Mechanised Metatheory for the Sail ISA Specification Language

Mark Wassell, Alasdair Armstrong, Neel Krishnaswami, Peter Sewell
University of Cambridge, UK
{mpew2,aa2019,nk480,pes20}@cl.cam.ac.uk

Abstract

Sail is a language for rigorous specification of instruction set architectures (ISAs); it has been used to model various production and research architectures, including ARMv8-A, RISC-V, and CHERI-MIPS, sufficiently completely to boot multiple operating systems. Intended to be engineer friendly, Sail is an imperative first-order language with a light-weight dependent type system; it can generate OCaml and C emulators and Isabelle, HOL4, and Coq definitions. A recent substantial redesign of the Sail type system has been done in conjunction with the development of a core calculus, MiniSail, along with paper proofs of type safety.

This paper presents further work to mechanise MiniSail in the Isabelle theorem prover, making use of the Nominal2 Isabelle library to address binding structures and alpha-equivalence; it includes the definition of the MiniSail type system and operational semantics and proofs of preservation and progress. We discuss how the mechanisation and paper formalisations relate to each other, including the benefits and pitfalls of each, and comment on how these have influenced and been influenced by the Sail implementation redesign. We also comment on whether the use of Nominal Isabelle allows us to write proofs of programming language safety that are human readable as well as being machine verified. This mechanisation should in future provide a platform for the mechanical generation of a verified implementation of a type checker and evaluator for the language.

Keywords Language Verification, Isabelle/HOL, CPU Specification

1 Introduction

CPU architectures such as ARM, IBM POWER, MIPS and x86 are a key part of our computing infrastructure and so it is vital that they are engineered without defects or vulnerabilities. Commonly, an instruction set architecture (ISA) is specified using a mix of English prose and imperative pseudo-code. These specifications are complex and difficult to reason with and so a more rigorous approach is warranted. Sail [2] has been developed as a language for modelling instruction set architectures. It supports exporting the model to theorem provers, for architecture property reasoning, and to C, to provide an executable of the model. The latter is comprehensive enough to run the model so that it boots a variety of operating systems. We introduce the Sail language in Section 2.

Sail has a novel light-weight dependent type system and uses the Z3 [6] SMT solver to assist with type checking. This type checker has had to be carefully engineered to ensure the decidability of type checking. To avoid the reliance on this engineering, our prior work work [2] introduced MiniSail, a kernel language for Sail and presented a paper formalisation of MiniSail that provided a number of guarantees about the Sail type system including decidability.

Decidability of type checking arises from the following: First, the typing rules are syntax directed. This ensures that there is a deterministic choice when deciding which rule to use to type check a particular term. Second, we only send to the SMT solver well-sorted problems in the quantifier-free linear arithmetic logic fragment. The problems are guaranteed to be in this logic fragment through a combination of limits on the syntax (e.g. we don't have quantifiers in the constraint syntax) and sort checking.

We review the choice of syntax, the type system and the operational semantics of MiniSail in Section 3.

This paper builds on the paper formalisation and describes a mechanisation of MiniSail in the Isabelle theorem prover [12]. This mechanisation provides a stronger guarantee than the paper proofs and gives a platform for the building of a validated type checker and executable for MiniSail making use of Isabelle's code generation features. The mechanisation has also uncovered a shortcoming in the paper proofs around the area of sort checking (that we detail later).

We make use of Isar, the structured proof language of Isabelle, which allows us to write proofs that resemble how paper proofs look. To address binding, α-equivalence and capture avoiding substitution, we make use of Nominal logic as provided by the Nominal2 Isabelle [4] package. The grammar of MiniSail is specified using Isabelle datatypes and the typing and reduction semantics by inductive predicates. We prove the type safety property for MiniSail in the usual way using progress and preservation lemmas. Progress tells us that a statement is either a value or can be reduced to another statement, and preservation tells us that types are preserved across reduction. The proofs of preservation make extensive use of substitution lemmas, which form the core of the mechanisation. We present the architecture of the Isabelle mechanisation in Section 4

PL'18, January 01–03, 2018, New York, NY, USA
2018.

The contributions of this work are as follows:

- We provide further guarantees of the soundness of the Sail ISA modelling language by mechanising it in a theorem prover.
- We show how a language with a light-weight dependent type system, that uses an SMT solver as part of typing checking, can be formalised using Nominal Isabelle.
- We argue that, with some up front cost, the Isar structured proof language and Nominal Isabelle allow us to write mechanised proofs that resemble proofs found in paper formalisation and are understandable by those without expert knowledge of theorem provers.
- We show the benefits of the three way co-development of this mechanisation with the development of the paper proofs and the full Sail implementation presented in [2].

2 Sail

We give a flavour of the Sail language. The Sail specification of an instruction set describes how bit strings map to instructions in the architecture and how these instructions are executed. The execution of an instruction manipulates the internal state of the CPU (such as the registers) and interacts with the memory accessible from the CPU.

As expected, a large part of ISA specifications involve the manipulation of bit vectors. For example, appending two bit vectors, extending a bit vector with zeros or pattern matching on bit vectors to decode an instruction. The instruction sets modelled go far beyond small research architectures and include commercial CPUs architectures that describe hundreds or thousands of instructions. Keeping track of vector lengths across all of the instructions is something that would assist the specification developer. To support this, Sail provides a light-weight dependent types system which allows the specification of constraints on the lengths of bit vectors and the range of integers.

An example of a type that can be specified is `range(0,32)` which is the type of integers that fall between 0 and 32. These constraints extend to function signatures with type variables that are universally quantified. For example, the type of a function that appends two bit vectors is:

```
val append = forall ('n:Int)('m:Int).
   (bits('n),bits('m))->bits('n+'m)
```

The `'n` and `'m` are integer type variables that specify the length of the bit vectors and are used to constrain the length of the bit vector returned by the function. Sail also makes use of these refinement types in its definition of arithmetic operators such as addition:

```
val add_atom = forall 'n 'm.
   (atom('n),atom('m)) -> atom('n+'m)
```

The type `atom('n)` is the type of integers that are equal to `'n`, and this is how program level values are lifted into type level values and fed into the result type of this function. Sail also provides record types (labelled product types) and union types (algebraic datatypes). The former is used to model the internal states of the CPU and the latter is used to model the AST of the instruction set.

Figure 1 shows an example Sail function. The function `double_bits` takes a bit vector of length `'m` and produces a bit vector of double the length with the bit pattern of the input duplicated. The signature of `double_bits` illustrates the dependent type nature of Sail: the length of the resulting vector is dependent on the length of the supplied vector. Furthermore, the signature includes the constraint that the length is at least 1. In the body of the function, the *mutable* variable `ys` is used to build the result vector. It is initialised to a vector of zeros with length equal to the result vector length. Then in the **`foreach`** loop, `ys` is first shifted left by the length of `xs` and then these new bits are set to the bits from `xs`. The final line of the function returns `ys`. Type checking must check at each assignment that the value being assigned to `ys` is a subtype of the original type of `ys` that was set when it is first assigned. In addition, the type of the return value, `ys`, is checked against the return type of the function.

```
$include <prelude.sail>

infix 7 <<
val operator << = "shift_bits_left" : forall 'n 'm.
   (bits('n), atom('m)) -> bits('n)
val or_vec : forall 'n.
   (bits('n), bits('n)) -> bits('n)
overload operator | = {or_bool, or_vec}

val length : forall 'n. bits('n) -> atom('n)

val zero_extend : forall 'n 'm, 'm >= 'n.
   (bits('n), atom('m)) -> bits('m)

val double_bits : forall 'm, 'm >= 1.
   (bits('m)) -> bits('m + 'm)

function double_bits(xs) = {
   let m = length(xs)
   ys : bits(m+m) = zeros(m+m);
   foreach (i from 1 to 2) {
     ys = ys << length(xs);
     ys = ys | zero_extend(xs, length(ys))
   };
   ys
}
```

Figure 1. Example Sail Function

We can not adopt a full dependent type system, such as that found in Agda[13], as we want the language to be understandable by the hardware engineers who are used to classical imperative languages. In addition, we want the type checking to be decidable and this motivates a light weight

dependent type system based on liquid types [17] and using an SMT solver to resolve constraints. Even so, liquid types are subtle and are easy to get wrong [21]. This motivates a formalisation of the language and proofs that provide soundness guarantees.

The Sail language includes the usual constructs for an imperative language: if statements, mutable variables, assignment and while loops. Also included is pattern matching in match statements and function arguments. Pattern matching supports complex matching on bit patterns and this is very useful when decoding bit patterns into instructions.

As well as making it engineer friendly, the imperative nature of language makes it possible to export to to C and OCaml to provide efficient execution of the model. In the case of the C export, this performs sufficiently well that an OS can be booted on this 'virtual CPU'. To support the formal validation of the model, the specification can also be exported as theorem prover code for the Isabelle, HOL and Coq theorem provers. Instruction set models have been developed for commercial CPUs such as ARM [1] as well as research architectures such as CHERI [22, 23].

A further motivation for a formal specification of Sail is the following: Instruction set architectures for commercial CPUs are long lived and evolve over decades and so if Sail is to be used to write these architectural models, Sail should itself be stable over this time. This stability could be achieved by freezing the language and its implementation, but this would mean that we can not take advantage of improvements to SMT solvers or the implementation language. A clear specification for Sail, that describes the language and type system would mitigate the risks associated with changes to the implementation technology.

To support the re-implementation of Sail, we decided that rather than attempt a specification of the full language, we would formalise a core-calculus of the language that contains the key interesting features of Sail, such as dependent-types and imperative constructs. This led us to MiniSail, which is described in the next section.

3 MiniSail

3.1 Kernel Language Design

Before describing the syntax and semantics of MiniSail, we motivate our choice of Sail features that have been included in MiniSail. We have shown above that Sail has a dependent type system, makes use of the Z3 SMT solver for type checking and has some imperative language features such as mutable variables.

We want the problems that we give to Z3 to be within a decidable logic fragment for Z3. We choose as this fragment quantifier-free linear arithmetic with algebraic datatypes which is decidable in Z3. The problems we pass to the solver are generated from the constraints within the types that pass through the type checker. Types arise from typing annotations in the MiniSail program or are synthesised by the type checker. We choose as values and expressions a representative sample from Sail that will give us a variety of typing rules that generate types.

Constraints in types can contain immutable variables but not mutable variables, so we need to ensure that synthesised types will not contain mutable variables. Also, we need to illustrate that type checking for mutable variable, binding and usage is sound as well; so we include some of the imperative features of Sail into MiniSail.

Liquid typing [17] that is used in Sail provides flow typing so we choose to include **if** and **match** statements in MiniSail language and their associated flow typing rules.

3.2 Syntax

MiniSail is intended to be a calculus for Sail rather than a programming language and has been designed for mathematical rather than programming convenience. To achieve this, the syntax of MiniSail is based on let-normal form and a stratification of what are expressions in the Sail language into values, expressions and statements in MiniSail. This leads to a corresponding stratification of the typing judgements and makes reasoning about the type system clearer. The grammar for values, expressions and statements is shown in Figure 2 along with other constructs in the language.

In the grammar we make use of the following meta-symbols:

- n - Numeric literals.
- x - Immutable program variables.
- u - Mutable program variables.
- z - Variables bound in refinement types.
- f - Function names.
- tid - Type identifiers for datatypes.
- C - Datatype constructors.

We can motivate the assignment of a strata to a term as follows: Values include the things that we want the reduction of a program to produce and the things stored in mutable variables. As values we have number, Boolean and bit vector literals, variables, pairing of values, construction of a datatype instance and the unit value.

Expressions are how we obtain new values from old and are: function application, application of the binary operations + and ≤, projection from a pair, mutable variables, concatenation of bit vectors and calculation of the length of a bit vector. Expressions can be formed from these operations applied to values only and so a complex expression, such as a function application to the sum of two variables, needs to be written as a nesting of let-statements where the parts of the complex expression are bound to intermediate variables. This ensures that the types of the components of complex expressions are made explicit since the typing of the nested let-statements will give us the type of each sub-expression.

We have two sorts of binding statements for variables: For immutable variables, we have the conventional let statement, **let** $x = e$ **in** s, that binds e to x in s, and for mutable variables we have **var** $u \ : \tau = v$ **in** s. Operationally, this allocates a slot in the mutable variable store to u, and puts v into the slot. The value can be accessed using mutable variable expressions in the statement s and subsequent assignments to u in s will update the value stored for u.

In the previous section we introduced Sail types and how they include type level variables and constraints on these variables. For MiniSail, we use a different style of syntax for types that is more in keeping with the Liquid type paradigm. The form for a MiniSail type is $\{z : b \mid \phi\ \}$ where z is the value variable, b is the base type and ϕ the refinement constraint. The base type is the universe from which values of the type will come from and the refinement constraint is a logical predicate that can reference the value variable and immutable program variables that are within scope. For example, $\{z : \textbf{int} \mid 0 \leq z\}$ is the type for all integers greater than or equal to zero. The refinement constraint can include expressions. However, to ensure that the constraints fall within a logical fragment that is decidable in Z3, we exclude from the expression syntax for constraints function application and mutable variable access; types can only depend on immutable variables This restricted form for expressions is indicated by e^- in the grammar.

Within the language are constructs for declaring the type of a function and for binding a function. The argument variable for the function is given explicitly in the signature and it can appear in the refinement constraint for the argument and the type for the function's return value. We also have the means to define new datatypes by associating a type identifier to a list of pairs $C : \tau$ where C is a constructor and τ is the type that the constructor needs to be applied to give an instance of the datatype.

3.3 Typing Judgements and Contexts

The MiniSail type system is described using a set of bidirectional [7] typing judgements. For values and expressions, the principal judgement is type synthesis, which has the form $\_ \vdash \_ \Rightarrow \tau$. For statements, the judgement is type checking, which has the form $\_ \vdash \_ \Leftarrow \tau$. The left placeholder stands for one or more contexts and the right placeholder by the term. Where we do need to check the type of an expression or a value, for example in function application, we perform type synthesis and then a check that the synthesised type is a subtype of the required type. The key judgements and contexts are listed in Figure 3.

Typing contexts are the structures that provide information about variables and other names that might appear in the terms that we are checking, and, importantly, that might appear in types. Different sorts of information comes into scope, or moves out of scope, at different times during the type checking of a program and so we divide this information into four structures:

- For type definitions, Θ. This is a map from type identifiers to datatype definitions. This context remains static during the evaluation of a program and so we will not need lemmas relating to changes to the context such as weakening. The well-scoping conditions for datatypes ensures that the definitions have no free variables and so we don't need to define a substitution function for this context.
- For function definitions, Φ. This is a map from function names to function definitions. This shares the same properties as the Θ context given above. Function bodies can reference only the immutable variable provided by the input argument, functions anywhere in Φ and datatype identifiers and constructors provided by a Θ context.
- For immutable variables, Γ. This is an ordered list of triples of the form $x : b[\phi]$ where x is an immutable variable, b is a base type and ϕ a constraint. The constraint can reference x and any other immutable variable given prior in the context as well as datatype identifiers and constructors provided by a Θ context. We define a substitution function for this context and the application of substitution to a Γ can result in the 'collapse' of the context.
- For mutable variables, Δ. This is a map from a mutable variable to a type. The type does not have to closed and can reference immutable variables in the Γ context and constructors from Θ

3.4 Sort Checking

Types are specified in program text or built up using the type synthesis rules. At subtype checks, the two types being checked are used to build an SMT problem that is then passed to the solver. We need to provide a guarantee that the constraints within these types lead us to a problem statement that falls within a decidable fragment of the SMT logic. This fragment is quantifier-free linear arithmetic with algebraic datatypes, it is multi-sorted and is decidable. As we will show later in an example, immutable variables appearing in the context and in types are mapped to variables in the SMT problem and assigned a type in the SMT space. We need to ensure that variables are assigned valid types and that the problem statement we construct sort checks.

The sort checking judgement for expressions is $\Phi; \Theta; \Gamma; \Delta \vdash_b e : b$. This says that e has base type (sort) b in the contexts on the left hand side. A small example of the sort checking rules are shown in Figure 4. When expressions are compared in a constraint, we require that they both have the same sort and, importantly, that they sort check in the context of empty Φ and Δ contexts. This ensures that function application and mutable variables will not appear in the expressions.

$$
\begin{array}{lrcl}
\text{Value} & v & ::= & x \mid n \mid \mathbf{T} \mid \mathbf{F} \mid (v, v) \mid \mathrm{C}\ v \mid () \\
\text{Expression} & e & ::= & v \mid v + v \mid v \leq v \mid f\ v \mid u \mid \mathbf{fst}\ v \mid \mathbf{snd}\ v \mid \mathbf{len}\ v \mid \mathbf{concat}\ v\ v \\
\text{Statement} & s & ::= & v \mid \mathbf{let}\ x = e\ \mathbf{in}\ s \mid \mathbf{let}\ x : \tau = s\ \mathbf{in}\ s \\
 & & & \mid \mathbf{if}\ v\ \mathbf{then}\ s\ \mathbf{else}\ s \mid \\
 & & & \mid \mathbf{match}\ v\ \{\ \mathrm{C}_1\ x_1 \Rightarrow s_1\ ,\ ..\ ,\ \mathrm{C}_n\ x_n \Rightarrow s_n\} \\
 & & & \mid \mathbf{var}\ u\ : \tau = v\ \mathbf{in}\ s \mid u := v \\
 & & & \mid \mathbf{while}\ (s_1)\ \ \mathbf{do}\ \{s_2\} \\
\text{Base type} & b & ::= & \mathbf{int} \mid \mathbf{bool} \mid \mathbf{unit} \mid \mathbf{bitvec} \mid \mathrm{tid} \mid \mathrm{b} \times \mathrm{b} \\
\text{Constraint} & \phi & ::= & \mathbf{T} \mid \mathbf{F} \mid e^- = e^- \mid e^- \leq e^- \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \neg\phi \\
\text{Refinement Type} & \tau & ::= & \{z : b \mid \phi\ \} \\
\text{Function Definition} & \mathrm{fd} & ::= & \mathbf{val}\ \ f : (x : b[\phi]) \rightarrow \tau \\
 & & & \mathbf{function}\ f(x) = s \\
\text{Datatype Definition} & \mathrm{td} & ::= & \mathbf{union}\ \mathrm{tid} = \{\mathrm{C}_1 : \tau_1, ..., \mathrm{C}_n : \tau_n\} \\
\text{Definition} & \mathrm{def} & ::= & \mathrm{td} \mid \mathrm{fd} \\
\text{Program} & \mathrm{p} & ::= & \mathrm{def}_1\ ;\ ..\ ;\ \mathrm{def}_n\ ;\ \mathrm{s}
\end{array}
$$

Figure 2. MiniSail Grammar

Φ	Function definition context	Θ	Type definition context
Γ	Immutable variable context	Δ	Mutable variable context
$\vdash_b \Theta$	Sort checking Θ	$\Theta; \Gamma \vdash_b \Delta$	Sort checking Δ
$\Theta \vdash_b \Phi$	Sort checking Φ	$\Theta; \vdash_b \Gamma$	Sort checking Γ
$\Theta; \Gamma \vdash_b v : b$	Sort checking values	$\Phi; \Theta; \Gamma; \Delta \vdash_b e : b$	Sort checking expressions
$\Theta; \Gamma \vdash_b \phi$	Sort checking constraints	$\Theta; \Gamma \vdash_b \tau$	Sort checking types
$\Theta; \Gamma \vdash v \Rightarrow \tau$	Type synthesis values	$\Theta; \Gamma \vdash v \Leftarrow \tau$	Type checking values
$\Phi; \Delta; \Gamma; \Theta \vdash e \Rightarrow \tau$	Type synthesis expressions	$\Phi; \Delta; \Gamma; \Theta \vdash s \Leftarrow \tau$	Type checking statements
$\Theta; \Gamma \vdash \tau_1 \lesssim \tau_2$	Subtyping	$\Theta; \Gamma \models \phi$	Validity

Figure 3. MiniSail Contexts and Judgements

Sort checking will also ensure that, for example, the two arguments to an addition operation have the **int** sort.

Sort checking rules were added late in the development of the mechanisation and replace and subsume the well-scoping rules that are used in the paper formalisation. The well-scoping rules were only checking that, for example, variables appearing in a term are present in the Γ context and were not checking that the base type of the variable was compatible with the term it was appearing in.

3.5 Subtyping and Validity

The subtyping relation is $\Theta; \Gamma \vdash \tau_1 \lesssim \tau_2$ and this breaks down to a validity relation $\Theta; \Gamma \models \phi_1 \longrightarrow \phi_2$ where ϕ_1 and ϕ_2 are the refinement predicates for the types τ_1 and τ_2. The SMT solver is used to check the validity relation.

We will illustrate how subtyping leads to an SMT problem with an extended example. Suppose we have a function with the signature:

$$\mathbf{val}\ f : (x : \mathbf{int}\ \times \mathbf{int}\ [0 \leq \mathbf{fst}\ x \wedge 0 \leq\ \mathbf{snd}\ x]) \rightarrow \{z :\ \mathbf{int} \mid \mathbf{fst}\ x \leq z \wedge\ \mathbf{snd}\ x \leq z\}$$

and that we need to prove the following typing judgement for the application of this function to the pair of variables (a, b)

$$\Phi; \Delta; a : \mathbf{int}\ [a = 9], b : \mathbf{int}\ [b = 10]; \Theta \vdash f\ (a, b) \Rightarrow \tau$$

where τ is $\{z : \mathbf{int} \mid a \leq z \wedge b \leq z\}$.

The derivation of the judgement will include this subtyping check:

$$\Theta; a : \mathbf{int}\ [a = 9], b : \mathbf{int}\ [b = 10] \vdash \{z : \mathbf{int}\ \times \mathbf{int}\ | z = (a, b)\ \} \lesssim \{x : \mathbf{int}\ \times \mathbf{int}\ | 0 \leq \mathbf{fst}\ x \wedge 0 \leq\ \mathbf{snd}\ x\}$$

that gives this validity checking judgement:

$$\Theta; a : \mathbf{int}\ [a = 9], b : \mathbf{int}\ [b = 10], z : \mathbf{int}\ \times \mathbf{int}\ [z = (a, b)] \models 0 \leq \mathbf{fst}\ z \wedge 0 \leq\ \mathbf{snd}\ z$$

which is equivalent to checking the validity of this logical expression:

$$a = 9 \wedge b = 10 \wedge z = (a, b) \longrightarrow 0 \leq\ \mathbf{fst}\ z \wedge 0 \leq\ \mathbf{snd}\ z$$

To check this, we generate the following Z3 script where we ask Z3 to check the satisfiability of the negated goal.

```
(declare-datatypes (T1 T2)
         ((Pair (mk-pair (fst T1) (snd T2)))))

(declare-const a Int)
(declare-const b Int)
(declare-const z (Pair Int Int))

(define-fun constraint () Bool (and (= a 9) (= b 10)
(= z (mk-pair a b)) (not (and ( <= 0 (fst z ))
                              ( <= 0 (snd z))))))
(assert constraint)
(check-sat)
```

The first line is a datatype declaration for a pair type. If our context had included variables for a datatype, then a declaration for this type would appear here. The next three lines declare the variables used; one declaration for each variable in the context. The Z3 type for the variable is derived from the MiniSail base type for the variable as per the context. The final three lines define the goal and request Z3 to check that the goal is satisfiable or not. If the goal is unsatisfiable, then it means that the subtype check has succeeded.

3.6 Typing Rules

A MiniSail program is composed of a set of datatype definitions, a set of function definitions and a single statement. To check a program, we will sort check the datatype and the function definitions. We will also check that the functions type-check meaning that a function body has the type indicated by the function's return type in a context containing just the functions argument and type. Typing checking of a statement cascades down the statement's structure and at points where checking of expressions or variables occurs, it flips over to type synthesis with subtyping checks occurring at the change over points. The important typing rules are show in Figure 5.

For all of the rules, if a context appears in the rule conclusion but not in a typing premise, then we need to include a sort checking judgement as a premise. For example, in rule 8 we include sort checking for Δ as we are introducing the Δ context. Sort checking of Δ will sort check the types in Δ, ensure that all free variables in the types are in the context Γ and that all data constructors are in Θ.

For the type synthesis rules for values, we only require the Θ and Γ contexts and the form for the type synthesised for a value v is always $\{z : b | z = v\}$. We are able to include v in the constraint as the rules check that v is well-sorted: for the base cases, literals are automatically well-sorted and variables are well sorted if they are in a well-sorted Γ context.

As mutable variables and function applications are included as expressions, we include the contexts Δ and Φ. The type synthesised can not have the form $\{z : b | z = e\}$, as not all expressions in our language are valid constraint-expressions. For mutable variables, the type synthesised is the type of the variable as recorded in the Δ context and for function application expressions, the type synthesised is the return type of the function with substitution of the argument value into the return type.

In a number of the rules, for example function application (12), mutable variable assignment (16) and while loops (20), we are required to check that the variable or expression has a particular type so we have a single type checking rule for values (rule 23) and for expressions (rule 24).

For statements, we have just a checking judgement. The statement **var** $u \ : \tau = v$ **in** s, that introduces and scopes mutable variables, has a typing rule where we check v is a subtype of τ and that s has the required type with $u : \tau$ added to the mutable variable context Δ. For assignment statements $u := v$, we require the type of these to be **unit** and check that the type of v is a subtype of τ where $u : \tau$ is an entry in Δ.

Typing for the **if** and the **match** statement incorporates flow typing. This is where the predicate for the type being checked for in the branches is checked under the assumption that the discriminator has the value associated to the branch. For example, in the positive branch of the **if** statement (rule 19), we type check against a type that has the constraint $v = \mathbf{T} \wedge (\phi_1[v/x]) \implies (\phi[z_1/z])$. For the match statement (rule 21), we use a slightly different form. Since we are binding a new variable x_i in each branch, we add this variable to the context with the constraint $\phi_i[x_i/z_i] \wedge v = C_i\ x_i$. While loops are straight-forward - the condition needs to be a Boolean and the body the unit. We don't attempt flow typing for while loops. One difficulty is that the discriminator for the while statement is a statement it is not obvious how to include a constraint on this in the type checking of the positive branch (while statement body) and the negative branch (the statements after the while loop).

3.7 Operational Semantics

We define the operational semantics using a small-step transition relation:

$$\Phi \vdash \langle \delta, s \rangle \longrightarrow \langle \delta', s' \rangle$$

Where δ is the mutable store that maps from mutable variable names to values and gives the current value of a mutable variable. For immutable variables, we rely on the substitution function to replace immutable variables with the value from the variable's binding statement.

We now outline how the transition relation works on different forms of statements. For **let** statements, **let** $x = v$ **in** s, we have a transition step case for each form for e. If e is a value, then the reduction is to $\langle \delta, s[v/x] \rangle$. For the case where e is the application of a binary operator, and the two arguments are integers, then we will perform the calculation and the resulting statement is s with the e replaced with

$$\frac{\begin{array}{l}\Theta;\Gamma \vdash_b \Delta \\ \Theta \vdash_b \Phi \\ \Theta;\Gamma \vdash_b v : b \\ \mathbf{val}\, f : (x : b[\phi]) \to \tau \in \Phi\end{array}}{\Phi;\Theta;\Gamma;\Delta \vdash_b f\ v : b(\tau)}\ \mathbf{2} \qquad \frac{\epsilon;\Theta;\Gamma;\epsilon \vdash_b e_1 : b \quad \epsilon;\Theta;\Gamma;\epsilon \vdash_b e_2 : b}{\Theta;\Gamma \vdash_b e_1 = e_2}\ \mathbf{1}$$

Figure 4. MiniSail Sort Checking

$$\frac{\Theta \vdash_b \Gamma}{\Theta;\Gamma \vdash n \Rightarrow \{z : \mathbf{int} | z = n\}}\ \mathbf{1} \qquad \frac{\Theta \vdash_b \Gamma}{\Theta;\Gamma \vdash \mathbf{T} \Rightarrow \{z : \mathbf{bool} | z = \mathbf{T}\}}\ \mathbf{2} \qquad \frac{\Theta \vdash_b \Gamma}{\Theta;\Gamma \vdash \mathbf{F} \Rightarrow \{z : \mathbf{bool} | z = \mathbf{F}\}}\ \mathbf{3}$$

$$\frac{\Theta \vdash_b \Gamma}{\Theta;\Gamma \vdash () \Rightarrow \{z : \mathbf{unit} | z = ()\}}\ \mathbf{4} \qquad \frac{\Theta \vdash_b \Gamma \quad x : b[\phi] \in \Gamma}{\Theta;\Gamma \vdash x \Rightarrow \{z : b | z = x\}}\ \mathbf{5} \qquad \frac{\Theta;\Gamma \vdash v_1 \Rightarrow \{z_1 : b_1 | \phi_1\} \quad \Theta;\Gamma \vdash v_2 \Rightarrow \{z_2 : b_2 | \phi_2\}}{\Theta;\Gamma \vdash (v_1, v_2) \Rightarrow \{z : b_1 * b_2 | z = (v_1, v_2)\}}\ \mathbf{6}$$

$$\frac{\begin{array}{l}\mathbf{union}\ tid = \{\ \overline{C_i : \tau_i}^{\,i}\ \} \in \Theta \\ \Theta;\Gamma \vdash v \Leftarrow \tau\end{array}}{\Theta;\Gamma \vdash C_j\ v \Rightarrow \{z : tid | z = C_j\ v\}}\ \mathbf{7} \qquad \frac{\begin{array}{l}\Theta \vdash_b \Phi \\ \Theta;\Gamma \vdash_b \Delta \\ \Theta;\Gamma \vdash v_1 \Rightarrow \{z_1 : \mathbf{int} | \phi_1\} \\ \Theta;\Gamma \vdash v_2 \Rightarrow \{z_2 : \mathbf{int} | \phi_2\}\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash v_1 + v_2 \Rightarrow \{z_3 : \mathbf{int} | z_3 = v_1 + v_2\}}\ \mathbf{8} \qquad \frac{\begin{array}{l}\Theta \vdash_b \Phi \\ \Theta;\Gamma \vdash_b \Delta \\ \mathbf{val}\, f : (x : b[\phi]) \to \tau \in \Phi \\ \Theta;\Gamma \vdash v \Leftarrow \{z : b | \phi\}\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash f\ v \Rightarrow \tau[v/x]}\ \mathbf{9}$$

$$\frac{\begin{array}{l}\Theta \vdash_b \Phi \\ \Theta;\Gamma \vdash_b \Delta \\ u : \tau \in \Delta\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash u \Rightarrow \tau}\ \mathbf{10} \qquad \frac{\begin{array}{l}\Theta \vdash_b \Phi \\ \Theta;\Gamma \vdash_b \Delta \\ \Theta;\Gamma \vdash v \Rightarrow \{z : b_1 * b_2 | \phi\}\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash \mathbf{fst}\ v \Rightarrow \{z : b_1 | z = \mathbf{fst}\ v\}}\ \mathbf{11} \qquad \frac{\begin{array}{l}\Theta \vdash_b \Phi \\ \Theta;\Gamma \vdash_b \Delta \\ \Theta;\Gamma \vdash v \Rightarrow \{z : b_1 * b_2 | \phi\}\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash \mathbf{snd}\ v \Rightarrow \{z : b_2 | z = \mathbf{snd}\ v\}}\ \mathbf{12}$$

$$\frac{\begin{array}{l}\Theta \vdash_b \Phi \\ \Theta;\Gamma \vdash_b \Delta \\ \Theta;\Gamma \vdash v \Leftarrow \tau\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash v \Leftarrow \tau}\ \mathbf{13} \qquad \frac{\Phi;\Delta;\Gamma;\Theta \vdash e \Rightarrow \{z : b | \phi\} \quad \Phi;\Delta;\Gamma, x : b[\phi[x/z]];\Theta \vdash s \Leftarrow \tau}{\Phi;\Delta;\Gamma;\Theta \vdash \mathbf{let}\ x = e\ \mathbf{in}\ s \Leftarrow \tau}\ \mathbf{14} \qquad \frac{\Phi;\Delta;\Gamma;\Theta \vdash e \Rightarrow \{z : b | \phi\} \quad \Phi;\Delta;\Gamma, x : b[\phi[x/z]];\Theta \vdash s \Leftarrow \tau}{\Phi;\Delta;\Gamma;\Theta \vdash \mathbf{let}\ x = e\ \mathbf{in}\ s \Leftarrow \tau}\ \mathbf{15}$$

$$\frac{\begin{array}{l}u \notin \mathrm{dom}(\delta) \\ \Theta;\Gamma \vdash v \Leftarrow \tau \\ \Phi;\Delta, u : \tau;\Gamma;\Theta \vdash s \Leftarrow \tau_2\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash \mathbf{var}\ u : \tau := v\ \mathbf{in}\ s \Leftarrow \tau_2}\ \mathbf{16} \qquad \frac{\begin{array}{l}\Theta \vdash_b \Phi \\ \Theta;\Gamma \vdash_b \Delta \\ u : \tau \in \Delta \\ \Theta;\Gamma \vdash v \Leftarrow \tau\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash u := v \Leftarrow \{z : \mathbf{unit} | \top\}}\ \mathbf{17} \qquad \frac{\Phi;\Delta;\Gamma;\Theta \vdash s_1 \Leftarrow \{z : \mathbf{unit} | \top\} \quad \Phi;\Delta;\Gamma;\Theta \vdash s_2 \Leftarrow \tau}{\Phi;\Delta;\Gamma;\Theta \vdash s_1; s_2 \Leftarrow \tau}\ \mathbf{18}$$

$$\frac{\begin{array}{l}\Theta;\Gamma \vdash v \Rightarrow \{x : \mathbf{bool} | \phi_1\} \\ \Phi;\Delta;\Gamma;\Theta \vdash s_1 \Leftarrow \{z_1 : b | (v = \mathbf{T} \wedge (\phi_1[v/x])) \implies (\phi[z_1/z])\} \\ \Phi;\Delta;\Gamma;\Theta \vdash s_2 \Leftarrow \{z_2 : b | (v = \mathbf{F} \wedge (\phi_1[v/x])) \implies (\phi[z_2/z])\}\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash \mathbf{if}\ v\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \Leftarrow \{z : b | \phi\}}\ \mathbf{19} \qquad \frac{\Phi;\Delta;\Gamma;\Theta \vdash s_1 \Leftarrow \{z : \mathbf{bool} | \top\} \quad \Phi;\Delta;\Gamma;\Theta \vdash s_2 \Leftarrow \{z : \mathbf{unit} | \top\}}{\Phi;\Delta;\Gamma;\Theta \vdash \mathbf{while}\ (s_1)\ \mathbf{do}\ \{s_2\} \Leftarrow \{z : \mathbf{unit} | \top\}}\ \mathbf{20}$$

$$\frac{\begin{array}{l}\mathbf{union}\ tid = \{\ \overline{C_i : \{z_i : b_i | \phi_i\}}^{\,i}\ \} \in \Theta \\ \Theta;\Gamma \vdash v \Rightarrow \{z : tid | \phi\} \\ \overline{\Phi;\Delta;\Gamma, x_i : b_i[\phi_i[x_i/z_i] \wedge v = C_i\ x_i \wedge (\phi[v/z])];\Theta \vdash s_i \Leftarrow \tau}^{\,i}\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash \mathbf{match}\ v\ \mathbf{of}\ \overline{C_i\ x_i \Rightarrow s_i}^{\,i} \Leftarrow \tau}\ \mathbf{21} \qquad \frac{\begin{array}{l}\Theta;\Gamma \vdash_b \{z_1 : b | \phi_1\} \\ \Theta;\Gamma \vdash_b \{z_2 : b | \phi_2\} \\ \mathbf{P};\Gamma, z_3 : b[\phi_1[z_3/z_1]] \models \phi_2[z_3/z_1]\end{array}}{\Theta;\Gamma \vdash \{z_1 : b | \phi_1\} \lesssim \{z_2 : b | \phi_2\}}\ \mathbf{22}$$

$$\frac{\begin{array}{l}\Theta;\Gamma \vdash v \Rightarrow \{z_2 : b | \phi_2\} \\ \Theta;\Gamma \vdash \{z_2 : b | \phi_2\} \lesssim \{z_1 : b | \phi_1\}\end{array}}{\Theta;\Gamma \vdash v \Leftarrow \{z_1 : b | \phi_1\}}\ \mathbf{23} \qquad \frac{\begin{array}{l}\Phi;\Delta;\Gamma;\Theta \vdash e \Rightarrow \{z_2 : b | \phi_2\} \\ \Theta;\Gamma \vdash \{z_2 : b | \phi_2\} \lesssim \{z_1 : b | \phi_1\}\end{array}}{\Phi;\Delta;\Gamma;\Theta \vdash e \Leftarrow \{z_1 : b | \phi_1\}}\ \mathbf{24}$$

Figure 5. MiniSail Type System

the result of the operator. For the case where e is $\mathbf{fst}\ (v_1, v_2)$ or $\mathbf{snd}\ (v_1, v_2)$, then the resulting statement is s with the e replaced by v_1 or v_2. For the case where e is a function application, $f\ v$, then the reduction is to $\mathbf{let}\ x : \tau[v/y] = s'[v/y]\ \mathbf{in}\ s$ where s' is the body of the function f, y is the function argument and τ is the return type of the function. We substitute the value the function is applied to into both the return type and the body of the function s'. This means

that, any types in s' (in **var** statements) will also undergo substitution. This step is conditional on f having an entry in the function context Φ.

For the statement **let** $x : \tau = s_1$ **in** s_2, we have two rules: if s_1 is a value v, then the reduction is to $(\delta, s_2[v/x])$ otherwise the reduction is to $(\delta', \textbf{let}\ x : \tau = s_1'\ \textbf{in}\ s_2)$ where (δ', s_1') is the reduction of (δ, s_1).

For the statement **if** v **then** s_1 **else** s_2, we have a case for when v is **T** and one for when v is **F**. For the statement **while** (s_1) **do** $\{s_2\}$, we unroll this to

$$\textbf{let}\ x : \{z : \textbf{bool} \mid \textbf{T}\} = s_1\ \textbf{in}$$
$$\textbf{if}\ x\ \textbf{then}\ s_2; \textbf{while}\ (s_1)\ \textbf{do}\ \{s_2\}\ \textbf{else}\ ()$$

For **var** $u : \tau = v$ **in** s, we extend the variable store with the pair (u, v) and the reduced statement is s. For assignment $u := v$, we update the value of u in the store with v. Note that at this stage, the value being assigned to the mutable variable will have no immutable variables.

For the sequence statement $s_1; s_2$, we have two rules: If s_1 is the unit value, then we reduce to s_2. If s_1 is not the unit value, then we reduce to $(\delta', s_1'; s_2)$ where (δ', s_1') is the result of the reduction of (δ, s_1).

For the match statement,

$$\textbf{match}\ v\ \{\ \mathrm{C}_1\ x_1 \Rightarrow s_1\ ,\ ..\ ,\ \mathrm{C}_n\ x_n \Rightarrow s_n\}$$

statements where v is of the form C v' and C matches one of the C_i, then the reduction is to $(\delta, s_i[v'/x_i])$.

4 Mechanisation in Isabelle

Isabelle has been used previously to formalise a number of programming languages. For example, Lightweight Java [19] and WebAssembly [24]. Languages have also been formalised using other provers. For example C in the Coq theorem prover [11], and CakeML using HOL4 [10].

In this section we describe how the grammar, type system, and operational semantics of MiniSail are mechanised in Isabelle.

4.1 General Approach

A key motivation for our mechanisation was our experience with proving soundness for MiniSail on paper. We began with a (paper) proof outline following the usual path for language soundness proofs but incorporating additional lemmas as needed by novel aspects of the type system. However we quickly realised that there was a lot of detail that needed to be tracked in the paper proof particularly around variables and the structure of the various inductive cases. The mechanisation was then begun with the intention that this would replace the paper proofs.

At the high-level, the main argument closely follows the structure of the paper formalisation. In addition to working on the main argument, there is also significant work providing the necessary scaffolding in the form of technical lemmas that don't appear in paper proofs. An example of one of these lemmas is the weakening lemma for the lookup function with a well-formed Γ-context. Spending time writing the technical lemmas is important for a number of reasons: First, we avoid having to re-prove similar facts in the proofs of the larger lemmas. Second, they act as a check of the definitions of data structures and functions; if we fail to prove a trivial technical lemma about a definition, or the proof is more complex than expect, then it can indicate a problem with the definition and that the definition needs fixing or optimising. Finally, these technical lemmas are available for use by Isabelle's automatic proof methods and sledgehammer.

At the proof-level, there was a lot of technical detail involved in each proof. Proving facts about freshness and basic manipulation of contexts occurred frequently. and the choices around supporting data structures and functions had to be made carefully to avoid making the proofs overly complex. For example, the Φ and Θ contexts given above were originally one context, Π. It was realised that when we added sort checking for expressions, we wanted a Π that had no functions but could contain datatype definitions. This required some messy filtering and associated lemmas and it was realised two separate contexts would be easier and make things clearer.

Many theorem provers follow a backward reasoning pattern - you start with the goal and the game is to pick a tactic that reduces the current goal to a new set of goals, and to iterate these steps until no goals are left. All that is captured in the text of the theory is the sequence of invocations of the tactics used - the proof script.

Proofs using Isar are written closer to how paper mathematics looks and will include not only the proof methods/-tactics used but also the goals and subgoals explicitly written out in a hierarchical way. This allowed us to use the high level outline of the proof from our paper formalisation (or a paper sketch if the paper formalism doesn't include the lemma we are working on) as the starting point for many proofs. From the outline, we then iteratively filled in the detail, and when we got to a goal that we thought was simple enough to be proved by Isabelle's automatic proof methods, or by sledgehammer, we invoked these to see if the goal could be proved.

Sledgehammer [14] is a tool in Isabelle, that is able to use external provers to provide a proof for goal. Isabelle will then reconstruct this proof making use of only proof methods available within Isabelle itself. The sledgehammer mechanism is able to choose a set of facts from those available in the current proof context to use in addition to those explicitly specified by the user. However sledgehammer didn't always come up with a proof but in these cases it was clear what lemmas needed to be used but not how to wire them up. Often the problem was that there was an error in the phrasing of the goal and a useful improvement to the automatic proof methods, and sledgehammer, would be for Isabelle to report

to the user the reasons, or possible reasons, why the method or sledgehammer failed.

The presentation style of Isar, whilst requiring each step and goal to be explicitly spelled out[1], does have some advantages when it comes to handling changes to proofs. For example, when we introduced sort checking to the mechanisation it was clear from the Isabelle user interface what proofs needed fixing. With a proof-script style of reasoning, this information is hidden inside the reasoning engine and only by attempting the reasoning steps would you see what was wrong from information shown in the user interface.

A major milestone was of course having everything proved. At this stage we are on stable platform and we can begin the process of incrementally tidying up and optimising the proofs. For the current version of the mechanisation, more optimisation can be done in conjunction with building up a better collection of technical lemmas.

4.2 Syntax

As with many formalisation of programming languages, substitution is an important operation. In the operational semantics we use it to drive the reduction of let and match statements as well as function application. Since MiniSail contains binding statements, we need to be careful about how substitution operates on these statements. The notation we use for substitution of a variable x for a value v in the statement s is $s[x ::= v]$. Where the statement is a binding statement then there are a number of preconditions associated with the substitution: In the substitution $(\textbf{let}\ w = e\ \textbf{in}\ s)[x ::= v]$ we want to ensure that no free variable in v is captured by the binding and if $x = w$ then we want the substitution to occur after renaming w in the statement to some other variable. In paper mathematics these requirements are handled by the convention that we are free to replace a term by one that is α-equivalent to it obtained by renaming bound variables with a new name. In mechanical formalisations there has to be something in the formalisation to handle this.

There are a variety of approaches to this including De Bruijn indices [5] and the locally nameless representation [3]. The approach that we adopted is to make use of Nominal logic ideas [16] as realised in the Isabelle Nominal2 package [8, 20]. The use of Nominal is not usually the first choice and partially justified in that we want to write the formal proofs in a way that matches the paper proofs. We also wanted to take this opportunity to see how Nominal-Isabelle works in a language formalisation.

The following is an outline of Nominal Isabelle as it pertains to this formalisation. The basic building block in Nominal Isabelle is the multi-sorted atom which correspond to variables. Atoms can be declared as part of the structure of terms and in MiniSail we have one sort of atoms for mutable variables and one for immutable variables. A basic operation on terms is the operation of permuting atoms. For example, in the syntax of Nominal-Isabelle, we have:

$$(z \leftrightarrow w) \bullet (\textbf{let}\ x = 1\ \textbf{in}\ z) = (\textbf{let}\ x = 1\ \textbf{in}\ w)$$

where $(z \leftrightarrow w)$ is the permutation that swaps the two atoms z and w and $\bullet$ is the permutation application operator. From this is defined the support for a term, which is equivalent in our case to the notion of free variables, and from this the definition of freshness, written as 'atom x # v', is defined as non-membership of the support of the term v.

Nominal-Isabelle uses these concepts to automatically generate, from provided binding information, definitions of α-equivalence for each sort of term. For example, the statements $\textbf{let}\ x_1 = e_1\ \textbf{in}\ s_1$ and $\textbf{let}\ x_2 = e_2\ \textbf{in}\ s_2$ are α-equivalent if $e_1 = e_2$ and for any c # (x_1, s_1, x_2, s_2) we have $(c \leftrightarrow x_1) \bullet s_1 = (c \leftrightarrow x_2) \bullet s_2$.

We represent the MiniSail grammar AST as a set of nominal datatypes. These are like conventional datatypes but allow us to specify the binding structure of terms and will also trigger Isabelle to generate the freshness, support and α-equivalence facts as described above. Isabelle will construct an internal representation for the datatype and quotient it using α-equivalence to give the surface datatype.

```
nominal_datatype case_s = — Statements
  C_final   dc x::x s::s binds x in s
| C_branch  dc x::x s::s case_s binds x in s
and s =
  AS_val "v"
| AS_let x::x  "e" s::s binds x in s
| AS_let2 x::x "τ"  "s" s::s binds x in s
| AS_if "v" "s" "s"
| AS_var u::u τ v s::s  binds u in s
| AS_assign u  v
| AS_case "v" case_s
| AS_while s s
| AS_seq s s
```

Figure 6. Statement Datatype

Figure 6 shows the nominal datatype definition for statements. The "binds" keyword is used to indicate the binding structure for the **let**, **let2** and **var** statements. The datatype for the branches of the case statement are also defined here and we enforce that there is at least one branch.

An important property of functions that is used in proofs is equivariance: permuting atoms on the result of a function is equal to applying the function to permuting atoms on the arguments. Informally, equivariance means that the function is well-behaved with respect to α-equivalence. Equivariance also extends to propositions and inductive predicates including typing judgements. When defining nominal functions,

[1] This effort can be reduced a little by using experimental tools such as 'explore' introduced on the Isabelle mailing list https://www.mail-archive.com/isabelle-dev@mailbroy.informatik.tu-muenchen.de/msg04443.html

Isabelle generates a proof obligation for equivariance that we need to prove. In most cases these are easy to prove.

The first set of functions that we define are the important substitution functions for the language terms. Nominal Isabelle allows us to encode the capture avoiding property substitution that we need by allowing the specification of a freshness condition for a branch of the function as shown in Figure 7. We prove a number of lemmas related

```
nominal_function subst_t :: "x ⇒ v ⇒ τ ⇒ τ" where
  "atom z ♯ (x,v) ⟹ subst_t x v ⦇ z : b | c ⦈ = ⦇ z : b | c[x::=v]c ⦈"
```

Figure 7. Substitution Function For Types

permutation of variables to substitution using the lemmas from [15] as a guide. An example is that substituting in a variable is the same as permutation provided y is fresh in t: $(x \leftrightarrow y) \bullet t = t[x := y]$. As program variables can appear in types, we define substitution for types. When we define substitution for statements, we include in the clause for the **var** statement substitution into the type annotation. The structures for contexts Φ, Π, Γ and Δ are defined using list, or list-like, datatypes. We define substitution for Δ, which is just a mapping of the substitution function over the types for the mutable variables. Substitution in Γ, $\Gamma[x/v]$ may result in the 'collapse' of Γ if x has an entry in Γ. Finally, for the syntax part, we define the inductive predicates capturing the well-scoping and sort checking rules.

4.3 SMT Model

As mentioned above, the subtyping check boils down to a validity check in a logic supported by the SMT solver Z3. In order to prove lemmas about subtyping, for example transitivity, substitution and weakening, we need to build a model of the SMT logic and setup definitions and supporting lemmas. The model also allows us to prove specific subtyping relations between types used by the progress lemma.

The logic is multi-sorted and we can reuse the base type datatype to define the sorts for variables and terms in this logic. We first define a datatype representing the values in this logic and define the type of valuation that is a map of variables to values. To ensure that the valuation maps variables to the value of the appropriate sort, we define a notion of well-formedness for valuations with respect to the contexts Θ and Γ. This well-formedness condition also ensures that all the variables in Γ are given a value by the valuation. We then define a series of evaluation relations for literal, values, expressions and constraints that define how these terms are evaluated by a valuation. Importantly we prove an existence lemma that says that there is a value for a term under a valuation if the term is well-sorted and the valuation is well-formed. We will then prove later that type synthesis for values and expressions produce well-formed constraints and so have shown that synthesised types fall within the decidable logic fragment.

Building on evaluation of constraints, we define satisfaction of a constraint in Θ and Γ contexts and then validity. We prove strengthening, weakening and substitution properties for validity.

4.4 Typing

In this subsection we describe how the typing rules presented in Figure 5 are translated into inductive predicates in Isabelle.

For the most part, the Isabelle rules are a transcription from the paper formalisation into Isabelle. The only difference is that we need to add freshness conditions to the premises of some of the rules. For example, for the type synthesis rules for values, we need to ensure that z appearing in a synthesised type does not also appear in v or the context Γ.

Equivariance for the predicate is proved automatically and induction rules are generated. For nominal predicates and datatypes we also have strong induction rules generated that we can use in nominal inductive proofs to ensure that freshness conditions are automatically built into the proof. We also instruct Isabelle to generate various forms of elimination rules that we can use in proofs.

For proving lemmas about the typing judgements, we can use induction over the syntax of the term sort that we are proving the lemma for, or induction over the structure of the derivation, making use of the induction rules generated from the typing predicate. The latter technique is preferable however for statements it initially took some care to construct the lemma correctly and to invoke the required proof method.

We prove lemmas that assure us that synthesised types are well-formed and are unique up to alpha-equivalence. Furthermore we prove weakening. This tells us that a judgement continues to hold if a context is replaced with a larger one - when considered as a set, the larger is a superset of the smaller and that the larger is also well-scoped. We do this for the Γ and Δ contexts.

4.5 Substitution Lemmas

The key enabling lemmas are the substitution lemmas, primarily the substitution lemma for statements which is shown in Figure 8. The substitution lemma ensures that reduction steps using substitution preserve the type over the reduction. Before being able to prove this lemma for statements, we need to prove it for expressions and values and, since substitution occurs into types as well, we need to prove lemmas involving substitution for types, subtyping and validity. Prior to this we need to prove a set of narrowing lemmas. These are lemmas telling us that if a variable x has an entry $x : b[\phi]$ in a context Γ and we replace ϕ with a more restrictive constraint in Γ, then type checking and synthesis

```
lemma subst_infer_check_s:
  fixes v::v and s::s and cs::case_s and x::x and c::c and b::b and Γ1::Γ and Γ2::Γ
  assumes "Θ ; Γ1 ⊢ v ⇒ τ"  and "Θ ; Γ1 ⊢ τ ≲ {| z : b | c |}" and  "atom z ♯ (x, v)"
  shows  "Φ; Δ; Γ; Θ  ⊢ s ⇐  τ'     ⟹ Γ = (Γ2@((x,b,c[z::=V_var x]c)#Γ1)) ⟹
                                        Φ; Δ[x::=v]Δ; Γ[x::=v]Γ; Θ ⊢ s[x::=v]s  ⇐ τ'[x::=v]τ" and
        "Φ; Δ; Γ; Θ; tid ⊢ cs ⇐ τ' ⟹ Γ = (Γ2@((x,b,c[z::=V_var x]c)#Γ1)) ⟹
                                        Φ; Δ[x::=v]Δ; Γ[x::=v]Γ; Θ; tid ⊢ cs[x::=v]s ⇐ τ'[x::=v]τ"
```

Figure 8. Substitution Lemma for Statements

judgements for statements, expressions and values will hold if they hold for the original Γ.

To prove the substitution lemma for statements we use rule induction and outline now how this proceeds for the let statement. The typing rule for the let statement is shown in Figure 9. Rule induction will invert the statement of the lemma and the premises for the rule will be added as premises for the case with the appropriate instantiation of variables. Our task is to apply the typing rule for the let statements in the forward direction to show substitution is true for the let statement. To get to the point where we can do this, there are some steps we need to do:

- Apply the substitution lemma for expressions to e to obtain a subtype of the type for e
- Use the induction hypothesis to get the substitution for s.
- Apply the narrowing lemma for the type of e in the context.
- Prove the required freshness conditions.

The first three steps are shared with a paper proof however the last is not.

A common proof pattern we encountered was that in order to satisfy a freshness condition, we needed to obtain a new variable that is fresh for a set of terms and contexts. Obtaining the new variable is a one-liner but the tricky step is proving facts that hold for terms containing the original variable continue to hold, in some way, for terms with the fresh variable. For example, in the weakening lemma for the **let** statement **let** $x = e$ **in** s, the typing rule for this requires that x does not occur in the context Γ. However with weakening there is no restriction on what might have been added to the expanded context, and this might have been x. Therefore we have to prove weakening for an α-equivalent let statement where the bound variable does not occur in the context.

4.6 Progress, Preservation and Safety

We finish the formalisation with proofs of progress, preservation and finally safety. The statement of these as Isabelle text is shown in Figure 10

For type preservation, we don't just prove preservation of the type of the statement under reduction but we also need to ensure that the types of the values in the possibly updated mutable store δ remain compatible with the Δ context. To this end, we introduce a new typing judgement: $\Phi;\Theta \vdash (\delta, s) \Leftarrow \tau;\Delta$. This judgement holds if we have $\Phi;\Theta;\epsilon;\Delta \vdash s \Leftarrow \tau$ and for every (u, v) in δ, if there is a pair $(u, \tau) \in \Delta$, then $\Theta;\epsilon \vdash v \Leftarrow \tau$.

We extend the preservation lemma to handle multi-step reductions (indicated by $\longrightarrow^*$). The progress lemma states that if a statement and store are well-typed, then the statement is either a value or a reduction step can be made. Finally, we prove the type safety lemma for MiniSail that says that if s is well-typed, and we can multi-step reduce s to s', then either s' is a value or we can make another step.

These final lemmas here are starting to look like the paper proofs as we have by this stage established a foundation with the substitution lemmas and have left the realm of syntax and typing.

5 Experience and Discussion

The development of the MiniSail has fed back into the development of the full Sail language. Early versions of Sail had an ad-hoc hand-written constraint solver, which was often incapable of proving all constraints found in our ISA specifications. By co-developing the MiniSail formalisation with re-development of the full Sail type checker, we were able to guarantee the decidability of all constraints, while simultaneously increasing the expressiveness of the constraint language.

Not only has the MiniSail formalisation significantly increased our confidence in the robustness of our type system design, the use of an off-the-shelf constraint solver (Z3), combined with the bi-directional type-checking approach used in MiniSail has increased the performance of the Sail language significantly: before starting our MiniSail formalisation, a fragment of ARMv8 specified in Sail would take several minutes to type-check due to pathological cases in the ad-hoc constraint solver. In recent versions, type-checking the entire 64-bit fragment of the ARMv8 instruction set can be done in under 3 seconds on a modern computer (Intel i7-8700).

We have both a paper formalisation of MiniSail, presented in [2] and a mechanised formalisation presented in this paper and so there is an opportunity to compare the two. The first point of comparison is in the lengths of the two works: The paper formalisation take 85 pages of typeset mathematics,

```
check_letI: "⟦
      atom x ♯ (z,c,τ,Γ); atom z ♯ Γ ;
      Φ ; Δ ; Γ ; Θ ⊢ e ⇒ ⦇ z : b | c ⦈ ;
      Φ ; Δ ; ((x,b,c[z::=V_var x]_c)#Γ) ; Θ ⊢ s ⇐ τ ⟧ ⟹
      Φ ; Δ ; Γ ; Θ ⊢ (AS_let x e s) ⇐ τ"
```

Figure 9. Typing rule for Let statement

```
lemma progress:
  fixes s::s
  assumes "Φ ; Θ ⊢ ⟨ δ , s ⟩ ⇐ τ ; Δ "
  shows "(∃v. s = AS_val v) ∨ (∃δ' s'. Φ ⊢ ⟨ δ , s ⟩ ⟶ ⟨ δ' , s' ⟩)"
lemma preservation:
  fixes s::s and s'::s and cs :: case_s
  assumes " Φ ⊢ ⟨ δ , s ⟩ ⟶ ⟨ δ' , s' ⟩" and "Φ ; Θ ⊢ ⟨ δ , s ⟩ ⇐ τ ; Δ"
 shows "∃Δ'. Φ ; Θ ⊢ ⟨ δ' , s' ⟩ ⇐ τ ; Δ' ∧ set Δ ⊆ set Δ'"
lemma safety:
  assumes " Φ ⊢ ⟨ δ , s ⟩ ⟶* ⟨ δ' , s' ⟩" and "Φ ; Θ ⊢ ⟨ δ , s ⟩ ⇐ τ ; Δ"
  shows "(∃v. s' = AS_val v) ∨ (∃δ'' s''. Φ ⊢ ⟨ δ' , s' ⟩ ⟶ ⟨ δ'' , s'' ⟩)"
```

Figure 10. Preservation, Progress and Safety Lemmas

with associated commentary, and the Isabelle mechanisation takes 402 pages of typeset Isabelle code. The Isabelle formalisation also took around 4 times as long to complete as the paper proofs. This included the initial mechanisation that was using the paper proofs as a starting point and the extra time to add the sort-checking that isn't in the paper proofs.

To decrease the effort better tools and practices are needed to remove some of the boilerplate proof code and the repetitive nature of the proofs. Ott [18] was used to typeset the typing rules and operational semantics for the paper proofs. Ott can be used to specifying binding structure and does export to Isabelle (but not in the Nominal style). So future work is to enhance Ott and to see if can be used to tame the boilerplate similar to the work described in [9].

A feature of paper proofs is that the author has greater control over the structure of the proof and can ensure that the main thread of the argument is made clear and that ancillary detail is elided. With a mechanised proof, the detail has to be present somewhere. The challenge in this mechanisation has been to ensure that the detail, particularly the detail about freshness, which wouldn't appear in the paper proof, doesn't swamp or hide the main argument. Isabelle provides a means to structure proofs and this has been taken advantage of in most places in the mechanisation. However some more work is required to clean up some of the proofs possibly with the help of suitable technical lemmas that will enable Isabelle automatic proof methods.

Another point of comparison between paper and mechanised proofs is around the level of abstraction. In the paper proofs, we deal directly with the language syntax, whereas in the mechanisation we have to go through datatypes that represent the syntax. When constructing the datatypes, we can choose to have a datatype constructor corresponds to a single production rule in the syntax or we can abstract and have the datatype represent a number of similar rules and have some parameterisation mechanism. For example, in the Isabelle mechanisation for the binary operators + and ≤, we have a single constructor for the binary operator and parameterise over the operator.

In the definition of the typing predicate, we give one rule for + and one for ≤. The impact of this choice is how the cases for the inductive proofs look when we do induction over the AST or the rules. If we go for a match between AST and grammar, such as with the fst and snd operators, then there will be a overlap in the proof text for the cases for these operators. The proofs follow the same overall pattern with difference being in how the value and type are projected out. In a paper proofs, the author can present the proof for just one case and appeal to proof by analogy for the other; in the mechanisation the analogy has to be made explicit.

There are number of avenues to explore making use of this work and the experience: The first is to investigate if the mechanisation can be used to generate an implementation of a type checking and interpreter for MiniSail. Code exporting is a well used feature of Isabelle however the challenge here is that code exporting for Nominal-Isabelle has not been solved. Second, formalise a larger subset of Sail either within the Nominal framework or use another representation. In the case of the latter, prove some sort of correspondence.

Acknowledgments

This work was partially supported by EPSRC grant EP/K008528/1 (REMS). We also thank Thomas Bauereiss for his suggestions on the Isabelle code.

References

[1] ARM 2015. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM. ARM DDI 0487A.h (ID092915).

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proc. ACM Program. Lang.* http://www.cl.cam.ac.uk/~pes20/sail/

[3] Arthur Charguéraud. 2011. The Locally Nameless Representation. *Journal of Automated Reasoning* 49 (2011), 363–408. https://doi.org/10.1007/s10817-011-9225-2

[4] Stefan Berghofer Christian Urban and Cezary Kaliszyk. 2013. Nominal 2. *Archive of Formal Proofs* (Feb. 2013). https://www.isa-afp.org/entries/Nominal2.html, Nominal 2.

[5] N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381 – 392. https://doi.org/10.1016/1385-7258(72)90034-0

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[7] Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *International Conference on Functional Programming (ICFP)*. arXiv:1306.6032[cs.PL].

[8] Brian Huffman and Christian Urban. 2010. A New Foundation for Nominal Isabelle. In *Proceedings of the First International Conference on Interactive Theorem Proving (ITP'10)*. Springer-Verlag, Berlin, Heidelberg, 35–50. https://doi.org/10.1007/978-3-642-14052-5_5

[9] Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. Needle & Knot: Binder Boilerplate Tied Up. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag New York, Inc., New York, NY, USA, 419–445. https://doi.org/10.1007/978-3-662-49498-1_17

[10] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. https://doi.org/10.1145/2535838.2535841

[11] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE, Toulouse, France. https://hal.inria.fr/hal-01238879

[12] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg.

[13] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP'08)*. Springer-Verlag, Berlin, Heidelberg, 230–266. http://dl.acm.org/citation.cfm?id=1813347.1813352

[14] Lawrence C. Paulson. 2010. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In *Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010*. 1–10. http://www.easychair.org/publications/paper/52675

[15] Lawrence C. Paulson. 2013. Gödel's Incompleteness Theorems. *Archive of Formal Proofs* 2013 (2013). https://www.isa-afp.org/entries/Incompleteness.shtml

[16] Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Information and Computation* 186, 2 (2003), 165 – 193. https://doi.org/10.1016/S0890-5401(03)00138-X Theoretical Aspects of Computer Software (TACS 2001).

[17] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

[18] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2007. Ott: Effective Tool Support for the Working Semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1291151.1291155

[19] Rok Strniša and Matthew Parkinson. 2011. Lightweight Java. *Archive of Formal Proofs* (Feb. 2011). https://www.isa-afp.org/entries/LightweightJava.html, Lightweight Java.

[20] Christian Urban and Christine Tasson. 2005. Nominal Techniques in Isabelle/HOL. In *Proceedings of the 20th International Conference on Automated Deduction (CADE' 20)*. Springer-Verlag, Berlin, Heidelberg, 38–53. https://doi.org/10.1007/11532231_4

[21] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. *SIGPLAN Not.* 49, 9 (Aug. 2014), 269–282. https://doi.org/10.1145/2692915.2628161

[22] Robert N. Watson et al. 2017. Capability Hardware Enhanced RISC Instructions (CHERI). http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/.

[23] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 20–37. https://doi.org/10.1109/SP.2015.9

[24] Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. ACM, New York, NY, USA, 53–65. https://doi.org/10.1145/3167082