Formalisation of MiniSail in the Isabelle Theorem Prover

Alasdair Armstrong Neel Krishnaswami Peter Sewell Mark Wassell

University of Cambridge {firstname.lastname}@cl.cam.ac.uk

Abstract: Sail is a language used to model instruction set architectures. It has an imperative syntax and a dependent type system. We formalise a core calculus of the language in the Isabelle theorem prover describing the language syntax, substitution, the type system and operational semantics. A number of classic theorems such as preservation and progress are then proved. The purpose of this formalisation is to ensure that the full language is built on sound foundations and to provide a platform for the generation of the implementation of a type checker and evaluator for the language.

1 Introduction

Sail [1, 2] is a language used to model instruction set architectures (ISAs) for CPUs such as ARM, IBM POWER, MIPS, CHERI, RISC-V, and x86. It is an imperative language similar to the vendor pseudocode languages; the semantics of instructions is expressed as imperative code that makes register and memory accesses. Academic ISA models are often for small fragments, but full ISAs typically contain hundreds or thousands of instructions, so Sail models will be large and complex. In order to tame this complexity, a light-weight dependent type system is used. The type system provides integer, boolean, bit vector, register, record and union types. Type level constraints can be specified that constrain the values for integer indexed types such as integers or bit vectors. Function constraints can be used to relate the return value of a functions to values of the function's parameters. To ensure tractablility, constraints are limited to those that are solvable by an external SMT solver, Z3.

It is important to ensure that the Sail language and type system is itself sound and so formalising the language is of benefit. MiniSail is a small subset of Sail intended to capture key aspects of the language making it amenable to formalisation. This paper describes the work in progress to formalise MiniSail in Isabelle.

2 Syntax, Wellformedness and Substitution

Figure 1 shows the grammar of MiniSail. We use let normal form so that complex expressions are unpacked into nested `let` statements. This exposes the types of the subexpressions of complex terms.

The nonterminal τ represents a constrained type (also known as a refinement or liquid type [3]), z ranges over values, b over base types and ϕ over constraints. For example, the type $\{z : int | 0 \leq z \wedge z \leq 32\}$ is the type of integers between 0 and 32 inclusive.

The language grammar is mapped directly into nominal datatypes in Isabelle with binding specifications for the `let` and `case` statements, function definition and liquid types. A context, Γ is an ordered list of (x, b, ϕ) tuples. Program variables (i.e. those introduced in let and function bindings) can be used in types. A set of inductive predicates

value, v	::=	$x \mid n \mid \mathbf{T} \mid \mathbf{F} \mid \mathbf{inl}\ v \mid \mathbf{inr}\ v$
expr, e	::=	$v \mid v + v \mid v \leq v \mid f\ v$
stmt, s	::=	$\mathbf{let}\ x = e\ \mathbf{in}\ s \mid$
		$\mathbf{if}\ v\ \mathbf{then}\ s\ \mathbf{else}\ s \mid$
		$\mathbf{case}\ v\ \mathbf{of}\ \mathbf{inl}\ x_1 \rightarrow s \mid \mathbf{inr}\ x_2 \rightarrow s \mid$
fundef, fd	::=	$\mathbf{fun}\ f\ (x : b[\phi]\ ) : \tau = s$
prog, p	::=	$fd_1\ ;\ ..\ ;\ fd_n\ ;\ s$
base, b	::=	$\mathbf{int} \mid \mathbf{bool} \mid b + b$
ϕ	::=	$\mathbf{T} \mid \mathbf{F} \mid e = e \mid e \leq e \mid \phi \wedge \phi \mid$
		$\phi \vee \phi \mid \neg\phi$
τ	::=	$\{z : b \mid \phi\ \}$

Figure 1: MiniSail Grammar

defines wellformedness with respect to a context ensuring that variables appear in a context before they can be used.

3 Validity, Subtyping and Typing

SMT solver logic is modelled using an inductive predicate where we define an inductive rule for each property that we expect the solver to have. For example, that $\Gamma \models \phi \Longrightarrow \phi$ and basic facts about $+$ and $\leq$ operators. The subtyping judgement, $\Gamma \vdash \tau_1 \leq \tau_2$, is key and allows a smaller type to be used where a larger type is expected. Subtyping holds when the base types match and the constraint of the smaller type implies the constaint of the larger; the latter being checked by the SMT solver logic.

The type system of MiniSail is defined using bidirectional typing: we have the type synthesis judgement $\Gamma \vdash e \Rightarrow \tau$ and the type checking judgement $\Gamma \vdash s \Leftarrow \tau$. We define a type checking nominal inductive predicate for statements and both synthesis and checking nominal inductive predicates for values and expressions. Sum types are an interesting case: the type of a sum value cannot be inferred unless there is a type annotation. For example, with $inl\ v$ we can infer the value of the left side of the sum type from v, but we have no information that will give us the right side of the sum type. So we need a type checking judgement for values and expressions and provide a place in the syntax where the programmer can include a type annotation. The usual solution is to include in the grammar

$$\frac{\Gamma \vdash v_1 \Rightarrow \{z_1 : int | \phi_1\} \quad \Gamma \vdash v_2 \Rightarrow \{z_2 : int | \phi_2\}}{\Gamma \vdash v_1 + v_2 \Rightarrow \{z_3 : int | z_3 = v_1 + v_2\}}$$

$$\frac{f : (z_1 : b[\phi_1]) : \tau \quad \Gamma \vdash v \Rightarrow \{z_2 : b | \phi_2\} \quad \Gamma \models \phi_2[z_1 ::= v] \Longrightarrow \phi_1[z_1 ::= v]}{\Gamma \vdash f\ v \Rightarrow \tau[z_1 ::= v]}$$

Figure 2: Typing Rules

general type annotations on values and expressions. We instead introduce annotations only where it is required - in `let` and `case` statements. We need an hereditary substitution operation for the operational semantics that picks up the type of a term and drops it into the type annotation of the `let` or `case` statement. A small sample of the typing rules is given in Figure 2.

4 Substitution Lemmas and Operational Semantics

With the type system in place, we are in the process of proving in Isabelle a set of lemmas relating the type of term and the type of that term with a value substituted in. A simplified example is: If $\Gamma \vdash v \Rightarrow \{z : b|\phi\}$ and $(x, b, \phi)\#\Gamma \vdash s \Leftarrow \tau$ then $\Gamma \vdash s[x ::= v] \Leftarrow \tau$.

Single step reduction is defined by an inductive predicate. Next, we will prove that if a statement has a type, then the result of reduction has the same type. This is proved using the substitution lemmas. We will also prove the progress lemma: a well typed statement is either a value or has a reduction step.

5 Experience

This work has guided Sail development leading to the simplification of the handling of constraints and removal of unification on numeric expressions in types.

Paper formalisations of languages have an underlying convention that terms are worked with up to alpha-equivalence and that, if necessary, renaming of bound variables can occur implicitly. Mechanical formalisations in a theorem prover need to make this convention explicit to the prover. Nominal Isabelle provides the framework for making this easier than encoding the convention explicitly. This makes a nominal formalisation closer to a paper formalisation than a conventional mechanical formalisation however some supporting lemmas are needed and, when defining functions that operate over nominal datatypes, a number of proofs need to be provided. These proofs are sometimes tricky to prove and result in a lot of proof code that looks to to be the same modulo the structure of the binding. However with an intuitive understanding of how Nominal Isabelle works, it is relatively easy to see the approach needed and how to prove lemmas. Looking at prior work and borrowing lemmas has been helpful and sledgehammer as usual is a great assistant. As equality between nominal terms is alpha-equivalence, care is needed when unpacking a term with a binder. For example, if we have $\{z : b|\phi\} = \{z' : b'|\phi'\}$ then it doesn't hold that $\phi = \phi'$ but it is true that swapping any fresh variable with z in ϕ and z' in ϕ' does give equality.

6 Questions and Further Work

Alongside this work, we have developed AST datatypes and inductive rules for typing and reduction that do not use the Nominal package and do not include proofs. From this, the code generation facilities of Isabelle have been used to build an implementation of this language for a larger subset of Sail. The question arises as to whether it is possible to generate code from nominal datatypes, functions and inductive relations. If this is not possible, then one approach is to hand craft an implementation as 'vanilla' functions in Isabelle and then prove this implementation matches the nominal-formalisation.

Ott [4] provides a unified way of specifying the syntax and semantics of a language. A specification can then be exported to LaTex, Ocaml, Coq and Isabelle. With Isabelle, the output is 'vanilla' datatypes for the AST, functions for substitition and free-variables, and inductive predicates for the rules. An extension to Ott would be for the export to target Nominal Isabelle where the datatypes would be nominal ones and the substitition functions defined in the nominal style. Furthermore, supporting lemmas could be automatically generated and this will also reduce the boilerplate.

Acknowledgements This work was partially supported by EPSRC grant EP/K008528/1 (REMS).

References

[1] Sail. `http://www.cl.cam.ac.uk/~pes20/sail/`.

[2] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 635–646, New York, NY, USA, 2015. ACM.

[3] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008.

[4] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 1–12, New York, NY, USA, 2007. ACM.