Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 254 (2009) 161–179

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

State Spaces — The Locale Way

Norbert Schirmer[1,2]

German Research Center for Artificial Intelligence (DFKI) Saarbrücken, Germany

Makarius Wenzel[1,3]

Technische Universität München, Institut für Informatik

Abstract

Verification of imperative programs means reasoning about modifications of a program state. So proper representation of state spaces is crucial for the usability of a corresponding verification environment. In this paper we discuss various existing state space models under different aspects like strong typing, modularity and scalability. We also propose a variant based on the locale infrastructure of Isabelle. Thus we manage to combine the advantages of previous formulations (without suffering from their disadvantages), and gain extra flexibility in composing state space components (inherited from the modularity of locales).

Keywords: software verification, theorem proving

1 Introduction

The core of any kind of imperative program is the update of a system state. Every theorem proving approach for reasoning about imperative programs involves a formal representation of the system state at the base of the program calculus, whether this is UNITY [1], TLA [15], Hoare logics [8,11,20], or others [9,22,6].

[1] Work supported by German Federal Ministry of Education and Research (BMBF) under grant 01 IS C38.
[2] Email: norbert.schirmer@dfki.de
[3] Email: wenzelm@in.tum.de

doi:10.1016/j.entcs.2009.09.065

An adequate formal model for representing program state is a delicate issue. The model has to be sufficiently detailed to express the properties to be verified, but excessive detail may pose a burden to interactive verification due to "formal noise" that conceals the interesting problems. Even more, some specific properties of a programming language may have to be reflected in the state space model. E.g. Java ensures that only initialised variables are accessed, whereas C lacks such guarantees. In the latter case, it is desirable to ensure the absence of illegal memory accesses by formal verification. For Java a state space model that abstracts from initialisation is fine, but it would be ill-suited for C.

This short discussion already shows that we cannot expect a single solution that fits best to all possible applications. Our particular motivation for this work was reasoning about C0 programs (a type-safe subset of C), within the Verisoft project.[4] Here the state was represented as a record in a general Hoare logic environment [20], implemented in Isabelle/HOL [17]. It turned out that the main limitation of this record representation is a lack of compositionality and scalability in the large verification tasks of Verisoft.

In the present paper we introduce an improved version of state spaces, which is also of interest beyond Hoare logic reasoning: it can be viewed as a general concept for abstract open records in HOL with support for multiple inheritance.

Preliminaries. Isabelle is a generic logical framework which works with different object logics. We only refer to Isabelle/HOL [17], which is an implementation of higher-order logic augmented with facilities for defining datatypes, records, inductive sets, recursive functions etc. The implementation language of Isabelle is SML, which can also be used at run-time to program and extend the system in a logically sound manner, according to the well-known "LCF approach".

Keywords like **lemma**, **record**, **locale**, etc. refer to Isabelle theory commands. The notation for types, terms and propositions approximates standard mathematical notation, with a bias towards λ-calculus. There are the usual type constructors $T_1 \times T_2$ for product and $T_1 \Rightarrow T_2$ for the total function space. For type variables we use greek letters α, β, γ or alternatively $'name$ for longer names. For type constants we use plain identifiers (e.g. $name$). The term language refers to abstraction $\lambda x{::}\alpha.\ b$ (where types are usually left implicit thanks to type-inference), and curried function application as in $f\ x\ y$. Functional update of f at position x with new value v is written $f(x := v)$. The Isabelle framework expresses proof rules in Natural Deduction style, using $\bigwedge$ for quantification and $\Longrightarrow$ for implication.

[4] http://www.verisoft.de

Overview. In §2 we examine some expected requirements of state space models. In §3 we discuss existing state space representations, covering functions, tuples, records, and abstract types. In §4 we introduce our variant based on locales in Isabelle, and report on its application in the context of a Hoare logic.

2 Requirements of State Space Models

In principle, modelling state spaces for imperative programming languages is trivial. In mathematics, a state may be seen as a function from a set of names to a set of values, with range restrictions $V\ n$ depending on field names n, i.e. $s\colon N \rightharpoonup \mathcal{V}$ where $\mathcal{V} = \bigcup_{n \in N} V n$ and $\forall\, n \in N.\ s\ n \in\ V\ n$. In an untyped logic like set theory this is a reasonable approach [21,13]; the same idea may be formalised in type-theory as a dependent function space. As we intend to produce formal reasoning tools in the end, we now step back from purely logical foundations and reconsider high-level requirements arising in practical verification.

Lookup and update. The most basic features of a state space is the lookup and update of a variable, as they appear in programming language expressions, assertions, or in statements. To reason about a global state it is also crucial to express so-called frame conditions, the parts of a state *that do not change* during certain operations. Putting those aspects together we need means to access an *individual* variable and also its *complement* (all other variables).

Typing. Typed programming languages structure the program state by assigning different types to the variables. Programming language types can either be mapped to HOL types or HOL terms (e.g. as sets). If the programming language is type-safe and the HOL type system is expressive enough it is desirable to map the program types directly to HOL types. Thus strong typing of the underlying logic is directly employed to support verification of imperative programs.

Modularity. When composing a system from several components the question of modularity of the reasoning framework appears. Immediate compositionality demands a uniform representation of the state space of the different components. Otherwise intermediate steps may have to be introduced to lift a component and a property to the combined state space. If components are replicated, renaming may also become important for compositionality. E.g. consider a library for linked heap lists that regards only a *next* pointer for operations like append, reverse, etc. This basic structure may appear in various kinds of lists, like strings or queues. Then it is desirable to verify the library only once for an abstract *next*-field, and instantiate it later for various kinds of lists present in the application.

Scalability. We want to handle non-trivial programs, say with hundreds of global variables (e.g. a compiler for C0 written in C0). The state space model needs to support this, despite the resource limitations of contemporary computers.

3 Existing Models

3.1 State as Function

The first attempt to embed a programming logic into HOL is the work of Gordon [8]. He represents the state as function from names to values: *name* ⇒ *value*. Variable names are first-class objects of HOL, which means that we can quantify over them. For example, the frame condition $\forall x.\ x \neq y \longrightarrow s_1\ x = s_2\ x$ expresses that states s_1 and s_2 may only be different for variable y. The domain of all variables is represented by the same HOL type, namely *value*. Gordon and later Homeier [11] only consider programs with variables ranging over numbers. Treating variables of different types or even composite types like arrays requires a more complex representation of values. In his formalisation of Dijkstra, Harrison [9] uses an inductive datatype to address this problem, e.g.:

$$\textbf{datatype}\ \mathit{value} = \mathit{Intg}\ \mathit{int} \mid \mathit{Bool}\ \mathit{bool} \mid \mathit{Array}\ \mathit{value}\ \mathit{list}.$$

The different representations for integers (*int*), Booleans (*bool*) and arrays (*value list*) are injected into the type *value* by the constructors *Intg*, *Bool* and *Array*. By modelling arrays as lists of values also nested arrays can be expressed. An example array of array of integers is *Array* [*Array* [*Intg* 1, *Intg* 2], *Array* [*Intg* 3, *Intg* 4]]. A drawback of this approach is that a mixed array like *Array* [*Intg* 1, *Bool b*] is a perfectly legal *value* but is typically ruled out by the type system of the programming language. This issue carries on to expressions, where we have to explicitly deal with programming language typing within HOL. Consider the simple statement `x := y + 1`. Such an assignment boils down to a function update in our state space. To handle the addition we somehow have to lift the HOL addition that is defined for type *int* to type *value*. There are two possibilities: *project the arguments* or *lift the operation*. Even in case of a deep embedding of the expression and statement language the evaluation function implements one of those two possibilities (or maybe a mixture of both of them).

Projecting arguments (aggressive evaluation): For each variable type we define a projection function. E.g. the function *the-Intg* for type *int*:

$$\textit{the-Intg} :: \textit{value} \Rightarrow \textit{int}$$

$$\textit{the-Intg}\ (\textit{Intg}\ i) = i$$

Since HOL functions are total the term *the-Intg* $(\textit{Bool}\ b)$ is legal, but results in an unspecified *int*. The assignment `x := y + 1` is modelled as:

$$s(x := \textit{Intg}\ (\textit{the-Intg}\ (s\ y) + 1)).$$

Here s is the current state, which is updated at position x. We have to insert projections and injections into the original expression, which carries on to assertions about the program and therefore clutters up the verification task, unless we find some means to hide these indirections.

Lifting operations (type-sensitive evaluation): In this approach we explicitly fix the binary operations and define their evaluation for *value*s:

$$\textbf{datatype}\ \textit{bop} = \textit{Add} \mid \textit{And}$$

$$\textit{eval} :: (\textit{bop} \times \textit{value} \times \textit{value}) \Rightarrow \textit{value}$$

$$\textit{eval}\ (\textit{Add},\ \textit{Intg}\ n,\ \textit{Intg}\ m) = \textit{Intg}\ (n + m)$$

$$\textit{eval}\ (\textit{And},\ \textit{Bool}\ b,\ \textit{Bool}\ c) = \textit{Bool}\ (b \wedge c)$$

Again *eval* is under-specified, if the arguments have different types. In this setting our assignment `x := y + 1` becomes $s(x := \textit{eval}\ (\textit{Add},\ s\ y,\ \textit{Intg}\ 1))$.

Since the set of possible operations is made explicit by the datatype *bop*, the evaluation function *eval* can take care of typing issues and implicitly perform the projections from *value*. However, primitive values like 1 have to be injected into type *value* now. Moreover, basic properties of the operations only hold for correctly typed expressions. E.g. commutativity of addition: $\textit{eval}\ (\textit{Add},\ n,\ m) = \textit{eval}\ (\textit{Add},\ m,\ n)$ only holds, if we know that both arguments are of the form *Intg i*. In this case we can reduce the addition on type *value* to the ordinary integer addition and inherit its properties. We need to insert those explicit type constraints into the assertions about the program to be able to lift the logical properties of the operations for types *int* or *bool* to type *value*. This basically means that we prove type safety of evaluation every time we reason about expressions. This is annoying, since for a type-safe programming language this can be shown once and for all.

Comparing the two approaches, the first works well when type-safety is already guaranteed by the programming language. Then it is sufficient to use the aggressive evaluation strategy for expressions and assertions. Type-safety is naturally reflected in the logical representation in the sense that the projections and injections cancel each other in individual programs. E.g. if variable i is supposed to store an *int*, every update introduces the constructor *Intg* and a lookup (as it may appear in an assertion) uses the corresponding destructor *the-Intg*, but not *the-Bool* etc. The abstraction level on which assertions are formulated is *the-Intg* $(s\ i)$ and not something like $s\ i = \mathit{Intg}\ n$. This uniform view on a state as holding atomic entities in the projected form *the-Intg* $(s\ i)$ avoids implicit type constraints that would have to be discharged later. This is exactly the same view as provided by the other state space representations discussed below.

The second approach demands explicit typing constraints in assertions. This only makes sense if type-safe execution is not guaranteed and is therefore an essential part of the verification or if the type-system of the programming language cannot easily be mapped to the simple types of HOL.

In general, representing the state as a function leads to a uniform representation for all components and thus the components can be developed independently of each other. To achieve compositionality the tool only has to ensure that the names used in different components are distinct. Using strings as names and some kind of name mangling seem to be appealing at first sight. However, strings in Isabelle/HOL are implemented as lists of characters and are rather heavyweight objects in this setting, where the only required property is to check whether names are different. Renaming of variables is not easily achieved, since it demands an explicit transformation of the program and the assertions.

The restriction to one common universal type of values is another (theoretical) burden. We need to know in advance which values are embedded. This contradicts the very idea of truly modular development of components. However, as long as all the different values of the programming language can be embedded into an inductive datatype once and for all, this is not a practical issue.

3.2 State as Tuple

As alternative to states as functions, Wright *et al.* [22] propose tuples. Variables are identified by position in the tuple rather than by name. E.g. the tuple $\mathit{int} \times \mathit{int} \times \mathit{bool}$ represents a state space with three variables of type *int*, *int*, and *bool*, respectively. Each variable thereby has an individual HOL type. The typing issues of the “state as function” approach are eliminated since the artificial super-type for all variables is avoided. Variable types are identified with HOL types, and type inference ensures well-typed expressions.

By choosing the names of bound variables when abstracting over the state space, one can even annotate expressions with the programming language variables. Abstraction naturally occurs in assertions, if they are represented as predicates $state \Rightarrow bool$, or in update functions $state \Rightarrow state$. E.g. the state update of our running example `x := y + 1` can be encoded in the following function:

$$\lambda(x,\ y,\ b).\ (y + 1,\ y,\ b)$$

Using λ-abstraction the variables of the tuple are named x, y and b. If all variables are known, this translation from the assignment to the state update can be handled by a mere syntactic translation. However, great care has to be taken, since those translations have to account for all variable names and their order in the tuple. Moreover, names of bound variables can only be considered as comments for the reader. Due to α-conversion there is no logical difference between $\lambda(x,\ y,\ b).\ (y + 1,\ y,\ b)$ and $\lambda(n,\ m,\ k).\ (m + 1,\ m,\ k)$. Note that a one-to-one translation between the input and output syntax is not always possible. Consider the two assignments `x := x` and `y := y`. Both would be mapped to the same internal form: $\lambda(x,\ y,\ b).\ (x,\ y,\ b)$, the identity function.

Since variables lack proper names, we cannot quantify over them. Fortunately, typical assertions do not quantify over variables, but merely refer to their values within the state. This works in the same fashion as the state update above. How can we express frame conditions? To specify that only the value of `y` may change, one can list all *other* variables: $x_1 = x_2 \wedge b_1 = b_2$. This is how frame conditions work out in principle, but the main drawback is poor modularity. Every time we add a new variable to the program, we have to adapt those specifications.

Poor compositionality is also caused by the lack of a uniform state for all components. If we attempt to combine two components we can first build the Cartesian product of the underlying state spaces and try to rerun the old proofs. However, this will only work if the variable names occurring in the proofs are distinct to begin with. Elsewise we could try to come up with a calculus for composition, that lifts components to the Cartesian products.

Scalability of the tuple approach is limited in Isabelle/HOL. The problem is that the state tuple is explicitly split in every expression like $\lambda(x,\ y,\ b).\ (x',\ y',\ b')$. The type information stored in such a split tuple grows quadratically with the size of the tuple: the underlying *Pair* constructor is polymorphic: $Pair{::}\alpha \Rightarrow \beta \Rightarrow \alpha \times \beta$. Every constructor application is fully annotated with its type and a tuple $(x_1,\ x_2,\ldots)$ is internally $Pair\ x_1\ (Pair\ x_2\ \ldots)$. The situation is similar for tuple abstraction, which is based on $split :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha \times \beta \Rightarrow \gamma$.

3.3 State as Record

Records are similar to tuples, but additionally allow us to give proper names to variables. They were proposed by Wenzel [23] as state space representation and successfully used by Prensa [19] for the verification of parallel programs and by Paulson [18] for the formalisation of UNITY. Records enhance tuples by supplying selection and update functions for each constituent. For example,

record *state* =
  $x :: int$
  $y :: int$
  $b :: bool$

yields the selectors $x :: state \Rightarrow int$, $y :: state \Rightarrow int$ and $b :: state \Rightarrow bool$, and update functions $x\text{-}update :: int \Rightarrow state \Rightarrow state$, $y\text{-}update :: int \Rightarrow state \Rightarrow state$ and $b\text{-}update :: bool \Rightarrow state \Rightarrow state$. A record update $x\text{-}update\ i\ s$ is written as $s(\!|x := i|\!)$. Then `x := y + 1` becomes $s(\!|x := y\ s + 1|\!)$.

With selectors and updates as explicit functions, it is also easy to provide notation for program expressions, commands and annotations [23], closely resembling informal presentations. For example, the assertion $\{s.\ y\ s = x\ s + a\}$ (a set-comprehension over states) may then be written as $\{\!|\mathsf{y} = \mathsf{x} + a|\!\}$.

As with tuples we still cannot quantify over variable names, since record field names are not first-class objects of HOL. A field is merely characterised by its selection and update functions. However, due to extensional equality we can now specify that only `y` may change, without having to mention the other variables: $\exists\, i.\ s_2 = s_1(\!|y := i|\!)$. Both in this specification and the assignment above, only the relevant portions of the state space occur. This improves modularity compared to tuples. If the framework takes care to assign distinct variable names to different components, we can replace the record of a component with the record containing all variable names of all components and the proofs still remain valid when we rerun them. Even better, we can avoid rerunning the session by exploiting the extensibility of records in Isabelle/HOL [16,17]. Every record has an extension field "..." of arbitrary type. By instantiating this slot with a new chunk of record-fields the record can be extended. This new chunk again contains a polymorphic "..." field for further extensions. Thanks to this structural sub-typing, we get linear extensions of records essentially for free. Brucker and Wolff [6] handle single inheritance of object oriented programs by adding another dimension to this construction, employing a polymorphic sum. This allows to add various subclasses on the same level.

To get beyond a linear development of component states, we can develop a calculus for record composition, by defining an operator that transforms the state space by lifting all operations and assertions from a component state to the compound state. This approach is implemented and elaborated for a Hoare logic [20], but it gets technically tedious to implement in the tool and is rather heavyweight. Paulson [18] has developed a theory of program composition for UNITY, experiencing similar inconveniences.

Scalability of records is better than for raw tuples. By using lookup and update functions the record is not explicitly split in every expression. Moreover, the representing type for records may be grouped according to the structure of record extensions, not individual record fields. This reduces the number of nested tuple constructions. However, the 2008/2009 version of the record package in Isabelle/HOL still decomposes records internally to prove some auxiliary theorems, e.g. in the simplification procedure that proves that an update of a field x does not affect the value of another field y. Hence the performance of records also suffers from their size. In Isabelle/HOL records are defined as an abstraction on tuples nested to the right. Better results may be achieved by balanced nesting of tuples as a binary tree, as for example implemented in HOL4.[5]

3.4 State as Abstract Type

Merz [15] aims at a uniform global state representation combined with strong typing of variables. The idea is to regard the state as an abstract type *state* with a co-algebraic structure imposed by inspectors like $N :: state \Rightarrow nat$ and $B :: state \Rightarrow bool$ to represent the variables. Rather than defining the type and the inspector functions, their behaviour is characterised axiomatically. Update functions are not directly supported but in a relational framework the effect can be described with the inspector functions alone, e.g. $\{(s_1,\ s_2).\ N\ s_2 = N\ s_1 + 1\}$. The axioms basically need to ensure that variables can be updated independently, which requires all variables to be known in advance. So modularity is limited to addition of new variables without old proof scripts breaking. Similarly, Back and von Wright [2] axiomatise both the lookup and update functions.

Without having to rely on global axioms, Heyd and Crégut [10] employ the *section*-concept of Coq to declare only the parts of the state that are used by a component within their formalisation of UNITY. The state is not completely abstract, but modelled as a dependent function from variable names to the domain of each variable. The sections maintain the assumptions on the distinctness of variables and their types and provide projection and injec-

[5] Personal communication with Michael Norrish.

tions similar to the "state as function" approach. Given a compound state, which holds all variables of all components, the assumptions of the individual components can be discharged. The individual correctness results for each component are transferred to the compound state. However, Coq sections cannot be easily reassembled after being closed. So incremental development of libraries building on each other demands a lot of manual work. Heyd and Crégut also request better tool support to handle notation that hides the projections and injections.

4 The Locale Way

Let us briefly reconsider the particular benefits of the different approaches discussed so far. State as a tuple gives us strong typing for the variables, and records provide a convenient view on the state, by providing dedicated lookup and update functions for variables. Regarding compositionality, the clear winner is the "state as function" approach, for the following intuitive reason: imperative programs and the typical specifications we intend to prove only require the state to have *at least* a particular set of distinct variables. This idea is directly captured in a function with an infinite domain of names. Whereas tuples and records are overly specific in the sense that they restrict the state to *exactly* these variables.

The problems described with the "state as function" approach essentially boil down to the need to make everything *concrete*, such as concrete names for variables and a concrete universal type for all possible values. This makes it hard for the tool implementor to provide a modular framework for the independent development of different components.

We now employ Isabelle locales [12,3,4], to make everything *abstract*. Names are represented by a type variable *′name* and we assume all free variables, of that type, which are used in the component, to be distinct. Similarly, values are of type *′value* and we assume the presence of injection and projection functions that we need for a concrete component. All reasoning about programs is carried out abstractly relative to these assumptions. Locales enable rename, merge and addition of state space components as we compose our program segments. When the main program (or procedure) is assembled we can discharge the accumulated assumptions by providing distinct names and a value type that is big enough to hold all the types occurring in the program (e.g. a suitable sum type). In analogy to a compiler, reasoning first proceeds in an abstract, symbolic state space; then we "link" the whole program by assigning concrete names. The resulting theorems no longer mention any hypotheses about the state space construction.

4.1 Proof Contexts and Locales

Every proof in Isabelle depends on a background data structure called *proof context*, which was originally introduced by Wenzel [24,25] to support structured proof texts written in the Isar language. In a sense, a proof context is just "abstract nonsense" that helps to organise formal reasoning; a context may hold arbitrary *data* that can be declared at compile-time in a type-safe manner.

The original model for this notion of context stems from certain aspects of the underlying logical calculus of Isabelle (and the HOL family in general): here the main judgement $\Gamma \vdash B$ means that proposition B is derivable within an environment $\Gamma = \alpha_1, \ldots, \alpha_l, x_1, \ldots, x_m, A_1, \ldots, A_n$ consisting of fixed type variables, term variables, and hypotheses. Isabelle provides explicit notation to establish theorems within a local context, for example:

lemma fixes $x{::}\alpha$ **assumes** a: $A\ x$ **shows** b: $B\ x$ $\langle proof \rangle$

The proof may refer to a fixed parameter $x{::}\alpha$ and local fact a: $A\ x$, while the final result is exported from that context as a rule b: $\bigwedge x.\ A\ x \Longrightarrow B\ x$. Note that types are usually left implicit, any type variable occurring in a statement is implicitly fixed according to schematic polymorphism. At the outer level, term parameters may be fixed automatically as well, i.e. the above **fixes** is optional.

Apart from such purely logical assumptions and conclusions, the context may also hold additional non-logical information (type constraints, concrete syntax, hints for proof tools etc.). Thus the content of a context may be understood as arbitrary data that is abstracted over logical entities (types, terms, theorems).

The *locale* mechanism [12] of Isabelle manages high-level composition of contexts, supporting incremental additions of conclusions later on. For example:

locale *loc* = **fixes** $x{::}\alpha$ **assumes** a: $A\ x$
lemma (**in** *loc*) b: $B\ x$ $\langle proof \rangle$
lemma (**in** *loc*) c: $C\ x$ $\langle proof \rangle$

The annotation "(**in** *loc*)" causes the context of locale *loc* to be reconstructed, such that its content is available during the proof; the local result is stored within that context for later use in further conclusions; a global version is exported to the toplevel as in the immediate version of **fixes**/**assumes**/**shows** above. Additional contextual hints may be given using *attributes* (written as postfix application), e.g. the following command declares rules to the simplifier:

declare (**in** *loc*) b [*simp*] **and** c [*simp*]

The *simp* declaration is essentially a function *thm* → *context* → *context* that adds a given theorem to the *simpset* maintained as context data.

Locale expressions [3] compose existing locales via merge and rename operations. Multiple inheritance between locales can be expressed here. A new locale definition may add assumptions to a locale expression. *Locale interpretation* [4] transfers results stemming from one locale into another context. Interpretation works between locales, within a proof body, or at the outer theory level.

A *morphism* formalises the idea of moving results between contexts, adapting logical dependencies accordingly. Hence a morphism φ may be represented as a tuple $(\varphi_{type}, \varphi_{term}, \varphi_{thm})$ of mappings on those formal categories. This is further abstract nonsense to organise logical reasoning systematically. E.g. an *export morphism* between contexts $\Gamma_1 \subseteq \Gamma_2$ imposes the difference of assumptions on resulting theorems, by discharging hypotheses and introducing $\Longrightarrow$ etc. This is how the above rules b and c are exported into the global theory environment. Another important special case is an *interpretation morphism*: given concrete types and terms for fixed variables, and theorems for hypotheses, the corresponding substitution operation transforms a result from an abstract theory into a concrete situation. Thus locale interpretation can be explained succinctly.

With the help of explicit morphisms, we can easily generalise the idea of declaring theorems to the context (cf. the *simp* attribute above) towards arbitrary data that may be re-interpreted in different situations. A *declaration* is any function of type *morphism* → *context* → *context* that augments a context in a monotonic fashion. Declarations may be added to a locale using the command **declaration** (**in** *loc*). The locale infrastructure maintains a canonical order of declarations $d_1, \ldots, d_n$. Whenever the locale context is re-entered in a situation described by a morphism φ, the context is augmented to become the collective declaration $d_n\ \varphi\ (\ldots(d_1\ \varphi\ \Gamma)\ldots)$. This means that every time a locale context is reconstructed, all the data will be back in its proper place, as the effect of invoking the collection of declarations. Here the morphism tells how to interpret abstract concepts in the present situation. This facility can be used in numerous ways, such as maintaining information about state space field names and types.

4.2 Abstract State Spaces as Locales

Isabelle allows to add new top-level commands to the system. Building on the locale infrastructure, we provide a command **statespace**[6] like this:

statespace *vars* =
  n::*nat*
  b::*bool*

This resembles a **record** definition (§3.3), but introduces sophisticated locale infrastructure instead of HOL type schemes. The resulting context postulates two distinct names n and b and projection / injection functions that convert from abstract values to *nat* and *bool*. The logical content of the locale is:

locale *vars'* =
  fixes n::*'name* **and** b::*'name*
  assumes *distinct* $[n,\ b]$
  fixes *project-nat*::*'value* $\Rightarrow$ *nat* **and** *inject-nat*::*nat* $\Rightarrow$ *'value*
  assumes $\bigwedge n.$ *project-nat* (*inject-nat* n) $= n$
  fixes *project-bool*::*'value* $\Rightarrow$ *bool* **and** *inject-bool*::*bool* $\Rightarrow$ *'value*
  assumes $\bigwedge b.$ *project-bool* (*inject-bool* b) $= b$

The HOL predicate *distinct* describes distinctness of all names in the context. Locale *vars'* defines the raw logical content that is defined in the state space locale. We also maintain non-logical context information to support the user:

- Syntax for state lookup and updates that automatically inserts the corresponding projection and injection functions.
- Setup for the proof tools that exploit the distinctness information and the cancellation of projections and injections in deductions and simplifications.

This extra-logical information is added to the locale in form of declarations, which associate the name of a variable to the corresponding projection and injection functions to handle the syntax transformations, and a link from the variable name to the corresponding distinctness theorem. As state spaces are merged or extended there are multiple distinctness theorems in the context. Our declarations take care that the link always points to the strongest distinctness assumption. With these declarations in place, a lookup can be written as $s{\cdot}n$, which is translated to *project-nat* $(s\ n)$, and an update as $s\langle n := 2\rangle$, which is translated to $s(n :=$ *inject-nat* $2)$. We can now establish the following lemma:

[6] This is part of the Isabelle distribution since Isabelle2008; the subsequent examples use Isabelle2009.

lemma (**in** *vars*) *foo*: $s\langle n := 2\rangle{\cdot}b = s{\cdot}b$ **by** *simp*

Here the simplifier was able to refer to distinctness of b and n to solve the equation. The resulting lemma is also recorded in locale *vars* for later use and is automatically propagated to all its interpretations. Here is another example:

statespace α *varsX* = *vars* [$n{=}N$, $b{=}B$] + *vars* + $x{::}\alpha$

The state space *varsX* imports two copies of the state space *vars*, where one has the variables renamed to upper-case letters, and adds another variable x of type α. This type is fixed inside the state space but may get instantiated later on, analogous to type parameters of an ML-functor. The distinctness assumption is now *distinct* [N, B, n, b, x], from this we can derive both *distinct* [N, B] and *distinct* [n, b], the distinction assumptions for the two versions of locale *vars* above. Moreover we have all necessary projection and injection assumptions available. These assumptions together allow us to establish state space *varsX* as an interpretation of both instances of locale *vars*. Hence we inherit both variants of theorem *foo*: $s\langle N := 2\rangle{\cdot}B = s{\cdot}B$ as well as $s\langle n := 2\rangle{\cdot}b = s{\cdot}b$. These are immediate consequences of performing the locale interpretation.

The declarations for syntax and the distinctness theorems also observe the morphisms generated by the locale package due to the renaming $n = N$:

lemma (**in** *varsX*) *foo*: $s\langle N := 2\rangle{\cdot}x = s{\cdot}x$ **by** *simp*

To assure scalability towards many distinct names, the distinctness predicate is refined to operate on balanced trees. Thus we get logarithmic certificates for the distinctness of two names by the distinctness of the paths in the tree. Asked for the distinctness of two names, our tool produces the paths of the variables in the tree (this is implemented in SML, outside the logic) and returns a certificate corresponding to the different paths. Merging state spaces requires to prove that the combined distinctness assumption implies the distinctness assumptions of the components. Such a proof is of the order $m \cdot \log n$, where n and m are the number of nodes in the larger and smaller tree, respectively.

4.3 Integration into the Hoare Logic Environment

We now examine the integration of the new state space implementation into the Hoare logic environment [20] and see how it benefits from it.[7] The underlying programming language model is generic wrt. the state space representation. In the current implementation the verification is partitioned on the granularity of procedures. To properly handle procedure calls the framework represents the state as a polymorphic pair where one component stores the local variables and the other the global ones (including the heap). On return of a procedure call the global variables of the callee are passed back to the caller, whereas the local variables of the caller are restored. With this mechanism the framework handles the scoping correctly without depending on any further details of the concrete representation for local and global variables. Previously, records were used for state spaces. Although the scoping is already handled by the Hoare logic, the usage of records to represent local variables has the odd effect that the local variables of different procedures appear side-by-side in the record, blowing up its size. It is however possible to share local variables of the same name and the same HOL type. Besides this inconvenience of local variables, the Hoare logic inherits the same advantages and disadvantages from the underlying record representation as discussed before (§3.3). Now records are replaced by state spaces, which we explain by the example of list reversal.

The heap may hold structured values (e.g. `struct` in C). Our heap model follows Bornat [5]: instead of a single heap of structured components there is a separate heap for each field. Type *ref* is an abstract type for references. A structure to represent a linked list in the heap is `struct list {struct list *next;}`. The structure contains only the component `next`. So we get one heap variable *next* of type $ref \Rightarrow ref$ in the global state space:

statespace *globals-list* =
  $next::ref \Rightarrow ref$
procedures (**imports** *globals-list*) $Rev(p::ref|q::ref)$
where $r::ref$
in q := *Null*; **WHILE** p $\neq$ *Null* **DO** r := p; p := p→next; r→next := q; q := r **OD**

The **procedures** command defines the new procedure *Rev* importing the global state space. The input parameter *p*, the output parameter *q* as well as

[7] Available from the Archive of Formal Proofs: http://afp.sourceforge.net/entries/Simpl.shtml.

the local variable r together define a separate local state space (which does not have to be merged with the global state space, as the Hoare logic already distinguishes local and global variables). The syntax for statements and assertions builds on the infrastructure of the state spaces so that the user is not bothered with projections and injections into the value type. The notation $x{\rightarrow}f$ mimics composition of dereferencing a pointer with field selection in C and is translated to function lookup or update, depending on its occurrence as value or *left*-value.

Whenever we refer to a variable of the state space it is typeset in a $\mathsf{sans\text{-}serif}$ font. So p expands to $s{\cdot}p$ for some bound state s that is introduced by the surrounding syntax. For example, the loop condition above formally is a state set. The expanded version reads like $\{s.\ s{\cdot}p \neq \mathit{Null}\}$ in Isabelle's set-comprehension notation combined with the lookup syntax of state spaces. Similarly the assertions in the following lemma are translated to Isabelle's set-comprehension.

The **procedures** command combines the state space locales in a new locale named *Rev-impl*. To be able to rename procedures, e.g. to use several instantiations simultaneously, this locale also contains the procedure name *Rev* as a parameter and the assumption that in the procedure environment Γ (function from procedure names to bodies) at the position *Rev* the corresponding body is found. Within this locale we can prove the following lemma for list reversal:

lemma (**in** *Rev-impl*) **shows**
$\mathit{Rev\text{-}spec}$: $\forall\, \mathit{Ps}.\ \Gamma\vdash\ \{\!|\mathit{List}\ \mathsf{p}\ \mathsf{next}\ \mathit{Ps}|\!\}\ \mathsf{q} := \textsf{\textbf{PROC}}\ \mathit{Rev}(\mathsf{p})\ \{\!|\mathit{List}\ \mathsf{q}\ \mathsf{next}\ (\mathit{rev}\ \mathit{Ps})|\!\}$

This specification of procedures on heap lists follows Mehta and Nipkow [14]. From the pointer structure in the heap we (relationally) abstract to HOL lists of references. The predicate *List p next Ps* expresses that we obtain the (HOL) list of references *Ps* by starting at reference *p* and following the *next* heap.

This specification of list reversal is quite abstract and conceptually works for any structure that contains some kind of "next" pointer. Consider, for example a program that implements strings and queues as linked lists.

```
struct string {              struct queue {
  char chr;                    int cont;
  struct string *strnext;}; struct queue *qnext;};
```

We can define this extended state space by importing two copies of the list state and adding the new components:

statespace *globals-compose* =
  globals-list [*next*=*strnext*] + *globals-list* [*next*=*qnext*] +
  chr :: *ref* ⇒ *char*
  cont:: *ref* ⇒ *int*

We can now use ordinary locale operations to merge and rename locales to create two instances of the list reversal procedure. One for strings, named *RevS* and one for queues named *RevQ*. The only thing we have to do is to rename the *next* component and the procedure name accordingly. This can be done with the following locale operations (new parameters are listed after **for**):

locale *RevS-impl* = *globals-compose* +
  Rev-impl **where** *next* = *strnext* **and** *Rev* = *RevS* **for** *RevS*
locale *RevQ-impl* = *globals-compose* +
  Rev-impl **where** *next* = *qnext* **and** *Rev* = *RevQ* **for** *RevQ*

The aforementioned **procedures** command ensures, that any procedure using one of the procedures *RevS* or *RevQ* imports the locales *RevS-impl* or *RevQ-impl*, respectively. Within this setup both instances of the procedure specification are immediately available for the further program verification. E.g.

$\forall Ps.\ \Gamma\vdash \{\!|\mathit{List}\ \mathsf{p}\ \mathsf{strnext}\ Ps|\!\}\ \mathsf{q} := \textbf{PROC}\ \mathit{RevS}(\mathsf{p})\ \{\!|\mathit{List}\ \mathsf{q}\ \mathsf{strnext}\ (\mathit{rev}\ Ps)|\!\}$

Since we may use Isabelle's type variables or type classes to specify abstract program variables we can also develop abstract procedures, like a generic sorting algorithm that can be instantiated later. Regardless of whether the underlying model of the programming language supports features like generics, we can employ the Isabelle/HOL infrastructure to reason abstractly about these procedures and specialise them to the different instances.

5 Conclusion

Our approach of representing state spaces for imperative programs is a combination of basic logical concepts with an extra-logical layer for type-checking and notation. The latter is based on existing locale infrastructure in Isabelle, which happily supports arbitrary declarations in proof contexts (such as program variables with their types). This careful arrangement in different layers allows to return to a simple logical model of states spaces as functions (as already seen in early experiments in HOL and in informal mathematics).

Strong typing for state spaces is essentially achieved by coercions (the projections and injections from an abstract value type) that are inserted automatically by our syntax layer. We did not need to consider the more complex notion of dependent function types, which are beyond HOL anyway. Instead, the Isabelle infrastructure is able to support a kind of *user space type system* outside the logic.

Logical simplicity is an important prerequisite for scalability and modularity: our motivation stems from non-trivial specification and verification tasks in the Verisoft project (C compiler, OS components, email client etc.). On the other hand, there is extra complexity in the design and implementation of the overall verification environment.

Here Isabelle locales have shown a great potential to model advanced concepts on top of the existing framework (and HOL object-logic). This flexibility is not accidental, but a consequence of the very design of Isabelle: foundations are frugal, but there are powerful mechanisms to implement add-on tools as user libraries. While the implementation of the latter is not trivial, it can be done with reasonable effort by experienced users, as has been demonstrated here.

References

[1] Andersen, F., K. D. Petersen and J. S. Pettersson, *Program verification using HOL-UNITY*, in: J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications: HUG '93*, LNCS 780 (1993).

[2] Back, R.-J. and J. von Wright, "Refinement Calculus: A Systematic Introduction," Springer-Verlag, 1998, graduate Texts in Computer Science.

[3] Ballarin, C., *Locales and locale expressions in Isabelle/Isar*, in: S. Berardi et al., editors, *Types for Proofs and Programs (TYPES 2003)*, LNCS 3085 (2004).

[4] Ballarin, C., *Interpretation of locales in Isabelle: Theories and proof contexts*, in: J. M. Borwein and W. M. Farmer, editors, *Mathematical Knowledge Management (MKM 2006)*, LNAI 4108 (2006).

[5] Bornat, R., *Proving pointer programs in Hoare Logic*, in: R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, LNCS 1837 (2000).

[6] Brucker, A. D. and B. Wolff, *Extensible universes for object-oriented data models*, in: J. Vitek, editor, *Proceedings of the European Conference of Object-Oriented Programming (ECOOP 2008)*, LNCS 5142, Springer-Verlag, Paphos, Cyprus, 2008 pp. 438–462.

[7] Ehmety, S. O. and L. C. Paulson, *Mechanizing compositional reasoning for concurrent systems: Some lessons*, Formal Aspects of Computing **17** (2005), pp. 58–68.

[8] Gordon, M., *Mechanizing programming logics in higher order logic*, in: G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, 1989.

[9] Harrison, J., *Formalizing Dijkstra*, in: J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs'98)*, LNCS 1497 (1998).

[10] Heyd, B. and P. Crégut, *A modular coding of UNITY in COQ*, in: *Theorem Proving in Higher Order Logics (TPHOLs'96)*, LNCS 1125 (1996).

[11] Homeier, P. V., "Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures," Ph.D. thesis, Dept. of Computer Science, University of California, Los Angeles (1995).

[12] Kammüller, F., M. Wenzel and L. C. Paulson, *Locales: A sectioning concept for Isabelle*, in: Y. Bertot et al., editors, *Theorem Proving in Higher Order Logics (TPHOLs'99)*, LNCS 2741 (1999).

[13] Lamport, L. and L. C. Paulson, *Should your specification language be typed?*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 502–526.

[14] Mehta, F. and T. Nipkow, *Proving pointer programs in higher-order logic*, Information and Computation **199** (2005), pp. 200–227.

[15] Merz, S., *Yet another encoding of TLA in Isabelle* (1999).
URL http://www.loria.fr/~merz/projects/isabelle-tla/doc/design.ps.gz

[16] Naraschewski, W. and M. Wenzel, *Object-oriented verification based on record subtyping in higher-order logic*, in: *Theorem Proving in Higher Order Logics (TPHOLs'98)*, LNCS 1479 (1998).

[17] Nipkow, T., L. Paulson and M. Wenzel, "Isabelle/HOL — A Proof Assistant for Higher-Order Logic," LNCS 2283, Springer, 2002.

[18] Paulson, L. C., *Mechanizing a theory of program composition for UNITY*, ACM Transactions on Programming Languages and Systems **25** (2001), pp. 626–656.

[19] Prensa Nieto, L., "Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL," Ph.D. thesis, TU München (2002).

[20] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, TU München (2006).

[21] Staples, M., "A Mechanised Theory of Refinement," Ph.D. thesis, The University of Cambridge Computer Laboratory (1998).

[22] von Wright, J., J. Hekanaho, P. Luostarinen and T. Långbacka, *Mechanizing some advanced refinement concepts*, Formal Methods in System Design **3** (1993), pp. 49–81.

[23] Wenzel, M., *Miscellaneous Isabelle/Isar examples for higher-order logic. Part of the Isabelle distribution* (2001).

[24] Wenzel, M., "Isabelle/Isar — a versatile environment for human-readable formal proof documents," Ph.D. thesis, TU München (2002).

[25] Wenzel, M. and B. Wolff, *Building formal method tools in the Isabelle/Isar framework*, in: K. Schneider and J. Brandt, editors, *TPHOLs 2007*, LNCS 4732, Springer-Verlag, 2007 pp. 351–366.