is only possible if the compiler can use parallel copies when converting the SSA-form program into machine code [18]. Parallel copies can be implemented in most computer architectures by means of `xor` instructions, or with an extra register, as shown by Hack et al [9].

Given a RTL program m, and a set of allocation directives d, the corresponding LTL representation f can be obtained by the algorithms **addNewInstructions** and **replaceRegisters**. The first algorithm, shown in Figure 4, adds to m new instructions to accommodate the loads, stores, and parallel copies found in d. In addition, it produces a new set d' of allocation directives, where the **load**, **store** and **parallelCopy** clauses of d have been replaced with **use** and **def** clauses. Figure 3 (a) contains the LTL representation of our example program. **ReplaceRegisters** uses d' in order to replace temporary registers t by pairs (t, r) denoting the register assignment.

In order to specify allocation directives, we define the function $\mathbf{dir} : \mathcal{D} \rightarrow \mathit{Instruction} \rightarrow \mathit{Temporary} \rightarrow \{\mathit{in}, \mathit{out}\} \rightarrow \mathit{Register}$. Figure 3 (b) shows the implementation of **dir** for the program given in Figure 3 (a). We use the set $\{\mathit{in}, \mathit{out}\}$ in order to distinguish uses (in) from definitions (out). For instance, $\mathbf{dir}(d', i_1, t_{13}, \mathit{out}) = r_0$ means that, according to the set of allocation directives

[2] SSA stands for Single Static Assignment

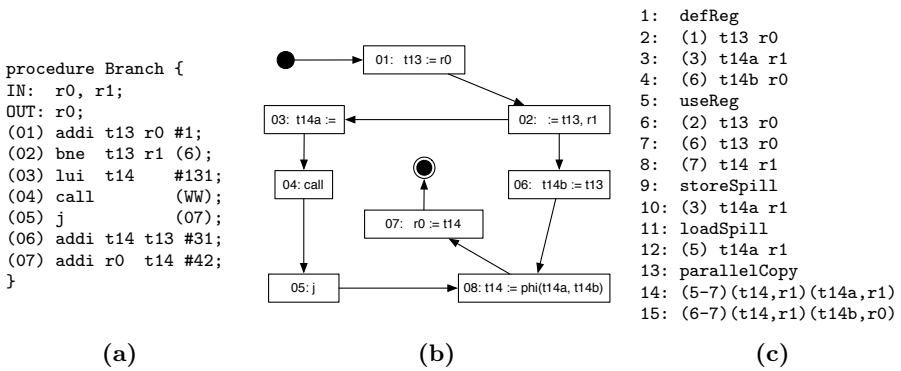


Fig. 2. (a) A program $\mathcal{P}$ represented as a set of MIPS instructions. (b) The RTL representation of $\mathcal{P}$, in SSA-form. (c) A possible set of allocation directives for $\mathcal{P}$.

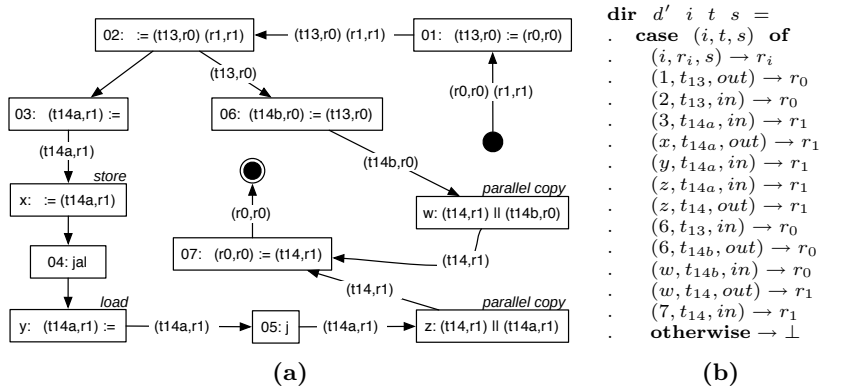


Fig. 3. (a) f: The binding of temporaries to machine registers. (b) The function **dir** that represents the register assignment given in (a).

d', the register allocator has determined that temporary t_{13} must be allocated to register r_0 when defined in instruction i_1. The location of a pre-colored register is always the machine register that this pre-colored represents.

3.3 Semantic Consistency of Register Allocation Directives

A program is a specification of how the state of the computer changes. For our purposes, the state of the machine is determined by the set of values stored in the register bank, plus the value stored in the program counter. We call the set of reachable states $\Sigma = (\mathbf{loc} \times \mathbf{temp} \times \mathbf{pc})$ [3]. We use inference rules in order to show how the machine state changes during program execution. Given a program

[3] For simplicity, we will treat **loc**, **temp**, and **pc** as variables, and will update them using the standard assignment notation (:=)

```
procedure addNewInstructions : M × D → M × D
1       input: (m, d)
2       output: (m′, d′)
3       let m′ = m and d′ = d in
4           For all (i, t, r) ∈ storeSpill_d
5               let i_ss : I such that i_ss ∉ m′ in
6                   use(i_ss) := {t}; def(i_ss) := ∅; in(i_ss) := ∅; out(i_ss) := ∅;
7                   succ(i_ss) := succ(i); pred(i_ss) := {i}; succ(i) := {i_ss};
8                   useReg_d′ := useReg_d′ ∪ {(i_ss, t, r)};
9           For all (i, t, r) ∈ loadSpill_d
10              let i_ld : I such that i_ld ∉ m′ in
11                  def(i_ld) := {t}; use(i_ld) := ∅; in(i_ld) := ∅; out(i_ld) := ∅ ;
12                  pred(i_ld) := pred(i); succ(i_ld) := {i}; pred(i) := {i_ld};
13                  defReg_d′ := defReg_d′ ∪ {(i_ld, t, r)};
14          For all i j, (t_1, r_1), (t_2, r_2) ∈ parallelCopy_d
15              let i_pc : I = edge(i, j) in
16                  def(i_pc) := def(i_pc) ∪ {t_1}; use(i_pc) := use(i_pc) ∪ {t_2};
17                  defReg_d′ := defReg_d′ ∪ {(i_pc, t_1, r_1)};
18                  useReg_d′ := useReg_d′ ∪ {(i_pc, t_2, r_2)};
```

Fig. 4. This algorithm adds instructions to m representing loads, stores, and parallel copies. It also updates d to reflect the new instructions. The function **edge** assigns to each edge of m a new instruction i', such that $i' \notin m$.

$f \in \mathcal{F}$, we generate the following inference rules for each instruction $i \in f$:

$$\frac{\forall(t,r) \in \mathbf{use}(i), (\mathbf{loc}(t) = r \ \wedge \ \mathbf{temp}(r) = t \ \wedge \ \mathbf{pc} = i \ \wedge \ \neg\mathbf{isCall}(i))}{\forall(t_0,r_0) \in \mathbf{def}(i), (\mathbf{loc}(t_0) := r_0; \mathbf{temp}(r_0) := t_0; \mathbf{pc} := \mathbf{any}(\mathbf{succ}(i)))} \quad (1)$$

$$\frac{\forall(t,r) \in \mathbf{use}(i), (\mathbf{loc}(t) = r \ \wedge \ \mathbf{temp}(r) = t \ \wedge \ \mathbf{pc} = i \ \wedge \ \mathbf{isCall}(i)}{\forall r, (\mathbf{isCallerSave}(r) \Rightarrow \mathbf{temp}(r) := \bot); \mathbf{pc} := \mathbf{any}(\mathbf{succ}(i)))} \quad (2)$$

$$\frac{\mathbf{pc} = i \ \wedge \ \exists(t,r) \in \mathbf{use}(i), \mathbf{loc}(t) \neq r}{\mathbf{pc} := \mathbf{error}} \quad (3)$$

$$\frac{\mathbf{pc} = i \ \wedge \ \exists(t,r) \in \mathbf{use}(i), \mathbf{temp}(r) \neq t}{\mathbf{pc} := \mathbf{error}} \quad (4)$$

Given a program f, we let i_0 be the entry point of its control flow graph. We have $\mathbf{def}(i_0) = \mathbf{input}(f)$, and $\mathbf{use}(i_0) = \bot$. The following inference rule defines the initial state of a computation:

$$\frac{true}{\mathbf{pc} := i_0; \forall r, \mathbf{temp}(r) := \bot; \forall t, \mathbf{loc}(t) := \bot} \quad (5)$$

We are now ready to formalize our definition of semantic consistency. In definition 1 below, let m be a RTL program, let $\mathcal{A}$ be the implementation of a register allocator, let $d = \mathcal{A}(m, K)$ be a set of allocation directives, and let f = **replaceRegisters (addNewInstructions** (m, d)**)**. Intuitively, f is semantically consistent if, every time one of its instruction is executed, all the used temporaries can be found in the locations stipulated by d.

Definition 1. [Semantic Consistency of Register Allocation Directives] *Given a solution f for the register allocation problem, let Σ be the set of states reachable through the inference rules (1) to (5) when applied on f. The mapping f is semantically consistent if Σ does not contain a state s such that $s \models \mathbf{pc} = \mathbf{error}$.*

```
procedure replaceRegisters : M × D → F × D
1     input: (m', d')
2     output: (f, d')
3     let f = m' in
4         For all i ∈ f
5             let use' = ∅ in
6                 For all t ∈ use(i)
7                     use(i) := use(i) \ t;
8                     use' := use' ∪ {(t, dir(d', i, t, r))};
9                 use(i) := use';
10            let def' = ∅ in
11                For all t ∈ def(i)
12                    def(i) := def(i) \ t;
13                    def' := def' ∪ {(t, dir(d', i, t, r))};
14                def(i) := def';
```

Fig. 5. This algorithm replace each temporary or pre-colored register t in $\mathcal{S}$ by a pair (t, r), where r is the register assigned to t by $\mathcal{D}'$.

3.4 Types of Register Allocation Errors

We define the following set of logic "macros" in order to make or presentation more readable:

$$\begin{array}{lcl} \mathbf{bindDef}(t,r) & \equiv & \exists i, (\mathbf{pc} = i) \ \wedge \ (t,r) \in \mathbf{def}(i) \\ \mathbf{bindUse}(t,r) & \equiv & \exists i, (\mathbf{pc} = i) \ \wedge \ (t,r) \in \mathbf{use}(i) \\ \mathbf{acrossCall}(t) & \equiv & \exists i, (\mathbf{pc} = i) \ \wedge \ \mathbf{isCall}(i) \ \wedge \ \mathbf{isCallerSave}(\mathbf{loc}(t)) \end{array}$$

If the set of reachable states Σ contains an inconsistent state s, such that, $s \models \mathbf{pc} = \mathbf{error}$, then s can only be reached via inference rules (3) and (4). Below we describe the situations that can cause these rules to be satisfied:

1. The inference rule (3) describes the situation in which an instruction i is about to be executed, there exists a temporary t, such that $t \in \mathbf{use}(i)$, but the actual location of this temporary is different from the expected location, i.e.: $\mathbf{loc}(t) \neq \mathbf{dir}(d', i, t, in)$. Such errors may happen in only two situations:
   (a) An error in the set of register allocation directives cause a temporary t to be defined into a register r_1 that is different from the register r_2 in which t is expected to be found [4]:

$$s \models \mathbf{bindDef}(t, r_1) \ \wedge \ \neg \mathbf{bindDef}(t, r_2) \, \mathcal{U} \, \mathbf{bindUse}(t, r_2) \qquad (6)$$

   (b) There is a path ρ in $\mathcal{S}$ from the initial instruction i_0 to an instruction i where t is used, and t is not defined along ρ. Such situation can be an error in the register allocator, or it can be a deficiency of the high level programing language been compiled. *Non-strict* languages, such as

[4] $s \models \phi_1 \mathcal{U} \phi_2$ if $s \models \phi_2$ or for some successor s' of s, $s' \models \phi_1 \vee \phi_1 \mathcal{U} \phi_2$.

C, allow the use of undefined variables. This is not possible in *strict* languages, such as Java.

$$s_0 \models \neg \mathbf{bindDef}(t, r)\ \mathcal{U}\ \mathbf{bindUse}(t, r) \tag{7}$$

2. The inference rule (4) describes the situation in which a machine register that holds the value of a live temporary has been overwritten, that is, $\mathbf{temp}(\mathbf{dir}(d', i, t, in)) \neq t$. We assume that only variable definitions and function calls can cause the overwriting of registers. The following events can lead to this situation:
   (a) There is a path in $\mathcal{S}$ in which the definition of a temporary t_1 is overwritten by the definition of a temporary t_2 while t_1 is still alive.

$$s \models \mathbf{bindDef}(t_2, r)\ \wedge\ \neg\mathbf{bindDef}(t_1, r)\ \mathcal{U}\ \mathbf{bindUse}(t_1, r) \tag{8}$$

   (b) A temporary t, stored in a caller save register, has its location overwritten because it is alive across a function call.

$$s \models \mathbf{acrossCall}(t)\ \wedge\ \neg\ \mathbf{bindDef}(t, \mathbf{loc}(t))\ \mathcal{U}\ \mathbf{bindUse}(t, \mathbf{loc}(t)) \tag{9}$$

4 Model Checking Register Allocation

Given a LTL program f, and Σ, the set of states that f produces, our model checking problem consists in verifying if s_0, the initial state specified by rule (5), does not lead to an inconsistent state, that is: $s_0 \models \neg(true\,\mathcal{U}\ (\mathbf{pc} = \mathbf{error}))$. The inference rules described in Section 3 produce $O(|I| \times |T| \times |R|)$ states. However, it is possible to use liveness analysis information to avoid the generation of the graph of reachable states. For the sake of completeness, we outline the well know liveness analysis algorithm in Figure 6. The control flow graph presented in Figure 3 has its edges augmented with liveness information.

Once liveness analysis has been performed, the algorithm **findBug**, shown in Figure 7, scans the control flow graph of f searching for inconsistency cases. There are four different types of inconsistency cases, one for each error described in Section 3.4. Figure 8 illustrates each case: (8 a) an instruction may expect temporary t at location r_2, although its actual location is r_1 – **findBug** returns **badAlloc**; (8 b) an unexpected pair alive at the **in** set of instruction i_0 denotes the use of an undefined variable – **findBug** may return **undefAlloc** or **badAlloc**; (8 c) The location of the live temporary t_1 is overwritten by t_2 – **findBugs** returns **instOvt**; (8 d) t has been stored into a caller save register, and its location is overwritten by a function call – **findBug** returns **callOvt**.

Theorem 1 states the correctness of the proposed algorithms. It is important to notice that we are not validating the mapping between spilled values and memory slots. For instance, it is possible that an ordinary memory access overwrite the cell been used to store an evicted temporary. We have opted for not dealing with this issue firstly because many register allocation algorithms abstract this information, secondly, because addressing modes are highly architecture dependent. If the memory address of spilled temporaries is available, it is

```
procedure livenessAnalysis : F → F
1     input: f
2     output: f
3     For all i ∈ f
4         in(i) := ∅; out(i) := ∅;
5     repeat
6         for all i ∈ f
7             in'(i) := in(i); out'(i) := in(i);
8             in(i) := use(i) ∪ (out(i) \ def(i));
9             out(i) := ⋃_{s∈succ(i)} in(s);
10    until in'(i) = in (i) ∧ out'(i) = out(i) for all i
```

Fig. 6. Liveness analysis algorithm as given by Appel and Palsberg [4, p.206].

```
procedure findBug : F → msg
1     input: f
2     output: { badAlloc, undefAlloc, instOvt, callOvt, correct }
3     for all i ∈ f
4         if ∃((t, r2) ∈ in(i)) ∧ ∃((t, r2) ∈ out(i)) ∧ ∃((t, r1) ∈ def(i)) ∧ r1 ≠ r2
5             return badAlloc;
6         if ∃((t2, r) ∈ in(i)) ∧ ∃((t2, r) ∈ out(i))
7             if ∃((t1, r) ∈ def(i)) ∧ t1 ≠ t2
8                 return instOvt;
9             else if isCall(i) ∧ isCallerSave(r)
10                return callOvt;
11    if ∃((t, r) ∈ in(i0)) ∧ t ∉ input(F)
12        return undefAlloc;
13    else
14        return correct;
```

Fig. 7. This algorithm reports the existence of a bug in the set of allocation directives. **msg** is any of the five possible result messages.

possible to circumvent such limitation by regarding memory cells as variables in our data flow analysis. In the proof of Theorem 1 we assume that the mapping between spilled values and memory is correct.

Theorem 1. *Let* f = **replaceRegisters (addNewInstructions** (m, d)**)**. **findBug (livenessAnalysis** (f)**)** *returns* **correct***, if, and only if,* m *and* d *are semantically consistent.*

Proof. $\Rightarrow$) If **findBug** $\neq$ **correct**, then four cases, one for each possible outcome of **findBugs** must be considered. All the cases have similar analysis; we present two of them:

1) **findBug** = **badAlloc** $\Rightarrow$ f contains the configuration shown in Figure 8 (a) $\Rightarrow$ There exists path $i_1, i_2, \ldots, i_n$ such that $(t, r_1) \in def(i_1)$, $(t, r_2) \in use(i_n)$, and $\forall k, 1 < k < n, (t, r_2) \notin def(i_k)$ $\Rightarrow$ There exists a sequence S of states

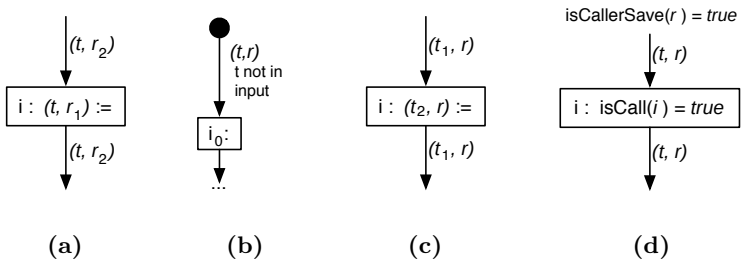


Fig. 8. Source of semantic inconsistencies: (a) Temporary found in unexpected location. (b) Undefined variable. (c) Live temporary overwritten by another temporary. (d) Caller save register overwritten in function call.

$s_1, s_2, \ldots, s_n$ such that $\forall k, 1 \leq k \leq n, \mathbf{pc} = i_k \Rightarrow s_1$ satisfies Equation (6) given in Section 3.4.

2) **findBug** = **callOvt** $\Rightarrow f$ contains the configuration show in Figure 8 (d) $\Rightarrow$ There exists path $i_1, i_2, \ldots, i_n$ such that, for some pair (t, r), i_1 satisfies conditions in lines (6) and (9) of **findBug** $\Rightarrow (t, r) \in use(i_n)$, and $\forall k, 1 < k < n, (t, r) \notin def(i_k) \Rightarrow$ There exists a sequence S of states $s_1, s_2, \ldots, s_n$ such that $\forall k, 1 \leq k \leq n, \mathbf{pc} = i_k. \Rightarrow s_1$ satisfies Equation (9) of Section 3.4.

The cases where **findBug** = **undefAlloc**, or **findBug** = **instOvt** are similar. In the first case, there will be a state s that satisfies equation (7), and in the latter, Equation (8) will be satisfied.

$\Leftarrow$) If (m, d) is not semantically consistent, then one of the four equations of Section 3.4 must be true. We consider the case where Equation 6 or Equation 8 is satisfied. The other cases are similar:

1)Equation 7 is satisfied. $\Rightarrow$ The initial state s_0 is such that $s_0 \models \neg\mathbf{bindDef}(t, r)\ \mathcal{U}\ bindUse(t, r)$. $\Rightarrow$ $s_0 \models (\neg\exists i_1, \mathbf{pc} = i_1 \wedge (t, r) \in \mathbf{def}(i_1))\ \mathcal{U}\ (\exists i_2, \mathbf{pc} = i_2 \wedge (t, r) \in \mathbf{use}(i_2)$. $\Rightarrow$ Let i_0 be the first instruction of the control flow graph. There is a path from i_0 to i_2, such that (t, r) is used at i_2, but never defined. $\Rightarrow$ (t, r) is alive at the entry point of i_0. $\Rightarrow$ Conditions in line (11) of **findBug** are true. $\Rightarrow$ **findBug** returns **undefAlloc**.

2) Equation 8 is satisfied $\Rightarrow$ There is a state s such that $s \models \mathbf{bindDef}(t_2, r) \wedge \neg\mathbf{bindDef}(t_1, r)\ \mathcal{U}\ bindUse(t_1, r) \Rightarrow s \models (\exists i_1, \mathbf{pc} = i_1 \wedge (t_2, r) \in \mathbf{def}(i_1)) \wedge \neg(\exists i_2, \mathbf{pc} = i_2 \wedge (t_1, r) \in \mathbf{def}(i_2))\ \mathcal{U}\ (\exists i_3, \mathbf{pc} = i_3 \wedge (t_1, r) \in \mathbf{use}(i_3) \Rightarrow$ The control flow graph contains a path from instruction i_1 to instruction i_3, such that (t_1, r) is used at i_3, but not defined until i_1, where (t_2, r) is defined. $\Rightarrow$ Conditions in lines (6) and (7) of **findBug** are true. $\Rightarrow$ **findBug** returns **instOvt**.

Using similar reasoning, one can show that if Equation 6 is satisfied, **findBug** returns **badAlloc**, and if Equation 9 is true, **findBug** returns **callOvt**. □

A number of properties can be proved about a program f that contains a semantically consistent mapping from temporaries to registers. Examples include:

1. Two live ranges of different temporaries reach a joint point assigned to different registers.

2. Two live ranges of the same temporary reach a joint point assigned to the same register.
3. All the temporaries in the **def** set of an instruction are assigned to machine registers immediately after the instruction is executed.
4. The interference graph derived from f is K colorable, where K is the number of available machine registers.

We give a proof of the last property as a corollary of Theorem 1. Similar reasoning can be used to proof the other properties.

Corollary 1. *If the allocation directives are correct, the interference graph derived from f is K-colorable, where K is the number of machine registers available in the target architecture.*

Proof. Assume that the interference graph is not K colorable $\Rightarrow$ Two interfering pairs, e.g. (t_1, r) and (t_2, r), are simultaneously alive at some point of the control flow graph $f \Rightarrow$ If both temporaries are defined in f, then, or the live range of (t_1, r) crosses the definition point of (t_2, r), or vice-versa. In this case, **findBug** reports **instOvt**. If at least one of the temporaries is undefined, then its live range reaches i_0, and **findBugs** reports **undefAlloc**. In both cases the set of allocation directives is not semantically consistent. □

The four equations defined in Section 3.4 yield a graph reachability algorithm that can be used to list the instructions that cause semantic inconsistencies. Figure 9 outlines such algorithm.

4.1 Complexity Analysis

The table below shows the complexity of the algorithms discussed in this paper. All the complexity results refer to a program m containing $|I|$ instructions and $|T|$ temporaries. K is the number of registers available in the target architecture. It is interesting to notice that the algorithms tend to run faster if m is semantically consistent. This difference is due to the fact that, in a semantically consistent program, there are at most K interferences at each program point, whereas the number of interferences in a inconsistent program is upper bounded by $|T|$ (see Corolary 1). Therefore, at most K set operations will be performed in the **in**'s and **out**'s bitsets of a correct program.

Algorithm	Worse case	correct program
addNewInstructions	$O(\|I\| \times K)$	$O(\|I\| \times K)$
replaceRegisters	$O(\|I\| \times K)$	$O(\|I\| \times K)$
livenessAnalysis	$O(\|I\|^2 \times \|T\|^2)$	$O(\|I\|^2 \times K^2)$
findBug	$O(\|I\| \times \|T\|)$	$O(\|I\| \times K)$
listBugs	$O(\|I\|^2 \times \|T\| \times K)$	$O(\|I\|^2 \times K^2)$

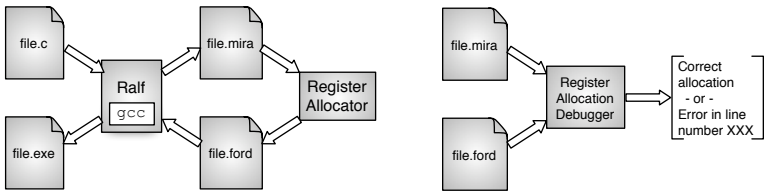
```
procedure visit : F × I × T × R → set of (I× msg)
1       input: f, i, (t, r)
2       output: (Instruction, { undefAlloc | badAlloc | instOvt | callOvt } )
3       external: array of Instruction: visited
4       if (not visited(i))
5           visited(i) := true;
6           if  ∃((t, r2) ∈ in(i))  ∧  ∃((t, r2) ∈ out(i))  ∧  ∃((t, r1) ∈ def(i))  ∧  r1 ≠ r2
7               return (i, badAlloc);
8           else if  ∃((t2, r) ∈ in(i))  ∧  ∃((t2, r) ∈ out(i))
9               if ∃((t1, r) ∈ def(i))  ∧  t1 ≠ t2
10                  return (i, instOvt);
11              else if isCall(i)  ∧  isCallerSave(r)
12                  return (i, callOvt);
13          else if  ∃((t, r) ∈ in(i0))  ∧  t ∉ input(f)
14              return (i, undefAlloc);
15          for all ip ∈ pred(i)
16              if((t, r) ∈ out(ip))
17                  print(visit(f, ip, (t, r)));

procedure listBugs : F → ⊥
1       input: f
2       output: ⊥
3       internal: array of Instruction: visited
4       livenessAnalysis(f);
5       for all i ∈ f
6           for all (t, r) ∈ use(i)
7               for all i ∈ f
8                   visited(i) := false;
9               print(visit(f, i, (t, r)));
```

Fig. 9. This algorithm recursively lists the sources of semantic inconsistencies in f.

4.2 Implementation

We have implemented the proposed algorithms as an extension of the Ralf framework. The main parts of this framework are outlined in Figure 10 (a). Ralf uses a format known as MIRA (Mathematical Intermediate representation for Register Allocation) to represent the RTL language $\mathcal{M}$. In order to represent allocation directives ($\mathcal{D}$), Ralf uses a format called FORD (FOrmat for Register Allocation Directives). Any implementation of a register allocator that is able to read MIRA and output FORD can be plugged into Ralf. Our translation validator compares the FORD and MIRA files in order to certify their semantic consistency, or to give a counter-example if they are not consistent, as shown in Figure 10 (b).

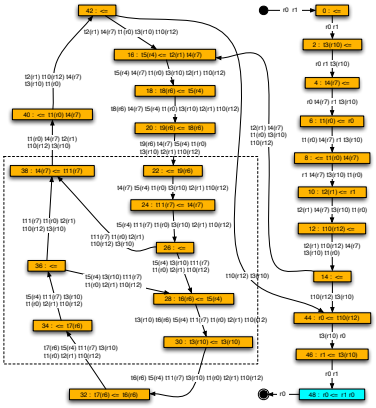
In order to facilitate the identification of errors, our debugging environment provides the application developer with a visual representation of the control flow graph. This software generates files in the ".dot" format [7], which is a popular representation of graphs. It can produce plain graphs for programs written in MIRA, or it can show register assignments, given the MIRA description and the set of register allocation directives. Figure 11 (a) shows a simple C program, and Figure 11 (b) gives its $\mathcal{F}$ representation, with liveness analysis results in the edges. The algorithms discussed in the previous sections have been implemented in Java, and are completely independent on the implementation of Ralf: they can be used with any compiler that read MIRA specifications, and output FORD directives. For more information about both formats, see [12].



(a) **(b)**

Fig. 10. (a) The main components of the Ralf framework. (b) The proposed translation validation tool.

```
/*
 * This program counts the number
 * of characters passed to it as
 * command line arguments.
 */
int main(int argc, char **argv) {
  int sum = 0;
  int c = 0;
  for(c = 1; c < argc; c++) {
    char *aux = argv[c];
    while(*aux != '\0') {
      aux++;
      sum++;
    }
  }
  printf("Sum = %d\n", sum);
}
```



(a) **(b)**

Fig. 11. (a) Target program written in C. (b) Its control flow graph $\mathcal{F}$, generated by our debugging tool.

5 Conclusion

This paper has introduced the concept of semantic consistency of register allocation directives, and has presented a collection of data flow algorithms that statically validate the output of register allocators. The interest for such algorithms arouse from the real necessity of debugging different implementations of register allocators, and greatly improved our ability to detect errors in their outputs. We hope that our validation framework can help other researchers who are developing new register allocation techniques as much as it has help us. The algorithms discussed in this paper can be employed to validate the output of most of the well-known register allocation paradigms: graph-coloring based, control flow based, ILP based, etc. Moreover, they don't depend on any particular instruction set. Our implementation and further reading material can be found at: `http://compilers.cs.ucla.edu/fernando/projects/debugger/`

References

1. Johan Agat. Types for register allocation. *Lecture Notes in Computer Science*, 1467:92–111, 1997.
2. Christian Andersson. Register allocation by optimal graph coloring. In *12th Conference on Compiler Construction*, pages 34–45. Springer, 2003.
3. Andrew W Appel and Lal George. Optimal spilling for cisc machines with few registers. In *International Conference on Programming Languages Design and Implementation*, pages 243–253. ACM Press, 2001.
4. Andrew W Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.
5. Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.
6. R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11):638–642, 1974.
7. Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SoftwarePractice and Experience*, 30(11):1203 – 1233, 2000.
8. Lal George and Andrew W Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.
9. Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *15th International Conference on Compiler Construction*. Springer-Verlag, 2006.
10. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Annual Symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
11. Mayur Naik and Jens Palsberg. Compiling with code size constraints. *Transactions on Embedded Computing Systems*, 3(1):163–181, 2004.
12. Venkata Krishna Nandivada. *Combining Stack Location Allocation with Register Allocation*. PhD thesis, University of California, Los Angeles, 2005.
13. Venkata Krishna Nandivada and Jens Palsberg. Sara: Combining stack allocation and register allocation. In *International Conference on Compiler Construction*. Springer-Verlag, 2006.
14. George C. Necula. Translation validation for an optimizing compiler. In *Conference on Programming Language Design and Implementation*, pages 83–95. ACM, 2000.
15. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.
16. Atsuchi Ohori. Register allocation by proof transformation. *Science of Computer Programming*, 50(1-3):161–187, 2004.
17. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *The Third Asian Symposium on Programming Languages and Systems*, pages 315–329. Springer, 2005.
18. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation after classic SSA elimination is np-complete. In *Foundations of Software Science and Computation Structures*. Springer, 2006.