Sequential, Parallel, and Quantified Updates of First-Order Structures

Philipp Rümmern

Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden
philipp@cs.chalmers.se

Abstract. We present a datastructure for storing memory contents of imperative programs during symbolic execution—a technique frequently used for program verification and testing. The concept, called updates, can be integrated in dynamic logic as runtime infrastructure and models both stack and heap. Here, updates are systematically developed as an imperative programming language that provides the following constructs: assignments, guards, sequential composition and bounded as well as unbounded parallel composition. The language is equipped both with a denotational semantics and a correct rewriting system for execution, whereby the latter is a generalisation of the syntactic application of substitutions. The normalisation of updates is discussed. The complete theory of updates has been formalised using Isabelle/HOL.

1 Introduction

First-Order Dynamic Logic [1] is a program logic that enables to reason about the relation between the initial and final states of imperative programs. One way to build calculi for dynamic logic is to follow the symbolic execution paradigm and to execute programs (symbolically) in forward direction. This requires infrastructure for storing the memory contents of the program, for updating the contents when assignments occur and for accessing information whenever the program reads from memory. Sequent calculi for dynamic logic often represent memory using formulas and handle state changes by renaming variables and by relating pre- and post-states with equations. All information about the considered program states is stored in the side-formulas Γ, Δ of a sequent Γ ⊢ ⟨α⟩ φ, Δ, like in inequations 0 < x and equations x' = x + 1.

As an alternative, this paper presents a datastructure called Updates, which are a generalisation of substitutions designed for storing symbolic memory contents. When using updates, typical sequents during symbolic execution have the shape Γ ⊢ {u} ⟨α⟩ φ, Δ. The program α is preceded by an update u that determines parts of the program state, for instance the update x := x + 1. Compared with side-formulas, updates (i) attach information about the program state directly to the program, (ii) avoid the introduction of new symbols, (iii) can be simplified and avoid the storage of obsolete information, like of assignments that have been overridden by other assignments, (iv) represent accesses to variables,
array cells or instance attributes (in object-oriented languages) in a uniform way, (v) delay case-distinctions that can become necessary due to aliasing, (vi) can be eliminated mechanically once a program has been worked off completely.

Historically, updates have evolved over years as a central component of the KeY system [2], a system for deductive verification of Java programs. They are used both for interactive and automated verification. In the present paper, we define updates as a formal language (independently of particular program logics) and give them a denotational semantics based on model-theoretic semantics of first-order predicate logic. The language is proposed as an intermediate language to which sequential parts of more complicated languages (like Java) can stepwise be translated. The thesis [3] related to this paper gives a rewriting system that allows to execute or eliminate updates mechanically. The main contributions of the paper are new update constructs (in particular quantification), the development of a complete metatheory of updates and its formalisation using the Isabelle/HOL proof assistant [4], including proofs of all lemmas about updates that are given in the present paper or in [3].

The paper is organised as follows: Sect. 2 motivates updates through an example. Sect. 3 and 4 introduce syntax and semantics of a basic version of updates in the context of a minimalist first-order logic. Sect. 5 describes the rewriting system for executing updates. Sect. 6 adds an operator for sequential composition to the update language. Sect. 7 shows how heap structures can be modelled and modified using updates, which is applied in Sect. 8 about symbolic execution. Sect. 9 discusses laws for simplification of updates.

2 Updates for Symbolic Execution in Dynamic Logic

We give an example for symbolic execution using updates in dynamic logic. Notation and constructs used here are later introduced in detail. The program fragment max is written in a Java-like language and is executed in the context of a class/record List representing doubly-linked lists with attributes next, prev and val for the successor, predecessor and value of list nodes:

\[
\text{max} = \begin{cases} 
\text{if } (a.\text{val} < a.\text{next}.\text{val}) \ g = a.\text{next}.\text{val}; \ & \text{else } g = a.\text{val}; 
\end{cases}
\]

where \(a\) and \(g\) are program variables pointing to list nodes. The initial state of program execution is specified in an imperative way using an update:

\[
\text{init} = a.\text{prev} := \text{nil} \ | b.\text{next} := \text{nil} \ | a.\text{next} := b \ | b.\text{prev} := a \ | \ a.\text{val} := c \ | b.\text{val} := d
\]

\text{init} can be read as a program that is executing a number of assignments in parallel and that is setting up a list with nodes \(a\) and \(b\). In case \(a = b\)—which is possible because we do not specify the opposite—the two nodes will collapse to the single node of a cyclic list and will carry value \(d\): assignments that literally

\[\text{www.cs.chalmers.se/~philipp/updates.thy} \approx 3500 \text{ lines Isabelle/Isar code.}\]