Data Refinement for Verified Model-Checking Algorithms in Isabelle/HOL

Peter Lammich

Theorem Proving Group, Institut für Informatik, TU-München
lammich@in.tum.de

Extended Abstract

Our goal is to verify model-checking algorithms with Isabelle/HOL. When regarding such algorithms on an abstract level, they often use nondeterminism like "take an element from this set". Which element is actually taken depends on the concrete implementation of the set. When formalizing these algorithms, one has to either fix the concrete implementation for the correctness proof, or describe the algorithm nondeterministically. The former approach makes it difficult to exchange the implementation afterwards. Moreover, using a set implementation (e.g. Red Black Trees) instead of the standard set datatype of Isabelle makes the use of automated reasoning tools more complicated, as they are tailored to Isabelle's standard types.

In this extended abstract, we briefly present our current effort to address the latter approach. We describe a framework that allows to specify nondeterministic algorithms on Isabelle's standard datatypes, prove them correct, and then refine them to executable algorithms. Our framework smoothly integrates with existing Isabelle/HOL specifications, is powerful enough to express model-checking algorithms, and automates tedious but canonical tasks.

Overview

Our framework is based on program and data refinement (cf. [1]). The possible results of a program are described by sets. Additionally, we add the result $\top$ that represents a failed assertion. Lifting the subset ordering, we get a complete lattice of results, where $\emptyset$ is the smallest element, and $\top$ is the greatest element. We write $\sqsubseteq$ for the lifted subset ordering.

Programs itself are described by a nondeterminism monad [9]. We define the operations *return* and *bind* as follows:

$$\mathsf{return}\ x := \{x\}$$

$$\mathsf{bind}\ M\ f := \begin{cases} \top & \text{if } M = \top \\ \bigsqcup\{f\ x \mid x \in M\} & \text{otherwise} \end{cases}$$

Intuitively, the return operation builds a result that contains a single value, and the bind operation applies the function f to each result in M. In addition to the

return and bind operations, we define the following elementary operations:

$$\mathsf{spec}\ \Phi := \{x \mid \Phi\ x\}$$

$$\mathsf{assert}\ \Phi := \begin{cases} \mathsf{return}\ () & \text{if } \Phi \text{ holds} \\ \top & \text{otherwise} \end{cases}$$

$$\mathsf{assume}\ \Phi := \begin{cases} \mathsf{return}\ () & \text{if } \Phi \text{ holds} \\ \emptyset & \text{otherwise} \end{cases}$$

Intuitively, $\mathsf{spec}\ \Phi$ describes all results that satisfy the predicate Φ, $\mathsf{assert}\ \Phi$ fails if Φ does not hold, and $\mathsf{assume}\ \Phi$ returns the empty set of results if Φ does not hold. Note that the empty set of results is the least element w.r.t. $\sqsubseteq$, and thus trivially satisfies any specification. Dually, $\top$ is the greatest element, and thus satisfies no specification.

Functions defined using our monad are always monotonic. Hence, we can use the *partial function package* [6] of Isabelle/HOL to define recursive functions. A particular interesting recursive function is the while-combinator $\mathsf{while}\ b\ f\ s_0$, that models a loop. When using the partial function package to define recursion, we get a notion of partial correctness, as the result contains exactly the values that are reached by finite executions. As Isabelle/HOL's code generator only guarantees partial correctness, too, this is adequate in our setting.

In order to model data refinement, we use a relation R that relates concrete values to abstract values. We assume that R is single-valued, i.e., $(x, y) \in R \wedge (x, z) \in R \implies y = z$. We then define $\Downarrow R$ as a function that maps abstract results to concrete results. Then, $S_1 \sqsubseteq \Downarrow R\ S_2$ describes that the results of S_1 only contain valid concretizations of the results of S_2, i.e., S_1 is a valid refinement of S_2, w.r.t. R. We define $\Downarrow R$ to map the result $\top$ to $\top$ again. This allows us to refine assertions with the following rule:

$$\frac{\Phi \implies \Phi',\ S \sqsubseteq \Downarrow R\ S'}{\mathsf{assert}\ \Phi\ S \sqsubseteq \Downarrow R\ \mathsf{assert}\ \Phi'\ S'}$$

Note that this rule does not hold if we would have defined $\top$ as the universal set of all results, as the image of the universal set of abstract values under $\Downarrow R$ is, in general, not the universal set of concrete values: There may be concrete values that make no sense, e.g. data structures that does not satisfy there invariants.

In a typical program development, first an initial program S_1 is written, and it is shown that $P\ i \implies S_1\ i \sqsubseteq \mathsf{spec}\ (Q\ i)$, where i is the input, P the precondition, and Q the postcondition. The program S_1 typically contains the basic structure of the algorithm, but leaves some details underspecified, using spec-statements to only specify the possible results. Moreover, it uses Isabelle's standard data types rather than efficient data structures. For example, selection of an element from a set X would be encoded by $\mathsf{spec}\ (\lambda x.\ x \in X)$. In order to prove that S_1 matches its specification, our framework provides Hoare-rules for all its program constructs and a verification condition generator to automate the application of the rules. Loop invariants may be either annotated in the program

text, or the verification condition generator inserts a schematic variable for them, that may be instantiated while proving the generated verification conditions.

After S_1 has been proven correct, it will be refined towards an efficient implementation, yielding programs $S_2, \ldots, S_n$. In these refinement steps, all spec-statements have to be refined to actual operations, and the datatypes have to be refined to the ones used for implementation. In our simple example, we first may implement sets by distinct lists, i.e., we show

$$(X', X) \in R \implies \mathsf{spec}\ (\lambda x.\ x \in \mathsf{set}\ X') \sqsubseteq \mathsf{spec}\ (\lambda x.\ x \in X)$$

where R is the relation that maps a distinct list to the set of its elements. Then, we may implement the selection operation by taking the first element from a list, i.e., we show

$$\mathsf{return}\ (\mathsf{hd}\ X') \sqsubseteq \mathsf{spec}\ (\lambda x.\ x \in \mathsf{set}\ X')$$

For this program, the Isabelle/HOL code generator can generate code. Our framework supports this refinement steps by a set of rules to show data refinements that preserve the structure of the program. These rules decompose a refinement goal between two programs into refinement goals between the elementary statements (return,spec) of the program. Newly introduced refinement relations are left schematic, and need to be instantiated after decomposition. For example, the rule for bind is as follows:

$$\frac{M \sqsubseteq \Downarrow R_1\ M',\ \forall x\ x'.\ (x, x') \in R_1 \implies f\ x \sqsubseteq \Downarrow R_2\ f'\ x'}{\mathsf{bind}\ M\ f \sqsubseteq \Downarrow R_2\ \mathsf{bind}\ M'\ f'}$$

When this rule is applied, the new refinement relation R_1 becomes a schematic variable, that can be instantiated later.

The general form of a refinement step is

$$(i, i') \in R^I \implies S\ i \sqsubseteq \Downarrow R^O\ (S'\ i')$$

where R^I is the refinement relation on inputs, and R^O is the refinement relation on results (outputs). Note that refinement is transitive, i.e., we have

$$S \sqsubseteq \Downarrow R\ S' \wedge S' \sqsubseteq \Downarrow R'\ S'' \implies S \sqsubseteq \Downarrow (RR')\ S''.$$

Thus, after the process of refining S_1 has ended with program S_n, we have

$$(i, i') \in R^I \wedge P\ i' \implies S_n\ i \sqsubseteq \Downarrow R^O\ \mathsf{spec}\ (Q\ i'),$$

where R^I is the composition of all input refinements, and R_O is the composition of all output refinements. This precisely describes the correctness of the refined program w.r.t. the abstract specification.

Earlier Work

We encountered the problem of formalizing nondeterministic algorithms when we tried to formalize the predecessor set computation for DPNs[3]. There[1], we formalized a WHILE-loop as the iteration of a step-relation. However, we did not define a notion of nondeterministic programs. Thus, the loops had to be handled separately from the other parts of the algorithm, and the actual algorithm was not assembled until all the refinement steps had been done. Moreover, we could not express nested loops. In our formalization of tree-automata [7], we essentially used the same approach.

Current and Future Work

Our framework is still in a prototype development stage. So far, we have applied it to two examples: First, we formalized a simple state-space exploration algorithm, that takes as input a start state s_0, a transition relation δ, and a predicate P over states, and returns true if a state s that satisfies P is reachable, i.e., we implement the specification $\mathsf{spec}\ (\lambda x.\ x \iff \exists s.\ (s_0, s) \in \delta^* \wedge P\ s)$. The algorithm is a simple workset-algorithm, i.e., it has an initialization phase and a main loop, that iterates until the workset is empty or a state satisfying P has been found. Second, we formalized Dijkstra's shortest path algorithm in our framework[2]. Both algorithms are first formalized on an abstract level, and then refined towards an efficient implementation using the Isabelle Collection Framework [8] to provide efficient data structures. The refinement of the state space exploration algorithm is straightforward and done in a single refinement step from the abstract algorithm S_1 to the executable version. It could also have been done with parameterization. However, the refinement of Dijkstra's algorithm is done via an intermediate step, that introduces some redundant information to come closer to the data structures eventually used in the implementation. Combining this intermediate step with the implementation step is not feasible due to an explosion of proof complexity[3]. Due to this intermediate step, modeling nondeterminism with parameterization is not possible.

Future work includes the automation of refining programs. Currently, one has to explicitly specify the abstract and the refined program, and our framework tries to automate the proof that the refinement is, indeed, correct. However, there are several instances of simple refinements, in particular data refinements, that could be done automatically. That is, the user would only specify the abstract program and how used data structures shall be implemented, and the framework would define and prove correct the refined program automatically.

---

[1] Unpublished, available at http://cs.uni-muenster.de/sev/staff/lammich/isabelle/

[2] Based on an unpublished formalization of Nordhoff

[3] This was attempted in the original formalization of the algorithm. The proof obligations quickly became very large and confusing, such that this attempt was given up in favor of introducing the intermediate refinement step.

Related Work

Our framework is based on the notion of program refinement, that has been established by Back [1]. See [2] for an overview. However, we describe functional (monadic) programs, that are shallowly embedded into the logic of Isabelle/HOL, while, up to our knowledge, most work on program refinement is focused on imperative programs.

Our programs are based on a nondeterminism monad, that is inspired by the set monad in [9], and fits the requirements of Isabelle/HOL's partial function package [6] to define recursive functions.

In the context of Isabelle/HOL's code generation [4, 5], there is also work in progress to automatically replace inefficient data structures by efficient ones upon code generation. However, these approaches cannot handle nondeterministic operations.

Bibliography

[1] R.-J. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, bo Akademi, Department of Computer Science, Helsinki, Finland, 1978. Report A–1978–4.

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

[3] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.

[4] F. Haftmann. *Code Generation from Specifications in Higher Order Logic*. PhD thesis, Technische Universität München, 2009.

[5] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming (FLOPS 2010)*, LNCS. Springer, 2010.

[6] A. Krauss. Recursive definitions of monadic functions. In *Proc. of PAR 2010*, 2010.

[7] P. Lammich. Tree automata. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/entries/Tree-Automata.shtml, Dec. 2009. Formal proof development.

[8] P. Lammich and A. Lochbihler. The isabelle collections framework. In *Proc. of ITP 2010*, volume 6172 of *LNCS*, pages 339–354. Springer, July 2010.

[9] P. Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.