A more precise, more correct stack and register model for CompCert

Gergö Barany Inria, France gergo.barany@inria.fr

1 Introduction

CompCert is a formally verified C compiler implemented and proved correct using the Coq proof assistant. Its correctness proof covers the transformation steps from ASTs for a simplified C-like language called Clight all the way to ASTs representing target-specific assembly code.

The present work came out of the plan to add a CompCert backend for the Kalray MPPA architecture. In this architecture, there are 64 general-purpose 32-bit registers R_i. Larger values, such as double-precision floating-point values, are stored in adjacent pairs of 32-bit registers, denoted $R_0R_1, R_2R_3, \ldots$. This notion, in particular the fact that modifying some register also modifies the values stored in its subregisters or superregister, is not modeled in CompCert.

The proposed talk describes ongoing work on modeling this kind of register aliasing in CompCert, and how exploring the design space uncovered long-standing errors in CompCert's semantic models.

2 Values and registers in CompCert

All semantics definitions inside CompCert use a notion of symbolic values as given by the following inductive data type:

```
Inductive val := Vundef | Vint(i : int) | Vlong(l : int64) | Vsingle(s : float32) | Vfloat(f : float64) | Vptr(...).
```

Values have types, with (simplifying slightly) Vint and Vsingle values being of type Tany32 and Vlong and Vfloat values of type Tany64. Pointer values (Vptr) have a target-dependent size and thus type. Vundef can be of any type.

A given architecture's general-purpose registers are constructors of an inductive datatype, e. g., for the MPPA we might define:

```
Inductive mreg := R0 | R1 | R2 | ... | R0R1 | R2R3 | ....
```

Registers have associated types, as do stack slots, which are modeled using a similar symbolic representation. Registers and stack slots are grouped together under a type loc of locations storing values. The contents of CPU registers and the current stack frame at any point in the execution of a program are represented by the type Locmap.t, defined as functions from locations to values: loc → val. Following standard functional programming practice, read access to such a store is simply function application:

```
Definition get (l : loc) (m : Locmap.t) := m l.
```

Similarly, writes into a register or a stack slot (set) are modeled as function update.

This model allows us to prove general, desirable properties of value stores. We illustrate their flavor with a lemma stating that if we write a value v into a register r and then read r immediately, we will get back the value v we stored:

```
Lemma gss_reg : forall (r : mreg) (v : val) (m : Locmap.t), (set (R r) v m) (R r) = v.
```

3 Modeling subregisters

We describe two register aliasing models we tried but discarded, and a third one under development.

3.1 Undefine aliasing registers

Modifying a superregister changes the values stored in the subregisters and vice versa. This is easy to model by defining all registers as essentially independent locations, but changing the set operation on a register r to first set all of r's subregisters and superregisters to Vundef before storing a value into r itself.

This model is attractive due to its simplicity, but it is too weak to model interactions with spilling correctly: Register values must sometimes be stored to the stack and reloaded later. For performance reasons, we only ever want to spill superregisters and ensure that restoring them restores the values of their subregisters as well. However, there is nothing in the model that would allow to conclude this. In fact, restoring the superregister in this model would always set the subregisters' values to Vundef.

3.2 Pairs of words as values

Another approach would avoid trying to model subregisters in the register file. Instead, a 64-bit register holding two independent 32-bit halves could be modeled as a register holding a *value* composed of two independent halves. That is, we could model 32-bit values using a type word and add constructors to val to model a 64-bit 'pair' value holding two such words. As subregisters do not exist in this formulation, spilling and restoring 64-bit registers works as expected, even when they hold pairs.

The main problem with this approach is its interaction with proof principles within CompCert. Its simulation proofs use a relation $\leq_{\mathsf{def}}$, defined as the smallest relation such that $\mathsf{Vundef} \leq_{\mathsf{def}} \mathsf{v}$ and $\mathsf{v} \leq_{\mathsf{def}} \mathsf{v}$ for all values v. In particular, CompCert proofs rely heavily on the reasoning principle that if $\mathsf{v}_1 \leq_{\mathsf{def}} \mathsf{v}_2$, then necessarily $\mathsf{v}_1 = \mathsf{Vundef}$ or $\mathsf{v}_1 = \mathsf{v}_2$. Pair values would have to allow for the case where only one half of the pair is defined, but then the $\leq_{\mathsf{def}}$ relation would have to be extended to three levels, e. g., $\mathsf{Vundef} \leq_{\mathsf{def}} \mathsf{Vpair}\ (\mathsf{Wint\ i})\ \mathsf{Wundef} \leq_{\mathsf{def}} \mathsf{Vpair}\ (\mathsf{Wint\ i})\ (\mathsf{Wint\ j})$. This would require large-scale re-engineering of many proofs in CompCert, and we abandoned it as impractical.

While working on this approach we realized that the lemma gss_reg above, essentially unchanged in the CompCert source code since 2006, is false: The lemma claims that *any* value can be written into *any* register and read back. In particular, it claims that 64-bit values can be stored in 32-bit registers without loss of information. In the register model we finally adopted, this lemma was modified to express that the value stored must be of a correct type for the register, otherwise the value read back is Vundef.

3.3 Registers (and stack slots) as blocks of bytes

We finally decided on a 'low-level' model treating the register file and stack slots as blocks of bytes, addressable like random access memory. Every register has an 'address' into this block as well as a size in bytes. The two 32-bit subregisters of a 64-bit superregister correspond directly to the two 4-byte halves of an 8-byte region in the block. Most operations in CompCert are defined in terms of values, not bytes; usual read and write accesses therefore use decoding and encoding functions that are also used for modeling main memory accesses. However, register copies as well as stack loads and stores copy bytes directly. This ensures that spilled and restored registers can be decoded to the same values as before.

The basic inconvenience of this approach is that some simulation proofs had to be reformulated at a lower level. Where previously states in the target and source language of a transformation were matched using a relation expressing that corresponding *values* in each state were in the $\leq_{\mathsf{def}}$ relation, we now relate states by addressable *bytes* using a corresponding $\leq_{\mathsf{def,bytes}}$ relation.

We discovered another semantic bug while developing this solution: Encoding values as bytes produces a list of symbolic 'value fragments' of the form Fragment v n, each representing the n-th byte of the encoded value v. Decoding such a list of bytes to a value should ensure that all the fragments are present in the correct order. However, the check as implemented was to weak, and the function also accepted non-empty suffixes of valid lists. In the extreme, this made it possible to decode a single byte to a full 64-bit value.

4 Conclusions

Modeling low-level language semantics is error-prone. Even in well-studied semantics definitions such as CompCert's, basic bugs can hide in plain sight for a long time. Going to the even lower level of bytes rather than values can be tedious, but it allows us to be more precise—and force us to be more correct!