Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 56 (2005) 363–401
Science of Computer Programming
www.elsevier.com/locate/scico

Rewriting of imperative programs into logical equations

Olivier Ponsini*, Carine Fédèle, Emmanuel Kounalis

Laboratoire I3S, UNSA, CNRS (UMR 6070), Les Algorithmes, 2000, route des Lucioles, B.P. 121, 06903 Sophia Antipolis Cedex, France

Received 29 September 2003; received in revised form 21 September 2004; accepted 14 October 2004
Available online 30 November 2004

Abstract

This paper describes *SOS*Sub$_{\mathcal{C}}$: a system for automatically translating programs written in Sub$_{\mathcal{C}}$, a simple imperative language, into a set of first-order equations. This set of equations represents a Sub$_{\mathcal{C}}$ program and has a precise mathematical meaning; moreover, the standard techniques for mechanizing equational reasoning can be used for verifying properties of programs. Part of the system itself is formulated abstractly as a set of first-order rewrite rules. Then, the rewrite rules are proven to be terminating and confluent. This means that our system produces, for a given Sub$_{\mathcal{C}}$ program, a unique set of equations. In our work, *simple* imperative programs are equational theories of a logical system within which proofs can be derived.

Keywords: Imperative program transformation; Operational semantics; Equational semantics; Compiling; Rewriting; Rewrite system; Program verification

1. Introduction

In most cases in which a program specification is done correctly, software deficiencies that come from the gap between the specification and its actual coding are far more

* Corresponding author. Tel.: +33 4 92 94 27 46; fax: +33 4 92 94 28 98.
E-mail addresses: ponsini@i3s.unice.fr (O. Ponsini), Carine.Fedele@unice.fr (C. Fédèle), kounalis@i3s.unice.fr (E. Kounalis).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.10.001

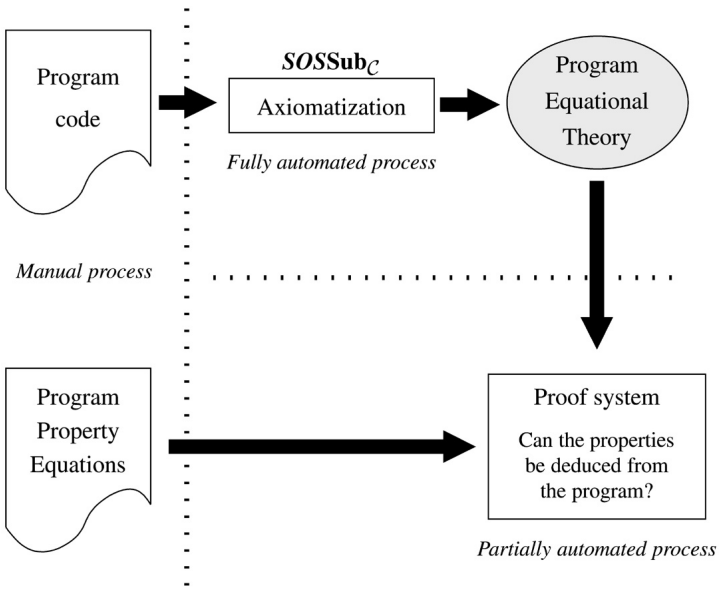

Fig. 1. The proof process overview.

numerous than errors due, for instance, to hardware failure or to the compiler. To increase confidence in code production, effort should be focused on verifying that programs meet their requirements, that is, that they are sound with respect to their specifications.

When dealing with *program soundness*, developers usually use empirical methods such as the test sets approach. But this is not sufficient for applications that need a high degree of reliability. For validation, applications would strongly benefit from *formal methods*, i.e. mathematical tools and techniques aimed at specifying and verifying software or hardware systems. By verification, we mean the analysis that demonstrates that a program has the desired properties.

1.1. Outline of our approach and related work

In [21,7], we promote the idea of generating equations from imperative programs. The principle is to translate source code into a set of first-order equations expressing the program semantics. This translation is part of a framework for automatically proving properties of programs. *Equational logic*, a subset of first-order logic, benefits from a simple and clear semantics (substitution of equals), as well as from efficient algorithms and existing tools.

The general outline of our framework is shown in Fig. 1. Developers write down the Sub$_C$ code of a program; they also write program specifications as a set of required properties expressed in equational logic: the *program property equations*. Then the *axiomatization* (*SOS*Sub$_C$ system) automatically transforms the source code into a set of equations: the *program equational theory*. These equations constitute a theory within equational logic. They also can be seen as an algebraic specification. The properties to be proved are conjectured theorems. Therefore, proving these theorems from the equational