A Generic Framework for Verified Compilers Using Isabelle/HOL's Locales

Martin Desharnais and Stefan Brunthaler

National Cyber Defence Research Institut (CODE), UniBw M, Germany
{martin.desharnais, brunthaler}@unibw.de

Abstract

In this paper, we present a prototype version of a generic framework for formalizing compiler transformations. Our framework leverages Isabelle/HOL's *locales*—a module system for generic formalizations—to abstract over concrete languages and transformations. The framework thus enables us to state common definitions for language semantics, program behaviours, forward and backward simulations, and compilers. We provide generic operations, such as compiler composition, and prove general theorems, resulting in reusable proof components. By demonstrating our idea on a concrete example, we provide evidence of how locales allow reuse and, therefore, enable encapsulation of verification artefacts into modules.

1 Introduction

The mechanically verified formalization of software components has been the subject of much research in the last decades. Especially influential were the CompCert [4] and CakeML [3] projects, which produced realistic compilers from a (large subset of) two real-world programming languages (C99 and Standard ML) to real hardware platforms. These compilers showed both that mechanized verification is feasible and that it has a measurable effect on the dependability of the compiler [7].

We can now observe a shift in perspective, where the idea of mechanically verified software components is becoming a concrete and desirable goal. Formalization projects are increasing in number, but also in size, complexity, and lifetime. There is an analogy to be made with the emergence, in the second half of the 20th century, of software engineering to the point that the term *proof engineering* starts to be used. New and interesting questions now emerge. How to avoid repetition in definitions and proofs? Which concepts can be generalized and reused? How to separate a formalization in independent components, so that multiple people can work in parallel? What should be the interface between such components? How can tooling make proof engineers more productive? What is a good balance between proof readability and the time required to (mechanically) verify it? etc.

In the case of compiler verification, we have a very well-understood domain, with well-known terminology, that builds on decades of research and empirical experience. But as is the case for a lot of small software prototypes, small-scale formalizations constantly redefined similar abstractions and concepts. This is something we wanted to avoid when we started a small formalization, in Isabelle/HOL [5], of three small stack-based languages implementing different optimizations. Inspired by the concept of modularization in software engineering, we separated the general concepts from the language-dependent parts. We learned about, and made use of, Isabelle's locales to devise a small generic framework[1] for the verification of program transformations.

[1] Available on the Archive of Formal Proofs (AFP) [2].

2 Background

In this section, we start with brief overview of the operational semantics of programming languages and follow with a short introduction to Isabelle's locales.

2.1 Programming Language Semantics

The operational semantics of a programming language can be defined as a transition system representing the execution of a program written in this language. A language $L = \langle S, I, F, \to \rangle$ is defined by a set S of program states, a set $I \subseteq S$ of initial states, a set $F \subseteq S$ of final states, and a transition relation $\to \subseteq S \times S$. The execution of a program is modelled as a sequence of states $s_1 \to s_2 \to ...$ with $s_1 \in I$. An execution is called terminating if there exists a state s_n such that $s_1 \to s_2 \to ... \to s_n$ and $\nexists s_{n+1}.\ s_n \to s_{n+1}$, and non-terminating otherwise. A terminating execution is said to be successful if $s_n \in F$ and to go wrong otherwise. These execution behaviours are usually called the program's behaviour and written $s \Downarrow b$.

The compiler from a language L_1 to L_2 is a partial function $\mathcal{C} : S_1 \rightharpoonup S_2$, which maps a program $s \in I_1$ to $\mathcal{C}(s) \in I_2$.

Two programs s and c are said to be equivalent if they exhibit the same behaviour, i.e. $\forall b, s \Downarrow b \iff c \Downarrow b$. This can be established using a bisimulation [6]: the conjunction of a backward and a forward simulation. Consider a binary relation $\approx$, between program states, expressing that two states are to be considered equivalent for a given use case. This relation is called a simulation whenever $\forall s\ s'\ c, s \approx c \wedge s \to s' \implies \exists c', s' \approx c' \wedge c \to c'$. A backward simulation, thus, shows that every behaviour of the compiled program is also a behaviour of the source program, i.e. the compilation is correct (sound). A forward simulation shows that every behaviour of the source program can be achieved by the compiled program, i.e. the compilation is complete.

2.2 Isabelle's locales

Locales are an Isabelle construct to define parametric theories [1]. They are based on the concept of proof contexts. A theorem of the form

$$\bigwedge p_1\, p_2 \ldots p_n.\, A_1 \Longrightarrow A_2 \Longrightarrow \cdots \Longrightarrow A_m \Longrightarrow C$$

has a set $\{p_1, p_2, \ldots, p_n\}$ of parameters, a set $\{A_1, A_2, \ldots, A_m\}$ of assumptions, and proves a conclusion C. Taken together, the sets of parameters and assumptions is called the proof context. Locales enable the user to define a named proof context and reuse it for multiple conclusions, thus avoiding having to repeat its components in every theorem. Consider for example a formalization of monoids.

```
locale monoid =
 fixes
  f :: 'a ⇒ 'a ⇒ 'a (infix ·) and
  e :: 'a
 assumes
  associativity: x · (y · z) = (x · y) · z and
  left-identity: e · x = x and
  right-identity: x · e = x
```

```
context monoid begin
 primrec pow :: nat ⇒ 'a ⇒ 'a where
  pow 0 x = e |
  pow (Suc n) x = x · pow n x
 lemma pow-add:
  pow (n + m) x = pow n x · pow m x
  proof ... qed
end
```

The *monoid* locale consists of a sequence of parameters, introduced by the **fixes** keyword, and a sequence of assumptions, introduced by the **assumes** keyword. When working in a locale context—introduced by the **context** command or directly following a locale definition—new definitions and theorems can be derived from the locale parameters, its assumptions, and previous derived terms and theorems.

The automatically introduced *locale predicate monoid* $:: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow bool$ identifies locale interpretations, i.e. parameters for which the assumptions hold. We can see that locale contexts really just are syntactic sugar for manual contexts by inspecting the theorem *monoid.pow-add* from outside the locale context.

$$monoid\, f\, e \Longrightarrow pow\, f\, e\, (n+m)\, x = f\, (pow\, f\, e\, n\, x)\, (pow\, f\, e\, m\, x)$$

Locales can also be extended by more parameters and assumptions.

```
locale monoid-homomorphisms =
  M_f: monoid f e_f + M_g: monoid g e_g
  for
    f :: 'a ⇒ 'a ⇒ 'a (infix ·) and e_f and
    g :: 'b ⇒ 'b ⇒ 'b (infix ◇) and e_g +
  fixes map :: 'a ⇒ 'b
  assumes
    map-distributive: map (x · y) = map x ◇ map y and
    map-identity: map e_f = e_g
```

The *monoid-homomorphisms* locale extends two instances of *monoid*, fixes a projection function *map* between their underlying types, and states two assumptions on its interaction with the two monoid structures. The extended locale contexts are named, so that elements can be accessed, e.g. by writing M_f*.left-identity*. The sequence of parameters required by the extended locales are introduced by the **for** keyword.

Finally, locales can be interpreted, with the **interpretation** command, by providing values for the parameters and proving that the assumptions hold.

```
interpretation monoid-nat-addition: monoid (+) (0 :: nat)
proof — Proof that the assumptions hold qed
```

Following interpretation, all derived definitions and theorems, specialized for the provided arguments, are available in the *monoid_nat_addition* namespace.

3 The Design of the Framework

The framework has three main components: some abstract definitions of languages and compilers using locales, a generic definition of program behaviour, and some composition operations over simulations and compilers.

3.1 Locales

The definition of programming languages is separated into two parts: an abstract semantics and a concrete program representation.

```
locale semantics =
  fixes step :: 'state ⇒ 'state ⇒ bool and final :: 'state ⇒ bool
  assumes final-finished: final s ⟹ finished step s
```

```
locale language = semantics step final
  for step and final :: 'state ⇒ bool +
  fixes load :: 'prog ⇒ 'state option
```

The *semantics* locale represents the semantics as an abstract machine. It is expressed by a transition system with a transition relation *step*—usually written as an infix $(\rightarrow)$ arrow—and final states *final*. The *language* locale represents the concrete program representation (type variable *'prog*), which can be transformed into a program state (type variable *'state*) by the *load* function. The set of initial states of the transition system is implicitly defined by the codomain of *load*.

locale *backward-simulation* =
L1: *semantics step1 final1* + *L2*: *semantics step2 final2* + *well-founded* $(\sqsubset)$
for
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and** *step2* :: *'state2* ⇒ *'state2* ⇒ *bool* **and**
final1 :: *'state1* ⇒ *bool* **and** *final2* :: *'state2* ⇒ *bool* **and**
order :: *'index* ⇒ *'index* ⇒ *bool* (**infix** $\sqsubset$) +
fixes *match* :: *'index* ⇒ *'state1* ⇒ *'state2* ⇒ *bool*
assumes
match-final: $match\ i\ s_1\ s_2 \Longrightarrow final2\ s_2 \Longrightarrow final1\ s_1$ **and**
simulation: $match\ i\ s_1\ s_2 \Longrightarrow s_2 \rightarrow_2 s_2{}' \Longrightarrow$
$(\exists\, i'\, s_1{}'.\ s_1 \rightarrow_1{}^{++}\, s_1{}' \wedge match\ i'\ s_1{}'\ s_2{}') \vee (\exists\, i'.\ match\ i'\ s_1\ s_2{}' \wedge i' \sqsubset i)$

A simulation is defined between two semantics *L1* and *L2*. A *match* predicate expresses that two states from *L1* and *L2* are equivalent. The *match* predicate is also parameterized with an ordering used to avoid stuttering.

The only two assumptions of a backward simulation are that a final state in *L2* will also be a final in *L1*, and that a step in *L2* will either represent a non-empty sequence of steps in *L1*—the $(\rightarrow_1{}^{++})$ relation is the transitive closure of the $(\rightarrow_1)$ relation—or will result in an equivalent state. Stuttering is ruled out by the requirement that the index on the *match* predicate decreases with respect to the well-founded $(\sqsubset)$ ordering.

locale *compiler* =
L1: *language step1 final1 load1* + *L2*: *language step2 final2 load2* +
backward-simulation step1 step2 final1 final2 order match
for
step1 **and** *step2* **and**
final1 **and** *final2* **and**
load1 :: *'prog1* ⇒ *'state1 option* **and** *load2* :: *'prog2* ⇒ *'state2 option* **and**
order :: *'index* ⇒ *'index* ⇒ *bool* **and** *match* +
fixes *compile* :: *'prog1* ⇒ *'prog2 option*
assumes *compile-load*: $compile\ p_1 = Some\ p_2 \Longrightarrow load1\ p_1 = Some\ s_1 \Longrightarrow \exists\, s_2\ i.\ load2\ p_2 = Some\ s_2 \wedge match\ i\ s_1\ s_2$

The *compiler* locale relates two languages, *L1* and *L2*, by a backward simulation and provides a *compile* partial function from a concrete program in *L1* to a concrete program in *L2*. The only assumption is that a successful compilation results in a program which, when loaded, is equivalent to the loaded initial program.

3.2 Behaviours

We define a generic datatype to encode three broad execution behaviours: successful termination (*Terminates*), non-terminating execution (*Diverges*), and going wrong (*Goes-wrong*).

datatype *'state behaviour* = *Terminates 'state* | *Diverges* | *Goes-wrong 'state*

Terminating behaviours are annotated with the last state of the execution in order to compare the result of two executions with the *rel-behaviour* :: $('a \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{behaviour} \Rightarrow 'b\ \mathit{behaviour} \Rightarrow \mathit{bool}$ relation.

$$\begin{aligned} f\, s_1\, s_2 \Longrightarrow\ & \mathit{rel\text{-}behaviour}\, f\ (\mathit{Terminates}\ s_1)\ (\mathit{Terminates}\ s_2) \\ & \mathit{rel\text{-}behaviour}\, f\ \mathit{Diverges}\ \mathit{Diverges} \\ f\, s_1\, s_2 \Longrightarrow\ & \mathit{rel\text{-}behaviour}\, f\ (\mathit{Goes\text{-}wrong}\ s_1)\ (\mathit{Goes\text{-}wrong}\ s_2) \end{aligned}$$

The exact meaning of the three behaviours is defined in the *semantics* locale, where a $(\Downarrow)$ :: $'state \Rightarrow 'state\ behaviour \Rightarrow bool$ relation is defined to assign an execution behaviour to a program state. The $(\rightarrow^*)$ relation is the reflexive transitive closure of the $(\rightarrow)$ relation and $(\rightarrow^\infty)$ is its coinductive, infinitely transitive closure. The predicate *finished* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ identifies a state that cannot make a transition.

$$\frac{s_1 \rightarrow^* s_2 \qquad \mathit{finished}\ (\rightarrow)\ s_2 \qquad \mathit{final}\ s_2}{s_1 \Downarrow \mathit{Terminates}\ s_2}\ \mathit{state\text{-}terminates} \qquad \frac{s_1 \rightarrow^\infty}{s_1 \Downarrow \mathit{Diverges}}\ \mathit{state\text{-}diverges}$$

$$\frac{s_1 \rightarrow^* s_2 \qquad \mathit{finished}\ (\rightarrow)\ s_2 \qquad \neg\, \mathit{final}\ s_2}{s_1 \Downarrow \mathit{Goes\text{-}wrong}\ s_2}\ \mathit{state\text{-}goes\text{-}wrong}$$

Even though the $(\rightarrow)$ transition relation in the *semantics* locale need not be deterministic, if it happens to be, then the behaviour of a program becomes deterministic too.

$$\frac{\bigwedge x\, y\, z.\ \dfrac{x \rightarrow y \qquad x \rightarrow z}{y = z} \qquad s \Downarrow b_1 \qquad s \Downarrow b_2}{b_1 = b_2}$$

The main correctness theorem states that, for any two matching programs, any not wrong behaviour of the later is also a behaviour of the former. In other words, if the compiled program does not crash, then its behaviour, whether it terminates or not, is a also a valid behaviour of the source program. The predicate *is-wrong* :: $'state\ behaviour \Rightarrow bool$ identifies wrong behaviours.

$$\frac{\mathit{match}\ i\ s_1\ s_2 \qquad s_2 \Downarrow_2 b_2 \qquad \neg\ \mathit{is\text{-}wrong}\ b_2}{\exists\, b_1\ i'.\ s_1 \Downarrow_1 b_1 \wedge \mathit{rel\text{-}behaviour}\ (\mathit{match}\ i')\ b_1\ b_2}$$

Because this theorem is proven in the context of the *backward-simulation* and, thus, only depends on its parameters and assumptions, it is independent of the concrete programming language, and need only be to be proven once. It automatically holds for all interpretations of *backward-simulation*.

As a corollary, the preservation of behaviour can be lifted to the compilation of concrete program representation.

$$\frac{\begin{array}{c}\mathit{compile}\ p_1 = \mathit{Some}\ p_2 \\ \mathit{load1}\ p_1 = \mathit{Some}\ s_1 \qquad \mathit{load2}\ p_2 = \mathit{Some}\ s_2 \qquad s_2 \Downarrow_2 b_2 \qquad \neg\ \mathit{is\text{-}wrong}\ b_2\end{array}}{\exists\, \mathit{b1}\ i.\ s_1 \Downarrow_1 \mathit{b1} \wedge \mathit{rel\text{-}behaviour}\ (\mathit{match}\ i)\ \mathit{b1}\ b_2}$$

3.3 Generic Composition of Simulations and Compilers

We define the generic composition of matching functions, $\diamond :: ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow bool) \Rightarrow ('d \Rightarrow 'c \Rightarrow 'e \Rightarrow bool) \Rightarrow 'a \times 'd \Rightarrow 'b \Rightarrow 'e \Rightarrow bool$, and orderings, $(<*lex*>) :: ('a \times 'a)\ set \Rightarrow ('b \times 'b)\ set \Rightarrow (('a \times 'b) \times 'a \times 'b)\ set$, such that the composition of two backward simulations is itself a backward simulation.

$$\frac{\begin{array}{c} \textit{backward-simulation}\ (\rightarrow_1)\ (\rightarrow_2)\ \textit{final}_1\ \textit{final}_2\ (\sqsubset_1)\ (\approx_1) \\ \textit{backward-simulation}\ (\rightarrow_2)\ (\rightarrow_3)\ \textit{final}_2\ \textit{final}_3\ (\sqsubset_2)\ (\approx_2) \end{array}}{\textit{backward-simulation}\ (\rightarrow_1)\ (\rightarrow_3)\ \textit{final}_1\ \textit{final}_3\ (\textit{lex-prodp}\ ({\sqsubset_1}^{++})\ (\sqsubset_2))\ ((\approx_1) \diamond (\approx_2))}$$

We define the generic $(\Leftarrow) :: ('a \Rightarrow 'b\ option) \Rightarrow ('c \Rightarrow 'a\ option) \Rightarrow 'c \Rightarrow 'b\ option$ composition operator on compilers, which corresponds to the monadic bind of the *option* type found in a compiler's codomain.

$$(\mathcal{C}_2 \Leftarrow \mathcal{C}_1)\ p \equiv \textit{Option.bind}\ (\mathcal{C}_1\ p)\ \mathcal{C}_2$$

Its correctness can then be generically proven for any two interpretations of the *compiler* locale.

$$\frac{\begin{array}{c} \textit{compiler}\ (\rightarrow_1)\ (\rightarrow_2)\ \textit{final}_1\ \textit{final}_2\ \textit{load}_1\ \textit{load}_2\ (\sqsubset_1)\ (\approx_1)\ \mathcal{C}_1 \\ \textit{compiler}\ (\rightarrow_2)\ (\rightarrow_3)\ \textit{final}_2\ \textit{final}_3\ \textit{load}_2\ \textit{load}_3\ (\sqsubset_2)\ (\approx_2)\ \mathcal{C}_2 \end{array}}{\textit{compiler}\ (\rightarrow_1)\ (\rightarrow_3)\ \textit{final}_1\ \textit{final}_3\ \textit{load}_1\ \textit{load}_3\ (\textit{lex-prodp}\ ({\sqsubset_1}^{++})\ (\sqsubset_2))\ ((\approx_1) \diamond (\approx_2))\ (\mathcal{C}_2 \Leftarrow \mathcal{C}_1)}$$

4 An Instantiation of the Framework

The first programming languages for which we instantiated the framework are three interpreted, stack-based languages. The first one, *Std*, is a standard assembly language with push/pop and load/store instructions, conditional jumps, n-ary built-in operations, and (possibly recursive) function calls. The second language, *Inca* expands *Std* with inline caching, i.e. operations that are faster for specific operand types but fallback to a generic operation otherwise. The third language, *Ubx*, goes one step further by introducing operations that operate on unboxed operands.

The matter of study for these languages was the preservation of a program's behaviour after its operations have been optimized. To this end, we abstracted away from many concrete, yet irrelevant details.

```
locale env = ...
locale nary-operations = ...
datatype ('var, 'fun, 'op) instr = ...
datatype 'instr fundef = ...
datatype ('fenv, 'menv, 'var, 'fun, 'op) state = ...
datatype ('fenv, 'henv, 'fun) prog = ...
locale std =
 Fenv: env F-empty F-get F-add F-to-list +
 Henv: env M-empty M-get M-add M-to-list +
 nary-operations eval-op arity-op
  for
   F-empty and F-get :: 'fenv ⇒ 'fun ⇒ ('var, 'fun, 'op) instr fundef option and
   F-add and F-to-list and
   M-empty and M-get :: 'menv ⇒ 'var ⇒ 'value option and
  M-add and M-to-list and
  eval-op :: 'op ⇒ 'value list ⇒ 'value and arity-op
begin
 inductive step :: ('fenv, 'menv, 'var, 'fun, 'op) state ⇒ ('fenv, 'menv, 'var, 'fun, 'op) state ⇒ bool where ...
 inductive final :: ('fenv, 'menv, 'var, 'fun, 'op) state ⇒ bool where ...
 definition load :: ('fenv, 'menv, 'fun) prog ⇒ ('fenv, 'menv, 'var, 'fun, 'op) state option where ...
 lemma final-finished: final s ⟹ finished step s ...
 sublocale std-sem: semantics step final ...
 sublocale std-lang: language step final load ...
end
```

The locale *env* expresses a dynamic key-value environment, of which we use two instances to hold function definitions and dynamic memory. The locale *nary-operations* expresses a set of operations, each of which may have a different arity, assorted with an evaluation function.

Because locales do not support the definition of new types, the *instr*, *fundef*, *prog*, and *state* datatypes needed to be defined in the top-level theory. Moreover, these datatypes need to abstract over multiples types, which are fixed only inside the *std* locale. To name the different abstract types in the **for** section, we provided the minimum possible mount of type annotations.

The *step* and *final* relations, and the *load* function all depend on the fixed locale parameters and types and, thus, need to be defined inside the locale. They instantiate the generic datatypes using the locale's fixed types.

The lemma *final-finished* can then be stated and proven. Finally, the *semantics* and *language* locales can be interpreted[2], thereby proving that *Std* corresponds to our abstraction of language with a semantics. This also specializes all general results of said locales to this concrete language definition. The two other languages, *Inca* and *Ubx*, follow the same structure.

```
locale std-inca-simulation =
 Lstd: std Fstd-empty Fstd-get Fstd-add Fstd-to-list M-empty M-get M-add M-to-list
  eval-op arity-op +
 Linca: inca Finca-empty Finca-get Finca-add Finca-to-list M-empty M-get M-add M-to-list
  eval-op arity-op eval-opinl inl deinl
 for
  Fstd-empty and Fstd-get :: 'fstd-env ⇒ 'fun ⇒ ('var, 'fun, 'op) Std.instr Std.fundef option and
  Fstd-add and Fstd-to-list and
  Finca-empty and Finca-get :: 'finca-env ⇒ 'fun ⇒ ('var, 'fun, 'op, 'opinl) Inca.instr Inca.fundef option and
  Finca-add and Finca-to-list and
  M-empty and M-get :: 'menv ⇒ 'var ⇒ 'value option and
  M-add and M-to-list and
  eval-op :: 'op ⇒ 'value list ⇒ 'value and arity-op and
  eval-opinl and inl and deinl :: 'opinl ⇒ 'inl
begin
 inductive match ::
  nat ⇒
  ('fstd-env, 'menv, 'var, 'fun, 'op) Std.state ⇒
  ('finca-env, 'menv, 'var, 'fun, 'op, 'opinl) Inca.state ⇒
  bool where ...
 lemma match-final: match i s1 s2 ⟹ Linca.final s2 ⟹ Lstd.final s1 ...
 lemma simulation: match i s₁ s₂ ⟹ Linca.step s₂ s₂' ⟹
  (∃ i' s₁'. Lstd.step⁺⁺ s₁ s₁' ∧ match i' s₁' s₂') ∨ (∃ i'. match i' s₁ s₂' ∧ i' < i) ...
 sublocale std-inca-backward-simulation:
```

[2] The **sublocale** command is a variation of the **interpretation** command.

```
  backward-simulation Lstd.step Linca.step Lstd.final Linca.final (<) match ...
end
```

Proving a backward simulation between *Std* and *Inca* requires to extend the *std* and *inca* locales, define the required *match*, prove the two required *match-final* and *simulation* lemmas, and finally interpret the *backward-simulation* locale. Both languages have distinct environment types for function definitions but share the same environment type for dynamic memory. They also share the same set of built-in operations.

```
context std-inca-simulation begin
 definition compile :: ('fstd-env, 'menv, 'fun) Std.prog ⇒ ('finca-env, 'menv, 'fun) Inca.prog option where ...
 lemma compile-load: compile p1 = Some p2 ⟹ Lstd.load p1 = Some s1 ⟹ ∃ s2 i. Linca.load p2 = Some s2
∧ match i s1 s2 ...
 sublocale std-to-inca-compiler:
  compiler Lstd.step Linca.step Lstd.final Linca.final Lstd.load Linca.load (<) match compile ...
end
```

Defining the compiler and proving its correctness can be done in the context of the *std-inca-simulation* locale, because no new fixed parameters or types are required. It only requires to define the compilation function, prove the *compile-load* lemma instantiate the *compiler* locale.

With this last instantiation, the framework automatically instantiates the theorem on preservation of behaviour for compiled programs, which is the property we were interested in.

5 Discussion

Using locales as a modularization tool for our generic framework turned out to be elegant at times and frustrating in other cases.

5.1 Strengths of the Approach

Parameters, assumptions, and derived elements are clearly separated. The syntax used to define a locale enables the user to clearly state the parameters and assumptions that are abstracted over. Derived elements such as function definitions and lemmas are clearly separated by being defined later in a locale context. The fact that these extensions can be done at any point following the locale's definition gives a lot of flexibility when structuring the formalization.

It is possible to abstract over multiple types. Locales enable fixed variables to depend on multiple type variables. This makes them more general than type classes, with which they have otherwise a lot in common. While traditional type classes permit to abstract over operations on a given abstract type, locales permit to abstract over both operations on concrete types and multiple abstract types. In fact, type classes in Isabelle/HOL are just syntactic sugar for locales with a single type variable.

It is possible to have multiple interpretations for a given set of type. Because a locale interpretation introduces a new namespace when specializing the derived elements, multiple instantiations are possible for a given set of types. A classical example for such a situation is a partial order over the integers. Using traditional type classes, one has to decide a canonical order that will be associated with the integer type. In order to use an alternate order, one has to define a bijection to an alternative type which instantiate the type class accordingly. As many distinct types and bijections are required as distinct instantiations are wished.

5.2 Weaknesses of the Approach

Parametric types and type aliases cannot be defined in locales. This limitation requires the user to generalize data types to abstract over any type variable fixed in the locale definition and define them outside of the locale. This was e.g. the case for the *instr*, *fundef*, *prog*, and *state* data types of the *Std*, *Inca*, and *Ubx* languages. This generalization is trivial, since a fixed type variable in a locale is akin to a type variable in a data type definition. The burden shows up when referring to the generic type in a type annotation, where it must be explicitly instantiated. Because parametric type aliases are also not supported, the instantiation has to be repeated over and over. As the number of type variables increases, type annotations become complex, long to write, and hard to read.

When extending existing locales, type annotations on fixed variables are required to name type variables. These variables appear in the **for** section and their types are inferred from their usage in instantiating the extended locales to be extended. Type inference even succeeds in cases some type variables must be unified between multiple locale instantiations, as is the case in the *compiler* locale. The user must nevertheless provide some type annotations in order to name the type variables that will be referred later. In practice, most of them are requires in type annotations.

Proving lemmas involving locale predicates have considerable syntactic overhead. Consider for example the *compiler_composition* lemma, where two hypotheses and the conclusion are locale predicates of the *compiler* locale. Proving this lemma involves accessing the *language* instance predicates accessible with expressions, such as *assms(1)[THEN compiler.axioms(1)].* The problem with this syntax is twofold: (i) it depends on the order in which the axioms were stated, and (ii) it does not scale well when the user needs to extract multiple axioms from multiple assumptions. The first problem could be solved by automatically adding lemmas using the name of the extension, e.g., *compiler.L1* would be a synonym of *compiler.axioms(1)* to refer to the first *language* instance predicate of the compiler's definition. The second could be alleviated if unnamed contexts supported locales extension.

References to a locale's fixed variables and derived definitions are syntactically different. When extending locales, as is the case in the *backward_simulation* locale, derived definitions of the two languages are accessible with uniform names in some namespaces, such as *L1.behaves* and *L2.behaves*. Fixed parameters, by contrast, are only accessible using their given name, e.g., *step1* and *step2*. Even though explicitly naming locale parameters may be omitted in simple cases, it is require as soon as two locales fix parameters with the same name. While writing locales for software abstractions, as opposed to mathematical structures, we observed that fixed parameters must be named in all but the most trivial cases.

The lack of a uniform syntax to access derived definitions and parameters also has an undesired impact on refactoring. When replacing a fixed parameter by an equivalent derived definition, the user may not just have adapt all interpretations, but also all derived definitions and theorems to use the prefixed name. Although this would not necessarily be a problem in the absence of name clashes, a uniform naming scheme allowing the systematical use of the interpretation's prefix benefits both, the locale's design and implementation.

The syntax overhead of locale extension increases with the number of fixed parameters or types. This direct proportional relation is evident from the definitions of the *std-inca-simulation* locale. The current syntax seems to serve two purposes: (i) provide unique names for fixed parameters, and (ii) state which abstract types are shared in parameters' types.

To satisfy the first purpose, one could name the locale instances and use the uniform naming scheme mentioned above, thereby increasing their utility and applicability. To satisfy second purpose, we believe that the locale mechanism should offer an alternative syntax that allows for a more succinct way to express type dependencies between different locale extensions.

6 Conclusion

We presented the first version of a generic framework for formalizing compilers in Isabelle/HOL. It is based on locales to abstract over the concrete languages and program transformations, provides general definitions for program behaviours and compiler compositions, and generically proves preservation of behaviour. This framework emerged as a side product of our formalization of three stack-based languages that implementing different optimizations. It helped us to emphasis the commonalities between the different theories and to reduce some duplication. Possible future work includes extending the semantics to support traces, exploring how to extract executable programs from such formalizations, and exploring how to simplify or automate repetitive operations such as the composition of multiple compilers.

Our experience indicates that the use of locales in Isabelle is held back by its potential. On the one hand, the additional power afforded by locales to structure proof developments is an enormous benefit. This benefit is particularly relevant for the domain of formalizing and verifying software artifacts, as demonstrated by our exemplary development.

On the other hand, the syntactic overhead experienced clearly represents an obstacle to adopting Isabelle. If the overhead were addressed by Isabelle in one way or another, we firmly believe that Isabelle could position itself as the premier choice for programming language semantics formalization and verification—a domain that is becoming increasingly important and that could well use the excellent automation capabilities offered by Isabelle.

7 Acknowledgement

This paper is part of the project CONCORDIA, a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927.

References

[1] C. Ballarin. Locales: A Module System for Mathematical Theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014.

[2] M. Desharnais. A generic framework for verified compilers. *Archive of Formal Proofs*, Feb. 2020. https://isa-afp.org/entries/VeriComp.html, Formal proof development.

[3] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, Jan. 2014.

[4] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[5] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[6] D. Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[7] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.