Battling windmills with Coq: formal verification of a compilation algorithm for parallel moves

Laurence Rideau (laurence.rideau@inria.fr)
INRIA Sophia-Antipolis Méditerranée

Bernard Paul Serpette (bernard.serpette@inria.fr)
INRIA Sophia-Antipolis Méditerranée

Xavier Leroy (xavier.leroy@inria.fr)
INRIA Paris-Rocquencourt

Abstract. This article describes the formal verification of a compilation algorithm that transforms parallel moves (parallel assignments between variables) into a semantically-equivalent sequence of elementary moves. Two different specifications of the algorithm are given: an inductive specification and a functional one, each with its correctness proofs. A functional program can then be extracted and integrated in the Compcert verified compiler.

1. Introduction

Parallel assignment is a programming construct found in some programming languages (Algol 68, Common Lisp) and in some compiler intermediate languages (the RTL intermediate language of the GNU Compiler Collection). A parallel assignment is written $(x_1, \ldots, x_n) := (e_1, \ldots, e_n)$, where the x_i are variables and the e_i are expressions possibly involving the variables x_i. The semantics of this parallel assignment is to evaluate the expressions $e_1, \ldots, e_n$, then assign their respective values to the variables $x_1, \ldots, x_n$. Since the left-hand side variables can occur in the right-hand side expressions, the effect of a parallel assignment is, in general, different from that of the sequence of elementary assignments $x_1 := e_1; \ldots; x_n := e_n$. Welch (1983) gives examples of uses of parallel assignments.

Compiling a parallel assignment instruction amounts to finding a *serialisation* – a sequence of single assignments $x'_1 = e'_1; \ldots; x'_m = e'_m$ – whose effect on the variables x_i and on other program variables is the same as that of the source-level parallel assignment. Since parallel assignment includes permutation of variables $(x_1, x_2) = (x_2, x_1)$ as a special case, the generated sequence of elementary assignments cannot, in general, operate in-place over the x_i: additional storage is necessary under the form of temporary variables. A trivial compilation algorithm, outlined by Welch (1983), uses n temporaries $t_1, \ldots, t_n$, not present in the original program:

$$t_1 := e_1;\ \ldots;\ t_n := e_n;\ x_1 := t_1;\ \ldots;\ x_n := t_n$$

As shown by the example $(x_1, x_2) = (x_2, x_1)$, it is possible to find more efficient sequences that use fewer than n temporaries.

Finding such sequences is difficult in general: Sethi (1973) shows that compiling a parallel assignment over n variables using at most K temporaries, where K is a constant independent of n, is an NP-hard problem. However, there exists a family of parallel assignments for which serialisation is computationally easy: parallel assignments of the form $(x_1, \ldots, x_n) := (y_1, \ldots, y_n)$, where the right-hand sides y_i are restricted to be variables (either the x_i variables or other variables). We call these assignments *parallel moves*. They occur naturally in compilers when enforcing calling conventions (Appel, 1992). May (1989) shows another example of use of parallel moves in the context of machine code translation. It is a folklore result that parallel moves can be serialised using at most one temporary register, and in linear time (May, 1989).

The purpose of this paper is to formalize a compilation algorithm for parallel moves and mechanically verify its semantic correctness, using the Coq proof assistant (Coq development team, 2007; Bertot and Castéran, 2004). This work is part of a larger effort, the Compcert project (Leroy, 2006; Blazy et al., 2006; Bertot et al., 2006), which aims at mechanically verifying the correctness of an optimizing compiler for a subset of the C programming language. Every pass of the Compcert compiler is specified in Coq, then proved to be semantics-preserving: the observable behavior of the generated code is identical to that of the source code.

The parallel move compilation algorithm is used in the pass that enforces calling conventions for functions. Consider a source-level function call $f(e_1, \ldots, e_n)$. Earlier compiler passes produces intermediate code that computes the values of the arguments $e_1, \ldots, e_n$ and deposits them in locations $l_1, \ldots, l_n$. (Locations are either processor registers or stack slots.) The function calling conventions dictate that the separately-compiled function f expects its parameters in conventional locations $l'_1, \ldots, l'_n$ determined by the number and types of the parameters. The register allocation phase is instructed to prefer the locations $l'_1, \ldots, l'_n$ for the targets of $e_1, \ldots, e_n$, but such preferences cannot always be honored. Therefore, the pass that enforces calling conventions must, before the call, insert elementary move instructions that perform the parallel move $(l'_1, \ldots, l'_n) := (l_1, \ldots, l_n)$. Only two hardware registers (one integer register, one floating-point register) are available at this point to serve as temporaries. Therefore, the naive compilation algorithm for parallel moves will not do, and we had to implement and prove correct the space-efficient algorithm.

The formal specification and correctness proof of this compilation algorithm is challenging. Indeed, this was one of the most difficult parts of the whole Compcert development. While short and conceptually clear, the algorithm we started with (described section 3) is very imperative in nature. The underlying graph-like data structure, called *windmills* in this paper, is unusual. Finally, the algorithm involves non-structural recursion. We show how to tackle these difficulties by progressive refinement from a high-level, nondeterministic, relational specification of semantics-preserving transformations over windmills.

The remainder of this paper is organized as follows. Section 2 defines the windmill structure. Section 3 shows the folklore, imperative serialisation algorithm that we used as a starting point. Two inductive specifications of the algorithm follow, a non-deterministic one (section 4), and a deterministic one (section 5), with the proofs that they are correct and consistent. Section 6 derives a functional implementation from these specifications and proves consistency with the inductive specifications, termination, and correctness. Section 7 proves additional syntactic properties of the result of the compilation function. In section 8, the correctness result of section 6 is extended to the case where variables can partially overlap. Concluding remarks are presented in section 9.

The complete, commented source for the Coq development presented here is available at `http://gallium.inria.fr/~xleroy/parallel-move/`.

2. Definitions and notations

An elementary move is written $(s \mapsto d)$, where the source register s and the destination register d range over a given set $\mathcal{R}$ of registers. Parallel moves as well as sequences of elementary moves are written as lists of moves $(s_1 \mapsto d_1) \bullet \cdots \bullet (s_n \mapsto d_n)$. The $\bullet$ operator denotes the concatenation of two lists. We overload it to also denote prepending a move in front of a list, $(s \mapsto d) \bullet l$, and appending a move at the end of a list, $l \bullet (s \mapsto d)$. The empty list is written $\emptyset$.

We assume given a set $\mathcal{T} \subseteq \mathcal{R}$ of temporary registers: registers that are not mentioned in the initial parallel move problem and that the compiler can use to break cycles. We also assume given a function $T : \mathcal{R} \to \mathcal{T}$ that associates to any register s a temporary register $T(s)$ appropriate for saving the value of s when breaking a cycle involving s. In the simplest case, only one temporary `tmp` is available and T is the

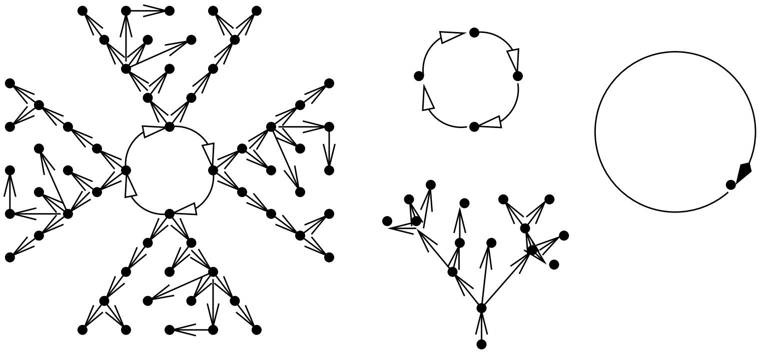

Figure 1. Examples of windmills

constant function $T(s) = \texttt{tmp}$. In more realistic cases, architectural or typing constraints may demand the use of several temporaries, for instance one integer temporary to move integer or pointer values, and one floating-point temporary to move floating-point values. In this example, $T(s)$ selects the appropriate temporary as a function of the type of s.

A parallel move $(s_1 \mapsto d_1) \bullet \cdots \bullet (s_n \mapsto d_n)$ is well defined only if the destination registers are pairwise distinct: $d_i \neq d_j$ if $i \neq j$. (If a register appeared twice as a destination, its final value after the parallel move would not be uniquely defined.) We can view a parallel move as a transfer relation. Each move $(s_i \mapsto d_i)$ corresponds to an edge in the graph of this relation. A parallel move is well defined if every register has at most one predecessor for the transfer relation.

Although this property is similar to the specification of forests (set of disjoint trees), it allows cycles such as $(r_1 \mapsto r_2) \bullet (r_2 \mapsto r_1)$. This is unlike a tree, where, by definition, there exists a unique element – the root – that has no predecessor.

Relations verifying the "at most one predecessor" property correspond to sets of disjoint *windmills*. A *windmill*, in our terminology, is a cycle – the *axle* – whose elements are the roots of trees – the *blades* –. Figure 1 figure shows a set of 4 windmills: the general case, the special case of a tree, a simple cycle with four registers, and the special case of a self-loop.

3. An imperative algorithm

There are two special cases of parallel move problems where serialization is straightforward. In the case of a simple cycle, i.e. an axle without any blade, the transfer relation is:

$$(r_n \mapsto r_1) \bullet \ldots \bullet (r_2 \mapsto r_3) \bullet (r_1 \mapsto r_2)$$

Serialization is done using a single temporary register $t = T(r_1)$:

$$t := r_1;\ r_1 := r_n;\ \ \ldots;\ r_3 := r_2;\ r_2 := t.$$

The other easy case corresponds to a transfer relation that is a tree, such as for instance

$$(r_1 \mapsto r_2) \bullet (r_1 \mapsto r_3) \bullet (r_2 \mapsto r_4)$$

In this case, serialization corresponds to enumerating the edges in a bottom-up topological order:

$$r_4 := r_2;\ r_2 := r_1;\ r_3 := r_1.$$

C. May (1989) describes an algorithm that generalizes these two special cases. This algorithm follows the topology of the transfer relation. First, remove one by one all the edges that have no successors, emitting the corresponding sequential assignments in the same order. (In windmill terminology, this causes the blades to disappear little by little.) Eventually, all that remains are simple, disjoint cycles (windmill axles) which can be serialized using one temporary as described above.

We now consider a variant of this algorithm that processes blades and axles simultaneously, in a single pass. The algorithm is given in figure 2 in Caml syntax. It can be read as pseudo-code knowing that `a.(i)` refers to the `i`-th element of array `a`.

It takes as arguments two arrays `src` and `dst` containing respectively the source registers $s_1, \ldots, s_n$ and the target registers $d_1, \ldots, d_n$, as well as the temporary-generating function `tmp`. The elementary moves produced by serialization are successively printed using the C-like `printf` function.

The algorithm implements a kind of depth-first graph traversal using the `move_one` function that takes an edge $(src_i \mapsto dst_i)$ of the transfer relation as an argument. In line 9 and 10, all successors of dst_i are handled. Line 17 handles the case of an already analysed edge. In the case of line 12, it is a simple recursive call. In the case of line 14, a cycle is discovered and a temporary register is used to break it.

When exiting this loop on line 19, all the nodes that can be reached by dst_i have been analysed, the edge $(src_i \mapsto dst_i)$ is serialized and marked as analyzed.

When a cycle is discovered, the stack of active calls to `move_one` correspond to the edges $((r_n \mapsto r_1), \ldots, (r_2 \mapsto r_3), (r_1 \mapsto r_2))$ and, at

```
1  type status = To_move | Being_moved | Moved
2
3  let parallel_move src dst tmp =
4      let n = Array.length src in
5      let status = Array.make n To_move in
6      let rec move_one i =
7          if src.(i) ≠ dst.(i) then begin
8              status.(i) ← Being_moved;
9              for j = 0 to n - 1 do
10                 if src.(j) = dst.(i) then
11                     match status.(j) with
12                     | To_move →
13                         move_one j
14                     | Being_moved →
15                         printf "%s␣:=␣%s;\n" (tmp src.(j)) src.(j);
16                         src.(j) ← tmp src.(j)
17                     | Moved →
18                         ()
19             done;
20             printf "%s␣:=␣%s;\n" dst.(i) src.(i);
21             status.(i) ← Moved
22         end in
23     for i = 0 to n - 1 do
24         if status.(i) = To_move then move_one i
25     done
```

Figure 2. A one-pass, imperative algorithm to compile parallel moves

line 14, the edge $(r_1 \mapsto r_2)$ is analysed again. The side effect of line 16 thus necessarily acts on the edge that started the recursion of the `move_one` function, i.e. the edge at the bottom of the stack. The main loop, lines 23 to 25, ensures that all edges are analyzed at least once.

4. Nondeterministic specification

The algorithm in figure 2 does not lend itself easily to program proof: first, it is written imperatively, making it quite removed from a mathematical specification; second, it commits to a particular strategy for solving the problem, obscuring the essential invariants for the correctness proof. In this section, we develop an abstract specification of the steps of the algorithm, and show that these steps preserve semantics. This specification is not deterministic: it does not constraint which step must be taken in a given state.

In the algorithm of figure 2, notice that every edge of the transfer relation begins in the state `To_move` (line 5), then takes the state `Being_moved` (line 8), and finally reaches the state `Moved` (line 21). Rather than associating a status to each edge, we will represent the current state of the compilation as three disjoint lists of edges, each list containing edges having the same status. The state is therefore a triple (μ, σ, τ) of lists of edges: μ is the *to-move* list, σ the *being-moved* list and τ the *moved* list.

Each step of the algorithm will either extract an element from the *to-move* list μ and add it to the *being-moved* list σ, or remove an element of the latter and add it to the *moved* list τ.

The *being-moved* list σ is used as a stack, simulating the recursive calls to the `move_one` function in the imperative algorithm. In addition to pushing and popping moves from the beginning of σ, the last element of σ can also be modified when a cycle is discovered.

The last component of the state, the *moved* list τ, is used to accumulate the sequence of elementary moves that the imperative algorithm emits using the `printf` function. Moves are successively added to the front of the list τ. Therefore, the elementary moves listed in τ are to be executed from right to left.

4.1. Inference rules

The inference rules below define a rewriting relation $\triangleright$ between states (μ, σ, τ). Each rule describes a step that the compilation algorithm is allowed to take. A run of compilation is viewed as a sequence of rewrites from the initial state $(\mu, \emptyset, \emptyset)$ to a final state $(\emptyset, \emptyset, \tau)$, where μ is the parallel move problem we set out to compile, and $\texttt{reverse}(\tau)$ is the sequence of elementary moves generated by this compilation run.

$$\frac{}{(\mu_1 \bullet (r \mapsto r) \bullet \mu_2, \sigma, \tau) \triangleright (\mu_1 \bullet \mu_2, \sigma, \tau)} \quad [\text{Nop}]$$

The first rule deals with the case where the transfer relation has an edge $(r \mapsto r)$ whose source and destination are identical. This case is specially handled at line 7 of the imperative algorithm. Note that in this imperative algorithm, this edge is not annotated as `Moved` and keeps the status `To_move` until the end of the algorithm. Nevertheless this edge cannot result from a recursive call of the function `move_one` because, in this case, the caller would correspond to an edge $(s \mapsto r)$ and that would violate the fact that the transfer relation is a windmill: indeed, there would exist two different edges with the same destination. It is therefore valid to move the test of line 7 up to the level of the main loop at line 24.

The [Nop] rule, as written, is a particular case of the following [Start] rule. This shows that the test on line 7 of the imperative algorithm is not required for the validity of the algorithm and can be regarded as an optimization.

$$\frac{}{(\mu_1 \bullet (s \mapsto d) \bullet \mu_2, \emptyset, \tau) \triangleright (\mu_1 \bullet \mu_2, (s \mapsto d), \tau)} \text{ [Start]}$$

The [Start] rule corresponds to the first call to the `move_one` function at line 24. We know that in this case no edge has the status `Being_Moved` and this rule will thus be the only one to handle the case where the *being-moved* list is empty.

$$\frac{}{\begin{array}{l}(\mu_1 \bullet (d \mapsto r) \bullet \mu_2, (s \mapsto d) \bullet \sigma, \tau) \\ \quad \triangleright (\mu_1 \bullet \mu_2, (d \mapsto r) \bullet (s \mapsto d) \bullet \sigma, \tau)\end{array}} \text{ [Push]}$$

The [Push] rule corresponds to the recursive call at line 13. If the edge being analysed is $(s \mapsto d)$ (i.e. the top of the *being-moved* list) and if there exists a successor $(d \mapsto r)$ in the *to-move* list, the latter is transferred to to the top of the *being-moved* stack.

$$\frac{}{(\mu, \sigma \bullet (s \mapsto d), \tau) \triangleright (\mu, \sigma \bullet (T(s) \mapsto d), (s \mapsto T(s)) \bullet \tau)} \text{ [Loop]}$$

The [Loop] rule corresponds to the special case at lines 15 and 16 where a cycle is discovered. This rule is of more general use. Indeed, it is not essential to check the presence of a cycle before inserting a transfer using a temporary register. Thus, lines 15 and 16 can also be moved up to the level of the main loop (line 24). The fact that one uses the temporary register only in the case of a cycle must be regarded as an optimization. The crucial issue is to make sure that a transfer through a temporary register must be used whenever a cycle arises. This is ensured by the next rule.

$$\frac{\texttt{NoRead}(\mu, d_n) \ \wedge \ d_n \neq s_0}{\begin{array}{l}(\mu, (s_n \mapsto d_n) \bullet \sigma \bullet (s_0 \mapsto d_0), \tau) \\ \quad \triangleright (\mu, \sigma \bullet (s_0 \mapsto d_0), (s_n \mapsto d_n) \bullet \tau)\end{array}} \text{ [Pop]}$$

The [Pop] rule corresponds to returning from a recursive call to the function `move_one`. The first premise of this rule, $\texttt{NoRead}(\mu, d_n)$, is formally defined as

$$\texttt{NoRead}(\mu, d_n) \stackrel{\text{def}}{=} (\mu = \mu_1 \bullet (s \mapsto d) \bullet \mu_2 \ \Rightarrow \ s \neq d_n).$$

This premise checks that the edge $(s_n \mapsto d_n)$ under study no longer has its successor d_n in the *to-move* list. It therefore prevents the [Push] rule from being used if the value of d_n is still needed. The second premise, $d_n \neq s_0$, makes sure that the [Pop] rule cannot be used if d_n participates in a cycle, forcing the use of the [Loop] rule instead in the case of a cycle.

$$\frac{\mathtt{NoRead}(\mu, d_n)}{(\mu, (s \mapsto d), \tau) \triangleright (\mu, \emptyset, (s \mapsto d) \bullet \tau)} \quad \text{[Last]}$$

The [Last] rule corresponds to returning from the main loop at line 24. It is a special case of the [Pop] rule. As in the special case of the [Nop] rule, observe that a *to-move* list of the form $((r \mapsto r))$ can either be eliminated by the [Nop] rule, or rewritten to itself in two steps ([Start] then [Last]), or rewritten to $((T(r) \mapsto r) \bullet (r \mapsto T(r)))$ using a temporary register (rules [Start] then [Loop] then [Last]). These three results are equally valid.

4.2. Well-formedness invariant

In preparation for proving the semantic correctness of the rewriting rules, we first define a well-formedness property that acts as a crucial invariant in this proof.

DEFINITION 1 (Invariant). *A triple (μ, σ, τ) is well-formed, written $\vdash (\mu, \sigma, \tau)$, if and only if:*

1. *$\mu \bullet \sigma$ is a windmill:*
   $\mu \bullet \sigma = (l_1 \bullet (s_i \mapsto d_i) \bullet l_2 \bullet (s_j \mapsto d_j) \bullet l_3) \;\Rightarrow\; d_i \neq d_j$.
2. *The μ list does not contain temporary registers:*
   $\mu = \mu_1 \bullet (s \mapsto d) \bullet \mu_2 \;\Rightarrow\; (s \notin \mathcal{T} \;\wedge\; d \notin \mathcal{T})$.
3. *The σ list can only use a temporary register as the source of its last edge: $\sigma = \sigma_1 \bullet (s_0 \mapsto d_0) \;\Rightarrow\; d_0 \notin \mathcal{T}$ and $\sigma_1 = \sigma_2 \bullet (s \mapsto d) \bullet \sigma_3 \;\Rightarrow$ $(s \notin \mathcal{T} \;\wedge\; d \notin \mathcal{T})$.*
4. *The σ list is a path: $\sigma = ((r_{n-1} \mapsto r_n) \bullet \ldots \bullet (r_2 \mapsto r_3) \bullet (r_1 \mapsto r_2))$*

Notice that if μ is a windmill and does not use any temporary register, then the initial triple $(\mu, \emptyset, \emptyset)$ is well-formed.

LEMMA 1 (The invariant is preserved). *The rewriting rules transform a well-formed triple into a well-formed triple. If $S_1 \triangleright S_2$ and $\vdash S_1$, then $\vdash S_2$.*

Proof. By case analysis on the rule that concludes $S_1 \triangleright S_2$. For $S_1 = (\mu, \sigma, \tau)$, only the [Push] rule adds an element to σ, but the edge thus added extends the path already present in σ.

4.3. Dynamic semantics for assignments

To state semantic preservation for the rewriting relation, we need to give dynamic semantics to the parallel moves μ and the sequential moves σ and τ occurring in intermediate states. Let $\mathcal{V}$ be the set of run-time values for the language. The run-time state of a program is represented as an environment $\rho : \mathcal{R} \to \mathcal{V}$ mapping registers to their current values. Assigning value v to register r transforms the environment ρ into the environment $\rho[r \leftarrow v]$ defined by

$$\rho[r \leftarrow v] = \begin{cases} r \mapsto v \\ r' \mapsto \rho(r') & \text{if } r' \neq r. \end{cases}$$

(We assume that assigning to a register r preserves the values of all other registers r', or in other terms that r does not overlap with any other register. We will revisit this hypothesis in section 8.)

The run-time effect of a parallel move μ is to transform the current environment ρ in the environment $[\![\mu]\!]_{//}(\rho)$ defined as:

$$[\![(s_n \mapsto d_n) \bullet \cdots \bullet (s_0 \mapsto d_0)]\!]_{//}(\rho) = \rho[d_0 \leftarrow \rho(s_0)] \ldots [d_n \leftarrow \rho(s_n)]$$

In this definition parentheses are omitted, as we require that μ is a windmill. The following lemma, showing that the order of moves within μ does not matter, is intensively used in our proofs:

LEMMA 2 (Independence). *If $\mu = \mu_1 \bullet (s \mapsto d) \bullet \mu_2$ is a windmill, then*

$$[\![\mu]\!]_{//}(\rho) = [\![(s \mapsto d) \bullet \mu_1 \bullet \mu_2]\!]_{//}(\rho)$$

Proof. By induction on μ_1. The base case $\mu_1 = \emptyset$ is trivial. For the inductive step, we need to prove that the first two elements of μ_1 can be exchanged: $(\rho[d_0 \leftarrow \rho(s_0)])[d_1 \leftarrow \rho(s_1)] = (\rho[d_1 \leftarrow \rho(s_1)])[d_0 \leftarrow \rho(s_0)]$. The windmill structure of μ_1 implies that $d_0 \neq d_1$, and therefore this equality is verified.

The run-time effect of a sequence of elementary moves τ, performed from left to right, is defined by:

$$\begin{aligned} [\![\emptyset]\!]_{\rightarrow}(\rho) &= \rho \\ [\![(s \mapsto d) \bullet \tau]\!]_{\rightarrow}(\rho) &= [\![\tau]\!]_{\rightarrow}(\rho[d \leftarrow \rho(s)]) \end{aligned}$$

The sequences of elementary moves τ appearing in the states of the rewriting relation are actually to be interpreted from right to left, or equivalently to be reversed before execution. It is therefore useful to define directly the semantics of a sequence of elementary moves τ performed from right to left:

$$\begin{aligned} [\![\emptyset]\!]_{\leftarrow}(\rho) &= \rho \\ [\![(s \mapsto d) \bullet \tau]\!]_{\leftarrow}(\rho) &= \rho'[d \leftarrow \rho'(s)] \text{ where } \rho' = [\![\tau]\!]_{\leftarrow}(\rho) \end{aligned}$$

LEMMA 3 (Reversed sequential execution). *For all sequences τ of elementary moves, $[\![\tau]\!]_{\leftarrow}(\rho) = [\![\mathtt{reverse}(\tau)]\!]_{\rightarrow}(\rho)$.*

Proof. Easy induction over τ.

DEFINITION 2 (Equivalent environments). *We say that two environments ρ_1 and ρ_2 are equivalent, and we write $\rho_1 \equiv \rho_2$, if and only if all non-temporary registers have the same values in both environments:*

$$\rho_1 \equiv \rho_2 \stackrel{\mathrm{def}}{=} \forall r \notin \mathcal{T},\ \rho_1(r) = \rho_2(r).$$

DEFINITION 3 (Semantics of a triple). *The dynamic semantics of the triple (μ, σ, τ) corresponds to first executing τ sequentially from right to left, then executing $\mu \bullet \sigma$ in parallel:*

$$[\![(\mu, \sigma, \tau)]\!](\rho) = [\![\mu \bullet \sigma]\!]_{//}([\![\tau]\!]_{\leftarrow}(\rho)).$$

Notice that for initial states, $[\![(\mu, \emptyset, \emptyset)]\!](\rho) = [\![\mu]\!]_{//}(\rho)$, while for final states, $[\![(\emptyset, \emptyset, \tau)]\!](\rho) = [\![\tau]\!]_{\leftarrow}(\rho) = [\![\mathtt{reverse}(\tau)]\!]_{\rightarrow}(\rho)$.

4.4. Correctness

LEMMA 4 (One-step semantic preservation). *The rewriting rules preserve the semantics of well-formed triples: if $S_1 \triangleright S_2$ and $\vdash S_1$, then $[\![S_1]\!](\rho) \equiv [\![S_2]\!](\rho)$ for all environments ρ.*

Proof. By case analysis on the rule used to derive $S_1 \triangleright S_2$. We write $S_1 = (\mu, \sigma, \tau)$.

– Rule [Nop]: use the fact that $\rho[r \leftarrow \rho(r)] = \rho$.

– Rules [Start] and [Push]: use the order independence lemma 2.

– Rule [Loop]: exploit the fact that μ and σ do not use the temporary register introduced by the rule.

– Rule [Pop]: the expected result follows from

$$[\![(s_n \mapsto d_n) \bullet \mu \bullet \sigma]\!]_{//}([\![\tau]\!]_{\leftarrow}(\rho)) = [\![\mu \bullet \sigma]\!]_{//}([\![(s_n \mapsto d_n) \bullet \tau]\!]_{\leftarrow}(\rho))$$

This equality holds provided that register d_n does not appear as a source in $\mu \bullet \sigma$. For μ, this is ensured by the first premise. For σ, this follows by induction from the fact that σ is a path. The base case comes from the premise $d_n \neq s_0$. The inductive case is provided by the windmill structure of σ.

– Rule [Last]: special case of rule [Pop].

This lemma is easily extended to $\triangleright^*$, the reflexive transitive closure of $\triangleright$:

LEMMA 5 (Semantic preservation). *Several steps of rewriting preserve the semantics of well-formed triples: if* $S_1 \triangleright^* S_2$ *and* $\vdash S_1$*, then* $[\![S_1]\!](\rho) \equiv [\![S_2]\!](\rho)$ *for all environments* ρ*.*

Proof. By induction on the number of steps.

As a corollary, we obtain the main semantic preservation theorem of this section.

THEOREM 1 (Correctness of $\triangleright^*$). *If* μ *is a windmill containing no temporary register, and if the triple* $(\mu, \emptyset, \emptyset)$ *can be rewritten into* $(\emptyset, \emptyset, \tau)$ *then the parallel execution of* μ *is equivalent to the sequential execution of* `reverse`(τ)*:*

$$(\mu, \emptyset, \emptyset) \triangleright^* (\emptyset, \emptyset, \tau) \;\Rightarrow\; [\![\mu]\!]_{//}(\rho) \equiv [\![\texttt{reverse}(\tau)]\!]_{\rightarrow}(\rho).$$

5. Deterministic specification

The next step towards an effective algorithm consist in *determinising* the inductive rules of section 4 by ensuring that at most one rule matches a given state, and by specifying which edge to extract out of the *to-move* list in the rules [Nop], [Push] and [Start].

$$\frac{}{((r \mapsto r) \bullet \mu, \emptyset, \tau) \hookrightarrow (\mu, \emptyset, \tau)} \quad [\text{Nop}']$$

$$\frac{s \neq d}{((s \mapsto d) \bullet \mu, \emptyset, \tau) \hookrightarrow (\mu, (s \mapsto d), \tau)} \quad [\text{Start}']$$

The first two rules treat the case of an empty *being-moved* list and consider only the first element on the left of the *to-move* list μ.

$$\frac{\texttt{NoRead}(\mu_1, d)}{\begin{array}{l}(\mu_1 \bullet (d \mapsto r) \bullet \mu_2, (s \mapsto d) \bullet \sigma, \tau) \\ \quad \hookrightarrow (\mu_1 \bullet \mu_2, (d \mapsto r) \bullet (s \mapsto d) \bullet \sigma, \tau)\end{array}} \quad [\text{Push}']$$

The [Push′] rule treats the case in which the analysed edge has a successor in the μ list. It imposes to consider the leftmost successor found in μ.

$$\frac{\texttt{NoRead}(\mu, r_0)}{\begin{array}{l}(\mu, (s \mapsto r_0) \bullet \sigma \bullet (r_0 \mapsto d), \tau) \\ \quad \hookrightarrow (\mu, \sigma \bullet (T(r_0) \mapsto d), (s \mapsto r_0) \bullet (r_0 \mapsto T(r_0)) \bullet \tau)\end{array}} \quad \text{[LoopPop]}$$

$$\frac{\texttt{NoRead}(\mu, d_n) \ \wedge \ d_n \neq s_0}{\begin{array}{l}(\mu, (s_n \mapsto d_n) \bullet \sigma \bullet (s_0 \mapsto d_0), \tau) \\ \quad \hookrightarrow (\mu, \sigma \bullet (s_0 \mapsto d_0), (s_n \mapsto d_n) \bullet \tau)\end{array}} \quad \text{[Pop}'\text{]}$$

These two rules treat the case where the analysed edge has no successor in the *to-move* list, and the *being-moved* list contains at least two elements. The [LoopPop] rule is the [Pop] rule in the case of a cycle. Remark that this rule, unlike the corresponding nondeterministic rules, pops the analysed edge. This particularity ensures that every rule moves at least one edge. The termination proof in section 6 uses this fact.

$$\frac{\texttt{NoRead}(\mu, d)}{(\mu, (s \mapsto d), \tau) \hookrightarrow (\mu, \emptyset, (s \mapsto d) \bullet \tau)} \quad \text{[Last}'\text{]}$$

The [Last′] rule treats the final case where the *being-moved* list contains only one element.

Since these deterministic rules are instances or combinations of the nondeterministic ones, the following two lemmas are trivial.

LEMMA 6 (Inclusion $\hookrightarrow \subseteq \rhd^*$). *If $S_1 \hookrightarrow S_2$, then $S_1 \rhd^* S_2$.*

Proof. By case analysis. The [LoopPop] case uses the transitivity of $\rhd^*$.

LEMMA 7 (Inclusion $\hookrightarrow^* \subseteq \rhd^*$). *If $S_1 \hookrightarrow^* S_2$, then $S_1 \rhd^* S_2$.*

Proof. By induction on the number of steps.

Notice that the deterministic rules implement a strategy that is slightly different from that of the imperative algorithm. Indeed, during the imperative algorithm, when a temporary register is used to break a cycle, the analysed edge can still have a successor. This is not the case for the [LoopPop] rule.

Semantic preservation for the deterministic rules follows immediately from theorem 1 and lemma 7. However, to obtain an algorithm, we still need to prove that the rules are indeed deterministic and that a normal form (i.e. the final state $(\emptyset, \emptyset, \tau)$) always exists. To do so, we now give a functional presentation of the deterministic rules.

6. The functional algorithm

From the deterministic rules in section 5, it is easy to build a function `stepf` that performs one step of rewriting. Compiling a parallel move problem, then, corresponds to iterating `stepf` until a final state is reached:

```
pmov S =
    match S with
    | (∅, ∅, _) ⇒ S
    | _ ⇒ pmov (stepf S)
    end
```

However, such a function definition is not accepted by Coq, because it is not structurally recursive: the recursive call is performed on `stepf`(S) and not on a structural sub-term of S. To define the `pmov` function, we will use the Coq command `Function`, which provides a simple mechanism to define a recursive function by well-founded recursion (Balaa and Bertot, 2002; Barthe et al., 2006). To this end, we first define the `stepf` function that executes one computation step, then a measure function which will be used to justify termination, and finally the theorem establishing that the measure decreases at each recursive call.

6.1. The one-step rewriting function

We define the one-step rewriting function `stepf` in a way that closely follows the deterministic rules of section 5. It takes a state S as argument and returns a state S' such that $S \hookrightarrow S'$. The function is defined by case analysis in figure 3.

The first case (line 3) does not correspond to any of the inductive rules: it detects a final state $(\emptyset, \emptyset, \tau)$ and returns it unchanged. The other cases correspond to the deterministic rules indicated in comments.

The `reg_eq` function decides whether two registers are equal or not. The `split_move` function takes as arguments a mill t, a list of edges, and a register d. If this register is the source of an edge from list t, the function returns this edge – $(d, \_)$, and the two remaining sub-lists. This breaks the t list into $t_1 \bullet (d, r) \bullet t_2$ like the `[Push]` rule does.

Finally, the `replace_last_source` function takes as argument a list b of edges and a register r, and replaces the source of the last edge of b with r: $hb \bullet (s \mapsto d)$ becomes $hb \bullet (t \mapsto d)$ (confer the [LoopPop] rule).

Since the definition of the `stepf` function is close to the deterministic specification, the following theorem is easily proved:

```
Function stepf (st: state) : state :=
  match st with
  | State ∅ ∅ _ ⇒ st                                  (* final state *)
  | State ((s, d) • tl) ∅ l ⇒
      if reg_eq s d
      then State tl ∅ l                               (* s = d; rule [Nop] *)
      else State tl ((s, d) • ∅) l                    (* s ≠ d; rule [Start] *)
  | State t ((s, d) • b) l ⇒
      match split_move t d with
      | Some (t1, r, t2) ⇒                            (* t = t1 • (d ↦ r) • t2 *)
          State (t1 • t2)                             (* rule [Push] *)
                ((d, r) • (s, d) • b)
                l
      | None ⇒
          match b with
          | ∅ ⇒ State t ∅ ((s, d) • l)                (* rule [Last] *)
          | _ ⇒
              if is_last_source d b                   (* check if b = _ • (d ↦ _) *)
              then State t                            (* rule [LoopPop] *)
                         (replace_last_source (temp d) b)
                         ((s, d) • (d, temp d) • l)
              else State t b ((s, d) • l)             (* rule [Pop] *)
          end
      end
  end.
```

Figure 3. The one-step rewriting function

LEMMA 8 (Compatibility of `stepf` and $\hookrightarrow$). *$S \hookrightarrow$ `stepf`(S) if S is not a final state (i.e. not of the form $(\emptyset, \emptyset, \tau)$).*

Proof. By case analysis on the shape of S and application of the corresponding rules for the $\hookrightarrow$ relation.

6.2. The general recursive function

The general recursive function consists in iterating the `stepf` function. To ensure termination of this function, we need a nonnegative integer measure over states that decreases at each call to the `stepf` function, except in the final case $(\emptyset, \emptyset, \tau)$. Examination of the deterministic rewriting rules and of the lengths of the *to-move* and *being-moved* lists reveals that:

– either an edge is removed from the *to-move* list and and the *being-moved* list is unchanged (rule [Nop]);

- or an edge is transferred from the *to-move* list to the *being-moved* list (rules [Start] and [Push]);
- or the *to-move* list remains the same while an edge is transferred from the *being-moved* list to the *moved* list (rules [LoopPop], [Pop], and [Last]).

Therefore, either one of the *to-move* or *being-moved* lists loses one element, or one element is transferred from the former to the latter. Decrease is achieved by giving more weight to the *to-move* edges, as the following `meas` function does:

DEFINITION 4 (The triple measure).
$\texttt{meas}(\mu, \sigma, \tau) = 2 \times \texttt{length}(\mu) + \texttt{length}(\sigma)$.

LEMMA 9 (Measure decreases with $\hookrightarrow$).
If $S_1 \hookrightarrow S_2$*, then* $\texttt{meas}(S_2) < \texttt{meas}(S_1)$.

Proof. By case analysis on the rules of the $\hookrightarrow$ relation and computation of the corresponding measures.

Combining this lemma with lemma 8 (compatibility of `stepf` with $\hookrightarrow$), we obtain:

LEMMA 10 (Measure decreases with `stepf`). *If* S *is not a final state, i.e. not of the form* $(\emptyset, \emptyset, \tau)$*, then* $\texttt{meas}(\texttt{stepf}(S)) < \texttt{meas}(S)$.

We can then define the iterate of `stepf` using the `Function` command of Coq version 8:

```
Function pmov (S: State) {measure meas st} : state :=
    if final_state S then S else pmov (stepf S).
```

where `final_state` is a boolean-valued function returning `true` if its argument is of the form $(\emptyset, \emptyset, \tau)$ and `false` otherwise. Coq produces a proof obligation requiring the user to show that the recursive call to `pmov` is actually decreasing with respect to the `meas` function. This obligation is trivially proved by lemma 10. Coq then generates and automatically proves the following functional induction principle that enables us to reason over the `pmov` function:

$$\begin{array}{rl} & (\forall S,\ \texttt{final\_state}(S) = \texttt{true} \ \Rightarrow\ P(S,S)) \\ \wedge & (\forall S_1, S_2,\ \texttt{final\_state}(S_1) = \texttt{false} \ \wedge\ P(\texttt{stepf}(S_1), S_2) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow\ P(S_1, S_2)) \\ \Rightarrow & (\forall S, P(S, \texttt{pmov(S)})) \end{array}$$

Using this induction principle and lemma 8, we obtain the correctness of `pmov` with respect to the deterministic specification.

LEMMA 11. *For all initial states S,* $\texttt{pmov}(S)$ *is a final state* $(\emptyset, \emptyset, \tau)$ *such that* $S \hookrightarrow^* (\emptyset, \emptyset, \tau)$.

6.3. The compilation function

To finish this development, we define the main compilation function, taking a parallel move problem as input and returning an equivalent sequence of elementary moves.

```
Definition parmove (mu: moves) : moves :=
  match pmov (State mu ∅ ∅ ) with
  | State _ _ tau ⇒ reverse tau
  end.
```

The semantic correctness of `parmove` follows from the previous results.

THEOREM 2. *Let μ be a parallel move problem. If μ is a mill and does not mention temporary registers, then* $[\![\texttt{parmove}(\mu)]\!]_{\rightarrow}(\rho) \equiv [\![\mu]\!]_{//}(\rho)$ *for all environments ρ.*

Proof. By definition of `parmove` and lemma 11, we have $\texttt{parmove}(\mu) = \texttt{reverse}(\tau)$ and $(\mu, \emptyset, \emptyset) \hookrightarrow^* (\emptyset, \emptyset, \tau)$. By lemma 7, this entails $(\mu, \emptyset, \emptyset) \rhd^* (\emptyset, \emptyset, \tau)$. The result follows from theorem 1.

Here is an alternate formulation of this theorem that can be more convenient to use.

THEOREM 3. *Let* $\mu = (s_1 \mapsto d_1) \bullet \cdots \bullet (s_n \mapsto d_n)$ *be a parallel move problem. Assume that the d_i are pairwise distinct, and that $s_i \notin \mathcal{T}$ and $d_i \notin \mathcal{T}$ for all i. Let* $\tau = \texttt{parmove}(\mu)$. *For all initial environments ρ, the environment* $\rho' = [\![\tau]\!]_{\rightarrow}(\rho)$ *after executing τ sequentially is such that*

1. *$\rho'(d_i) = \rho(s_i)$ for all $i = 1, \ldots, n$;*

2. *$\rho'(r) = \rho(r)$ for all registers $r \notin \{d_1, \ldots, d_n\} \cup \mathcal{T}$.*

Proof. By theorem 2, we know that $\rho' \equiv [\![\mu]\!]_{//}(\rho)$. For (1), since $d_i \notin \mathcal{T}$, we have $\rho'(d_i) = [\![\mu]\!]_{//}(\rho)(d_i) = \rho(s_i)$ by lemma 2. For (2), we have $\rho'(r) = [\![\mu]\!]_{//}(\rho)(r)$, and an easy induction on μ shows that $[\![\mu]\!]_{//}(\rho)(r) = \rho(r)$ for all registers r that are not destination of μ.

Finally, we used the Coq extraction mechanism (Letouzey, 2003) to automatically generate executable Caml code from the definition of the `parmove` function, thus obtaining a verified implementation of the parallel move compilation algorithm that can be integrated in the Compcert compiler back-end.

7. Syntactic properties of the compilation algorithm

In this section, we show a useful syntactic property of the sequences of moves generated by the `parmove` compilation function: the registers involved in the generated moves are not arbitrary, but either appear in the initial parallel move problem, or are results of the temporary-generating function T.

More formally, let $\mu_0 = (s_1 \mapsto d_1) \bullet \cdots \bullet (s_n \mapsto d_n)$ be the initial parallel move problem. We define a property $P(s \mapsto d)$ of elementary moves $(s \mapsto d)$ by the following inference rules:

$$\frac{(s \mapsto d) \in \mu_0}{P(s \mapsto d)} \qquad \frac{P(s \mapsto d)}{P(T(s) \mapsto d)} \qquad \frac{P(s \mapsto d)}{P(s \mapsto T(s))}$$

We extend this property pointwise to lists l of moves and to states:

$$P(l) \stackrel{\text{def}}{=} \forall (s \mapsto d) \in l,\ P(s \mapsto d) \qquad P(\mu, \sigma, \tau) \stackrel{\text{def}}{=} P(\mu) \wedge P(\sigma) \wedge P(\tau)$$

Obviously, the property P holds for μ_0 and therefore for the initial state $(\mu_0, \emptyset, \emptyset)$. Moreover, the property is preserved by every rewriting step.

LEMMA 12. *If $S_1 \triangleright S_2$ and $P(S_1)$ hold, then $P(S_2)$ holds.*

Proof. By examination of the nondeterministic rewriting rules. The result is obvious for all rules except [Loop], since these rules do not generate any new moves. The [Loop] rule replaces a move $(s \mapsto d)$ that satisfies P with the two moves $(T(s) \mapsto d)$ and $(s \mapsto T(s))$. The new moves satisfy P by application of the second and third inference rules defining P.

It follows that the result of `parmove`(μ_0) satisfy property P.

LEMMA 13. *For all moves $(s \mapsto d) \in$ `parmove`(μ_0), the property $P(s \mapsto d)$ holds.*

Proof. Follows from lemmas 7, 11 and 12.

As a first use of this lemma, we can show that for every move in the generated sequence $\texttt{parmove}(\mu_0)$, the source of this move is either a source of μ_0 or a temporary, and similarly for the destination.

LEMMA 14. *For all moves* $(s \mapsto d) \in \texttt{parmove}(\mu_0)$*, we have* $s \in \{s_1, \ldots, s_n\} \cup \mathcal{T}$ *and* $d \in \{d_1, \ldots, d_n\} \cup \mathcal{T}$*.*

Proof. We show that $P(s \mapsto d)$ implies $s \in \{s_1, \ldots, s_n\} \cup \mathcal{T}$ and $d \in \{d_1, \ldots, d_n\} \cup \mathcal{T}$ by induction on a derivation of $P(s \mapsto d)$, then conclude with lemma 13.

Another application of lemma 13 is to show type preservation for the compilation of parallel moves. Assume that every register r has an associated type $\Gamma(r)$. We say that a list of moves l is well typed if $\Gamma(s) = \Gamma(d)$ for all $(s \mapsto d) \in l$.

LEMMA 15 (Type preservation). *Assume that temporaries are generated in a type-preserving manner:* $\Gamma(T(s)) = \Gamma(s)$ *for all registers* s*. If* μ_0 *is well typed, then* $\texttt{parmove}(\mu_0)$ *is well typed.*

Proof. We show that $P(s \mapsto d)$ implies $\Gamma(s) = \Gamma(d)$ by induction on a derivation of $P(s \mapsto d)$, then conclude using lemma 13.

8. Extension to overlapping registers

So far, we have assumed that two distinct registers never overlap: assigning one does not change the value of the other. This is not always the case in practice. For instance, some processor architectures expose hardware registers that share some of their bits. In the IA32 architecture, for example, the register `AL` refers to the low 8 bits of the register `EAX`. Assigning to `AL` therefore modifies the value of `EAX`, and conversely.

Overlap also occurs naturally when the “registers” manipulated by the parallel move compilation algorithm include memory locations, such as variables that have been spilled to memory during register allocation, or stack locations used for parameter passing. For example, writing a 32-bit integer at offset δ in the stack also changes the values of the stack locations at offsets $\delta + 1$, $\delta + 2$ and $\delta + 3$.

It is not straightforward to generalize the parallel move compilation algorithm to the case where source and destination registers can overlap arbitrarily. We will not attempt to do so in this section, but set out to show a weaker, but still useful result: the unmodified parallel move algorithm produces correct sequences of elementary assignments even if registers can in general overlap, provided destinations and sources do not overlap.

8.1. Formalising overlap

In the non-overlapping case, two registers r_1 and r_2 are either identical $r_1 = r_2$ or different $r_1 \neq r_2$. When registers can overlap, we have three cases: r_1 and r_2 are either identical ($r_1 = r_2$) or completely disjoint (written $r_1 \perp r_2$) or different but partially overlapping (written $r_1 \bowtie r_2$).

We assume given a disjointness relation $\perp$ over $\mathcal{R} \times \mathcal{R}$, which must be symmetric and such that $r_1 \perp r_2 \;\Rightarrow\; r_1 \neq r_2$. We define partial overlap $r_1 \bowtie r_2$ as $(r_1 \neq r_2) \;\wedge\; \neg(r_1 \perp r_2)$.

In the non-overlapping case, the semantics of an assignment of value v to register r is captured by the update operation $\rho[r \leftarrow v]$, characterized by the "good variable" property:

$$\rho[r \leftarrow v] = \begin{cases} r \mapsto v & \\ r' \mapsto \rho(r') & \text{if } r' \neq r. \end{cases}$$

To account for the possibility of overlap, we define the weak update operation $\rho[r \Leftarrow v]$ by the following "weak good variable" property:

$$\rho[r \Leftarrow v] = \begin{cases} r \mapsto v & \\ r' \mapsto \rho(r') & \text{if } r' \perp r \\ r' \mapsto \text{undefined} & \text{if } r' \bowtie r. \end{cases}$$

As suggested by the fatter arrow $\Leftarrow$, weak update sets the target register r to the specified value v, but causes "collateral damage" on registers r' that partially overlap with r: their values after the update are undefined. Only registers r' that are disjoint from r keep their old values.

Using weak update instead of update, the semantics of a sequence τ of elementary moves becomes

$$\begin{aligned} [\![\emptyset]\!]_{\Rightarrow}(\rho) &= \rho \\ [\![(s \mapsto d) \bullet \tau]\!]_{\Rightarrow}(\rho) &= [\![\tau]\!]_{\Rightarrow}(\rho[d \Leftarrow \rho(s)]) \end{aligned}$$

8.2. Effect of overlap on the parallel move algorithm

In this section, we consider a parallel move problem $\mu = (s_1 \mapsto d_1) \bullet \cdots \bullet (s_n \mapsto d_n)$ and assume that the sources $s_1, \ldots, s_n$ and the destinations $d_1, \ldots, d_n$ satisfy the following hypotheses:

1. No temporaries: $s_i \perp t$ and $d_i \perp t$ for all i, j and $t \in \mathcal{T}$.

2. Destinations are pairwise disjoint: $d_i \perp d_j$ if $i \neq j$.
3. Sources and destinations do not partially overlap: $s_i \neq d_j \;\Rightarrow\; s_i \perp d_j$ for all i, j.
4. Temporaries do not partially overlap with any register: $t \neq r \;\Rightarrow\; t \perp r$ for all $t \in \mathcal{T}, r \in \mathcal{R}$.

Hypotheses (1), (2) and (4) ensure that assigning a destination or a temporary preserves the values of other destinations and temporaries. Hypotheses (1), (3) and (4) ensure that assigning a destination or a temporary preserves the values of all source registers.

These hypotheses are easily satisfied in our application scenario within the Compcert compiler. Hardware registers never partially overlap in the target architecture (the PowerPC processor). Properties of the register allocator ensure that the only possibility for partial overlap is between stack locations used as destinations for parameter passing, but such overlap is avoided by the calling conventions used.

Since $r_1 \perp r_2 \;\Rightarrow\; r_1 \neq r_2$, hypothesis (1) ensures that no temporary occurs in sources and destinations, and hypothesis (2) ensures that μ is a mill. Therefore, the initial state $(\mu, \emptyset, \emptyset)$ is well-formed.

We now set out to prove a correctness result for the sequence of elementary moves $\tau = \texttt{parmove}(\mu)$ produced by the parallel move compilation function. Namely, we wish to show that the final values of the destinations d_i are the initial values of the sources s_i, and that all registers disjoint from the destinations and from the temporaries keep their initial values. To this end, we define the following relation $\cong$ between environments:

$$\rho_1 \cong \rho_2 \;\stackrel{\text{def}}{=}\; (\forall r \notin \mathcal{D},\; \rho_1(r) = \rho_2(r))$$

where $\mathcal{D}$ is the set of registers that partially overlap with one of the destinations:

$$\mathcal{D} \;\stackrel{\text{def}}{=}\; \{r \mid \exists i, r \bowtie d_i\}$$

In other terms, $\rho_1 \cong \rho_2$ holds if ρ_1 and ρ_2 assign the same values to registers, except perhaps those registers that could be set to an undefined value as a side-effect of assigning one of the destinations.

LEMMA 16. *$r \notin \mathcal{D}$ if r is one of the destinations d_i, or one of the sources s_i, or a temporary $t \in \mathcal{T}$.*

Proof. Follows from hypotheses (1) to (4).

Using the relation $\cong$, we can relate the effect of executing moves using normal, overlap-unaware update and weak, overlap-aware update.

LEMMA 17. *Consider an elementary move* $(s \mapsto d)$ *where* $s \in \{s_1, \ldots, s_n\} \cup \mathcal{T}$ *and* $d \in \{d_1, \ldots, d_n\} \cup \mathcal{T}$*. If* $\rho_1 \cong \rho_2$*, then* $\rho_1[d \leftarrow \rho_1(s)] \cong \rho_2[d \Leftarrow \rho_2(s)]$.

Proof. We need to show that

$$\rho_1[d \leftarrow \rho_1(s)]\ (r) = \rho_2[d \Leftarrow \rho_2(s)]\ (r) \qquad (*)$$

for all registers $r \notin \mathcal{D}$. By definition of $\mathcal{D}$, either $r = d_i$ or $r \perp d_i$ for all destinations d_i. By hypothesis (4), either $r = t$ or $r \perp t$ for all temporaries $t \in \mathcal{T}$. Therefore, either $r = d$ or $r \perp d$.

In the first case $r = d$, the left-hand side of (*) is equal to $\rho_1(s)$ and the right-hand side to $\rho_2(s)$. We do have $\rho_1(s) = \rho_2(s)$ by hypothesis $\rho_1 \cong \rho_2$ and the fact that $s \notin \mathcal{D}$ by lemma 16.

In the second case $r \perp d$, using the good variable property and the fact that $r \neq d$, the left-hand side of (*) is equal to $\rho_1(d)$. Using the weak good variable property, the right-hand side is $\rho_2(d)$. The equality $\rho_1(d) = \rho_2(d)$ follows from hypothesis $\rho_1 \cong \rho_2$.

The previous lemma extends to sequences of elementary moves, performed with normal updates on one side and with weak updates on the other side.

LEMMA 18. *Let* τ *be a sequence of moves such that for all* $(s \mapsto d) \in \tau$*,* $s \in \{s_1, \ldots, s_n\} \cup \mathcal{T}$ *and* $d \in \{d_1, \ldots, d_n\} \cup \mathcal{T}$*. If* $\rho_1 \cong \rho_2$*, then* $[\![\tau]\!]_{\rightarrow}(\rho_1) = [\![\tau]\!]_{\Rightarrow}(\rho_2)$.

Proof. By structural induction on τ, using lemma 17.

THEOREM 4. *Let* $\mu = (s_1 \mapsto d_1) \bullet \cdots \bullet (s_n \mapsto d_n)$ *be a parallel move problem that satisfies hypotheses (1) to (4). Let* $\tau = \texttt{parmove}(\mu)$*. For all environments* ρ*, writing* $\rho' = [\![\tau]\!]_{\Rightarrow}(\rho)$*, we have*

1. $\rho'(d_i) = \rho(s_i)$ *for all* $i = 1, \ldots, n$*;*
2. $\rho'(r) = \rho(r)$ *for all registers* r *disjoint from the destinations* d_i *and from the temporaries* $\mathcal{T}$*.*

Proof. Define $\rho_1 = [\![\tau]\!]_{\rightarrow}(\rho)$. By theorem 3, we have $\rho_1(d_i) = \rho(s_i)$ for all i, and $\rho_1(r) = \rho(r)$ for all $r \notin \{d_1, \ldots, d_n\} \cup \mathcal{T}$. By lemma 14, every move $(s \mapsto d) \in \tau$ is such that s is either one of the sources s_i or a temporary, and d is either one of the destinations d_i or a temporary. Since $\rho \cong \rho$ holds trivially, lemma 18 applies and shows that $\rho_1 \cong \rho'$.

Let d_i be one of the destinations. We have $\rho'(d_i) = \rho_1(d_i)$ since $d_i \notin \mathcal{D}$ by lemma 16. Moreover, $\rho_1(d_i) = \rho(s_i)$. The expected result follows.

Let r be a register disjoint from the destinations d_i and from the temporaries $\mathcal{T}$. By definition of $\mathcal{D}$, $r \notin \mathcal{D}$, therefore $\rho'(r) = \rho_1(r)$. Moreover, $\rho_1(r) = \rho(r)$ since $r \notin \{d_1, \ldots, d_n\} \cup \mathcal{T}$. The expected result follows.

9. Conclusions

Using the Coq proof assistant, we have proved the correctness and termination of a compilation algorithm that serialises parallel moves. The main difficulty of this development was to find an appropriate set of atomic transitions between intermediate states of the algorithm, and prove that they preserve semantics; once this is done, the proof of the algorithm proper follows easily by refinements. The Coq development is relatively small: 570 lines of specifications and 625 lines of proof scripts.

The approach followed in this article enabled us to slightly improve the imperative algorithm from section 3 used as a starting point. In particular, the functional algorithm looks only for cycles involving the edge that started the recursion of the `move_one` function (i.e. the last element of the *being-moved* list, whereas the imperative algorithm looks for cycles anywhere in this list. The correctness proof for the $\triangleright$ relation implies that looking for the cycle from the bottom of the *being-moved* stack is sufficient: in the case for the [Pop] rule, we proved that all elements of the *being-moved* list except the last one cannot have the destination d_n as source.

While specified in terms of parallel moves, our results can probably be extended to parallel assignments $(x_1, \ldots, x_n) := (e_1, \ldots, e_n)$ where every expression e_i mentions at most one of the variables x_j.

Although efficient in practice, neither the initial imperative algorithm nor the functional algorithm proved in this paper are optimal in the number of elementary moves produced. Consider for instance the parallel move $(r_3, r_2, r_1) := (r_1, r_1, r_2)$. The algorithms generate a sequence of 4 moves: $r_3 := r_1;\ t := r_2;\ r_2 := r_1; r_1 := t$, where t is a temporary. However, the effect can be achieved in 3 moves, using the destination register r_3 to break the cycle: $r_3 := r_1;\ r_1 := r_2;\ r_2 := r_3$. Preliminary investigations suggest that the nondeterministic specification of section 4 can be extended with one additional rule to support this use of a destination to break cycles. However, the functional algorithm would become much more complex and require backtracking, possibly leading to exponential complexity.

A direction for further work is to prove the correctness of an imperative, array-based formulation of the parallel move compilation algorithm, similar to the algorithm given in section 3. We are considering using the Why tool (Filliâtre, 2007; Filliâtre, 2003) to conduct this proof. It raises several difficulties. The two nested loops and the number of arrays needed (`src, dst` and `status`) imply large invariants. Moreover the proof obligations generated by Why are huge and hard to work with. Furthermore, due to the side effects over arrays, delicate non-aliasing lemmas must be proved.

References

Appel, A. W.: 1992, *Compiling with continuations.* Cambridge University Press.

Balaa, A. and Y. Bertot: 2002, 'Fonctions récursives générales par itération en théorie des types'. In: *Journées Francophones des Langages Applicatifs 2002*. INRIA.

Barthe, G., J. Forest, D. Pichardie, and V. Rusu: 2006, 'Defining and reasoning about recursive functions: a practical tool for the Coq Proof Assistant.'. In: *Proc. 8th Int. Symp. on Functional and Logic Programming (FLOPS'06)*, Vol. 3945 of *Lecture Notes in Computer Science*. pp. 114–129, Springer.

Bertot, Y. and P. Castéran: 2004, *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*, EATCS Texts in Theoretical Computer Science. Springer.

Bertot, Y., B. Grégoire, and X. Leroy: 2006, 'A structured approach to proving compiler optimizations based on dataflow analysis'. In: *Types for Proofs and Programs, Workshop TYPES 2004*, Vol. 3839 of *Lecture Notes in Computer Science*. pp. 66–81, Springer.

Blazy, S., Z. Dargaye, and X. Leroy: 2006, 'Formal verification of a C compiler front-end'. In: *FM 2006: Int. Symp. on Formal Methods*, Vol. 4085 of *Lecture Notes in Computer Science*. pp. 460–475, Springer.

Coq development team: 1989–2007, 'The Coq proof assistant'. Software and documentation available at `http://coq.inria.fr/`.

Filliâtre, J.-C.: 2003, 'Verification of non-functional programs using interpretations in type theory'. *Journal of Functional Programming* **13**(4), 709–745.

Filliâtre, J.-C.: 2003–2007, 'The Why software verification tool'. Software and documentation available at `http://why.lri.fr/`.

Leroy, X.: 2006, 'Formal certification of a compiler back-end, or: programming a compiler with a proof assistant'. In: *33rd symposium Principles of Programming Languages*. pp. 42–54, ACM Press.

Letouzey, P.: 2003, 'A new extraction for Coq'. In: *Types for Proofs and Programs, Workshop TYPES 2002*, Vol. 2646 of *Lecture Notes in Computer Science*. pp. 200–219, Springer.

May, C.: 1989, 'The parallel assignment problem redefined'. *IEEE Transactions on Software Engineering* **15**(6), 821–824.

Sethi, R.: 1973, 'A note on implementing parallel assignment instructions'. *Information Processing Letters* **2**(4), 91–95.

Welch, P. H.: 1983, 'Parallel assignment revisited'. *Software Practice and Experience* **13**(12), 1175–1180.