Representing Component States in Higher-Order Logic

Sidi O. Ehmety and Lawrence C. Paulson

Computer Laboratory
University of Cambridge
Cambridge CB2 3QG, England
{soe20,lcp}@cl.cam.ac.uk

Abstract. Component states can be formalized in higher-order logic as (1) functions from variables to values and (2) records, among other possibilities. Variable-to-value maps are natural, but they yield weak typing and restrict the user to a predefined value space. Record types define component signatures and properties need to be transferred between the various signatures. The method yields strong typing, but transferring properties requires an elaborate theory and not all properties can be transferred.

The paper reports experiments with a third method: the state is represented by an abstract type. The method is described and contrasted with respect to the others.

1 Introduction

Compositional reasoning is becoming increasingly popular. The idea of deriving a system's properties from those of its components is attractive.

This paper concerns compositional models of concurrent systems within the programming formalism UNITY [8, 9]. These theories have been supported by various tools. Heyd and Crégut [6], Vos [12], and Paulson [11] implement UNITY using the proof tools HOL, Coq and Isabelle/HOL, respectively. The resulting systems include all relevant definitions and basic laws of compositional reasoning as well as some illustrative examples.

A common point to these mechanizations is that most of the effort has been put on codifying the theory itself. Others aspects, such as usability and applicability, have been secondary considerations.

In their mechanization, Heyd and Crégut are mainly concerned by the application of the so-called *substitution axiom*, which is unsound in compositional proofs. For this purpose they introduce a notion of *context.* They obtain strong versions of UNITY properties upon which the axiom can safely be applied. However, while in early days the axiom was considered as an obstacle to compositional reasoning, today many proofs are being done without it: that the axiom can simply be avoided.

In her mechanization, Vos is more aware of the end user. She defines a whole new syntax in order to hide implementation details and to ease the use of the resulting tool for the end users. However her representation of states is potentially restrictive.

Our work is a continuation of Paulson's mechanization which implements the recent Chandy and Sanders compositional theory [1, 2] on the top of UNITY. In his mechanization, Paulson leaves the state unspecified: the type of components is polymorphic in the state `'a program`. The choice of representing states is delayed until component specifications and is left to the end user.

Using Isabelle/UNITY, we [5] have successfully mechanized two examples [3] illustrating universal and existential composition. A *universal property* is one that holds in a system provided all components have the property. An *existential property* is one that holds in a system provided some components have the property.

2 Representing Component States

The state of a program is an assignment from values to the program's variables.

Most UNITY proofs are done by hand and this notion of state, among many other things, is implicit. However there are two main common considerations that most authors take into account:

- Strong typing: each variable has a declared type and is only assigned values of that type.
- Uniformity of state space: when composing two programs F and G the state type is common to both components, as well as their composition, $F \sqcup G$. In others words the state type is considered to be common to all components.

In addition, variables names have global scope. Furthermore, some authors quantify over variables, as in following specification for a component with two variables x and y:

$$\text{For all variables } v \text{ other than } x \text{ and } y,\ \forall k.\, \texttt{stable}\,(v = k)$$

which expresses that component does not modify variables other than its own. A similar quantification can express that a variable is local. Quantification over variables can concisely express specifications where many variables remain unchanged. Note that there is a clear distinction between variables and their values.

Is there a representation of states that combines strong typing and uniformity, while allowing quantification over variables? There is little literature on state representation in higher-order logic. In her PhD thesis [12], Vos discusses two different representations: tuples and variable-to-value maps. Vos rightly dismisses tuples because they identify variables by their position in the tuple rather than by name. Most authors represent the state as a function from variables to values. As Paulson has noted [10], this representation is uniform but restricts the user to the built-in type of values. If the value type is a recursive disjoint sum, then its constructor and destructor functions will obscure specifications. Vos gets around that problem by defining a concrete syntax for expressions built over the disjoint sum. A variable-to-value map gives all variables the same type, yielding a weakly typed formalism; well-typing has to be expressed as an invariant that must be proved and used explicitly.

Paulson [10] suggested using record types to define component signatures. It permits a variety of state types rather than requiring a uniform one. It provides a means of transferring properties from one type to another. The mechanization of a large example proposed by Chandy and Mandy [4] has shown that unless the state type is common to all components, the need to reconcile the differences between the various types can cause grave difficulties.

Heyd and Crégut, in their mechanization using the Coq system, represent a state by a dependent function from variables to types. The type of variables is defined by enumeration and the type of states can map each variable to a distinct type. The representation preserves strong typing but not the uniformity of state space. Furthermore, there is a need for all logical operations to be lifted into predicate states.

A last method for representing states is used by some authors in the so-called coalgebraic formalization of object-oriented classes [7]. In this method the state is modelled as a black-box. Inspectors (or methods) are provided to obtain information about the state. Inspectors also allow to change the state. As far as we know this method has never been investigated by the UNITY community.

Next we describe this approach in Isabelle/HOL.

3 State as abstract type

This method has first been used by Merz in his mechanization of the TLA formalism in Isabelle/HOL. However it has never been documented. So by this description we hope to clarify some characteristics of this interesting method as well as to provoke a discussion about it in the HOL community.

Now let us describe the method. Assume an abstract type `state`:

```
typedecl state
```

Here command **typedecl** introduces a new type but without defining it. Thus we know nothing about the type other than it exists.

Now variables can be defined as inspector functions over the type `state`. One merely has to define as many inspector functions as necessary. For example below we declare two state inspectors, namely `x` and `y`:

```
consts x :: state ⇒ nat
       y :: state ⇒ int
```

The inspector `x` takes its values over the natural numbers. It specifies a variable of type `nat`. The inspector `y` defines a variable of type `int`.

Clearly in this way strong typing is preserved: the result types of variables are precisely specified. Uniformity is also preserved: all specifications share the global type `state`.

However we still need an axiom in order to be able to prove the enabledness of component actions. Enabledness is essential in order to prove that a property is transient, which in turn is needed to prove liveness.

For illustration, let us consider a component with just one variable, namely the variable `x` previously declared. The component has only one action, `act`, given by the following relation:

$$\begin{array}{ll}\textbf{constdefs}\ \texttt{act}\ ::\ \texttt{state}\times\texttt{state} \\ \qquad\quad\ \texttt{act}\ \equiv\ \{(s,s')\,|\,\mathtt{x}(s')=\mathtt{x}(s)+1\}\end{array}$$

Command **constdefs** defines the constant `act` to be the binary relation over `state` given (by comprehension) as a set of pairs of states. The $\equiv$ notation means equality by definition. Thus `act` specifies the assignment $x := x + 1$. Clearly this action is always enabled, but to prove this claim one has to show that

$$\exists\, s'.\ \mathtt{x}(s') = \mathtt{x}(s) + 1,$$

for any state s.

More generally, to prove the enabledness of the action `act` we introduce the following axiom: for any valid value for the variable `x` there exists a state s that assigns that value to `x`. Formally,

$$\forall\, n.\ \exists\, s.\ \mathtt{x}(s) = n$$

Clearly the axiom is sound. HOL strong typing indicates that n is a valid value for `x`, that is, n is a natural number. Now we can prove the enabledness of the action `act` by applying the axiom taking $n = \mathtt{x}(s) + 1$.

The axiom is equivalent to

$$\mathtt{range(x)} = \mathtt{UNIV}. \tag{1}$$

Here `UNIV` is a polymorphic constant denoting the universal set in Isabelle/HOL.

Extending axiomatization (1) to many variables is a bit tricky. In his mechanization of TLA, Merz introduces three axioms for this purpose. Discussions with Merz led to the simpler definition, using the *pair function*:

$$\mathtt{basevars}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n) \ \equiv\ \mathtt{range}(\mathtt{x}_1 \oplus \mathtt{x}_2 \oplus \ldots \oplus \mathtt{x}_n) = \mathtt{UNIV} \tag{2}$$

Here $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$, $n > 0$, are state functions (variables). The $\oplus$ symbol is used as an infix notation for the pair function: $\mathtt{x} \oplus \mathtt{y} \equiv \lambda s.(\mathtt{x}(s), \mathtt{y}(s))$.

Definition (2) generalizes axiom (1) to tuples. Thus one merely has to declare variables of a component using `basevars`.

Intuitively, a `basevars` declaration asserts that for any valid values $v_1, \ldots, v_n$ for the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$, there exists a state that assigns n_1 to $\mathtt{x}_1$, …, and

v_n to $\mathtt{x}_n$ if and only if there exists a state that assigns the tuple $(v_1, \ldots, v_n)$ to the pair function $\mathtt{x}_1 \oplus \ldots \oplus \mathtt{x}_n$:

$$\begin{array}{l}\forall v_1, \ldots, v_n. \\ (\exists\, s.\ \mathtt{x}_1(s) = v_1 \wedge \ldots \wedge \mathtt{x}_\mathtt{n}(s) = v_n) \leftrightarrow (\exists\, s.\ (\mathtt{x}_1 \oplus \ldots \oplus \mathtt{x}_n)(s) = (v_1, \ldots, v_n))\end{array}$$

The $\rightarrow$ implication is trivial. The soundness of the $\leftarrow$ implication expresses an essential property of variables: $\mathtt{x}_1, \ldots, \mathtt{x}_n$ must be independent views of the component states. There should be no dependence between them. For illustration let us consider a trivial case of dependence: suppose that the user mistakenly asserts the declaration $\mathtt{basevars}(\mathtt{x}, \mathtt{x})$. Thus we can prove that there exists a state in which the pair function $\mathtt{x} \oplus \mathtt{x}$ has the value $(0, 1)$. However there can never be a state s that assigns simultaneously 0 and 1 to $\mathtt{x}$, since in such a state we would have $\mathtt{x}(s) = 0 \wedge \mathtt{x}(s) = 1$, which would lead to the equality $0 = 1$!

Furthermore, a `basevars` declaration should list just the variables of the component, no more, no less. Declaring more variables asserts an axiom that is stronger than necessary. For example declaring `basevars(x,y)` for a component with only one variable `x`, would require showing $\exists s.\mathtt{x}(s) = m \wedge \mathtt{y}(s) = n$ when it is only needed to prove $\exists s.\mathtt{x}(s) = m$. Declaring fewer variables asserts an axiom that is too weak to prove all possible enabledness properties.

4 An Example: the Counter System

Let us now illustrate this method with a simple example taken from Charpentier and Chandy [3]. Consider an I-indexed family of components sharing a global variable: a counter C. Each component i also has a local counter c_i and performs a certain action a. Components increase their local counters and the global counter by one each time they perform the action. Clearly, each c_i records the number of actions performed by component i and C always equals the sum of the c_i.

In other work [5], we have described the mechanization of this example. Here we will only review the relevant part to our illustration.

The formal specification of component i [3] consists of three safety properties:

$$\mathtt{initially}\ c_i = 0 \wedge C = 0 \tag{3}$$

$$\forall k.\, \mathtt{stable}\ C = c_i + k \tag{4}$$

$$\forall k.\, \mathtt{stable}\,(v = k) \quad \text{for all variables } v \text{, other than } c_i \text{ and } C \tag{5}$$

Additionally, variable c_i is declared to be local to component i:

$$\mathtt{local}\, c_i$$

Property (3) fixes the initial values of both C and c_i at zero. Property (4) means that component i always increases C and c_i by the same value. Property (5) states that component i changes no variables other than c_i and C. This property ensures that components are compatible: the components can safely be composed because they respect each others' local variables.

The variable declaration is formalized in two steps. First, we declare the variables' types:

```
consts  C :: state ⇒ int
        c :: nat ⇒ state ⇒ int
```

The family of variables $\{c_i\}_{i \in I}$ could be formalized as `c` :: `state` ⇒ `nat` ⇒ `int` instead.

Second, we introduce the `basevars` assertion:

```
rules   basevars(c(i), C)
```

Command **rules** adds the assertion as an axiom. Thus we assert that `c` i and `C` are the only variables of the component i and also that `c` i and `C` are independent views of the state.

In this example, the `basevars` assertion is not required because we are not going to prove transient properties. But for methodology, we always introduce it as part of a variable declaration. We can even extend the Isabelle/HOL theory syntax to accept variable declarations for UNITY programs. This new theory section could check that the listed variables are independent views of the state.

Having declared the component's variables, we are now able to formalize its properties. The Isabelle versions of properties (3) and (4) for a component i are given below:

$$\texttt{component}\, i \in \texttt{initially}\, \{s \mid (\texttt{c}\, i)(s) = 0 \wedge \texttt{C}(s) = 0\}$$

$$\texttt{component}\, i \in \texttt{stable}\, \{s \mid \texttt{C}(s) = (\texttt{c}\, i)(s) + k\}$$

Note that state predicates are represented as sets of states. A program property is represented as the set of programs satisfying that property.

Property (5), however, can not be expressed as it appears above. The property uses universal quantification over variables. We do not have a type of all variables and therefore cannot express a such quantification.

As we show in previous work [5], this axiom, which expresses auxiliary conditions over variables, is stronger than necessary. (It does not matter which state representation is used.) Instead of that axiom, we propose the following general one, for any program F:

$$F\, \texttt{ok}\, (\texttt{component}\, i) \rightarrow F \in \texttt{stable}\{s \mid (\texttt{c}\, i)(s) = k\}$$

Here, F `ok` G means that components F and G are *compatible*, which among other things implies they respect each others' local variables. This specification is more abstract than the original one. A drawback is that proofs become cluttered by compatibility assumptions. We would greatly prefer to be able to quantify over variables.

Had we instead represented states by variable-to-value maps, we could have defined the action `act` of component i using function update:

$$\textbf{constdefs}\ \ \mathtt{act}\,i\ \equiv\ \{(s,s') \mid s' = s(\mathtt{c}\,i := s(\mathtt{c}\,i)+1, \mathtt{C} := s(\mathtt{C})+1)\}.$$

Here variables $\mathtt{c}\,i$ and $\mathtt{C}$ would be not inspector functions but simply names, declared for example as a disjoint sum:

$$\textbf{datatype}\ \mathtt{name}\ =\ \mathtt{c}(\mathtt{nat}) \mid \mathtt{C}.$$

And states would be functions from `name` to the values space (here the type `int`). To inspect the value of a variable, function application is used but the other way around. For example, $s(\mathtt{C})$ returns the value of the variable $\mathtt{C}$ in map s. The notation $s(\mathtt{c}\,i := s(\mathtt{c}\,i)+1, \mathtt{C} := s(\mathtt{C})+1)$ specifies the map obtained from s by simultaneously changing the values of variables $\mathtt{c}\,i$ and $\mathtt{C}$ to be one more their previous values. From this definition of $\mathtt{act}\,i$, we can prove the required properties. Had we instead represented states by records, we could have proceeded similarly, using record updates.

The lack of an update notation (whether for records or maps) is a major drawback of the abstract state type approach. Consider a program with three variables, `x`, `y` and `z`. Suppose we want to express an action that only updates the variable `x`. If we have an update notation, then the final state is simply $s(\mathtt{x} := \mathtt{v})$. With abstract state method, we must write

$$\mathtt{x}(s') = \mathtt{v} \wedge \mathtt{y}(s') = \mathtt{y}(s) \wedge \mathtt{z}(s') = \mathtt{z}(s)$$

In other words, we have to define the values of all variables in the final state, including those that remain unchanged. With even ten variables, the notation becomes intolerable. This drawback would be avoided if we could express universal quantification, for we could then say that all variables other than `x` retain their original values.

5 Conclusion

In this paper we have discussed several methods for representing component states. We report experiments with a new representation, with the state as an abstract type. It has two advantages: uniformity and strong typing. Had this method permitted defining states by updating a previous state, it would certainly be the best of all. We do not see a way to do so.

In absence of this feature, variable-to-value maps seem worth reconsidering. To avoid the drawbacks of disjoint sum, we have started to mechanize UNITY in ZF set theory. ZF's formalization of the cumulative hierarchy (sets of the form V_α) provides huge value spaces. New types can be added by extension. Constructor and destructor functions are avoided. The main drawback of this approach will be weak typing; type-checking is done using invariants.

Acknowledgements

Thanks to the U.K.'s Engineering and Physical Sciences Research Council, who funded this research: grant GR/M 75440, 'Compositional Proofs of Concurrent Programs'.

References

1. K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24:129–148, 1995.
2. K. Mani Chandy and Beverly A. Sanders. Reasoning about program composition. Technical Report 2000-003, CISE, University of Florida, 2000. available via `http://www.cise.ufl.edu/~sanders/pubs/composition.ps`.
3. Michel Charpentier and K. Mani Chandy. Examples of program composition illustrating the use of universal properties. In José Rolim, editor, *Parallel and Distributed Processing*, LNCS 1586, pages 1215–1227, 1999.
4. Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, LNCS 1708, pages 570–589. Springer, 1999.
5. S. Ehmety and L. Paulson. Program composition in Isabelle/UNITY. 2001. submitted for publication.
6. Barbara Heyd and Pierre Crégut. A modular coding of UNITY in COQ. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: TPHOLs '96*, LNCS 1125, pages 251–266. Springer, 1996.
7. Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
8. Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995. Also at URL `ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/progress.ps.Z`.
9. Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995. Also at URL `ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/safety.ps.Z`.
10. Lawrence C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Computational Logic*. in press.
11. Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1):3–32, 2000.
12. Tanja E. J. Vos. *UNITY in Diversity, a Stratified Approach to the Verification of Distributed Algorithms*. PhD thesis, Utrecht University, 1999.