J. W. de Bakker W.-P. de Roever
G. Rozenberg (Eds.)

Semantics: Foundations and Applications

REX Workshop
Beekbergen, The Netherlands, June 1-4, 1992
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
W-7500 Karlsruhe, FRG

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editors

J. W. de Bakker
Centre for Mathematics and Computer Science
P. O. Box 4079, 1009 AB Amsterdam, The Netherlands

W.-P. de Roever
Institute of Computer Science and Practical Mathematics II
Christian-Albrechts-Universität zu Kiel, Preußerstraße 1-9, W-2300 Kiel, FRG

G. Rozenberg
Department of Computer Science, Leiden University
P. O. Box 9512, 2300 RA Leiden, The Netherlands

CR Subject Classification (1991): F.3, D.1-3

ISBN 3-540-56596-5 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-56596-5 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author/editor
45/3140-543210 - Printed on acid-free paper

Preface

The aim of the workshop on 'Semantics - Foundations and Applications' was to bring together researchers working on the semantics of programming languages. Faithfully reflecting the rich variety in present-day semantic research, the program of the workshop included presentations on a wide range of topics situated in the two areas:

Foundations

- comparative domain theory, category theory, information systems,

Applications

- concurrency - process algebras, asynchronous communication, trace nets, action semantics, process refinement, concurrent constraint programming,
- predicate transformers, refinement, weakest preconditions,
- comparative semantics of programming concepts, full abstraction,
- reasoning about programs - total correctness, epistemic logic,
- logic programming,
- functional programming - sequentiality, integration with concurrency,
- applied structured operational semantics,

and several others.

The present volume is based on this meeting which the editors organized June 1-4, 1992, in Conference Centre De Wipselberg, Beekbergen, The Netherlands. The workshop was an activity of the project REX - Research and Education in Concurrent Systems, one of the projects sponsored by the Netherlands NFI (Nationale Faciliteit Informatica) Programme. A short description of the REX project is given below.

The material presented in this volume was prepared by the lecturers (and their coauthors) after the meeting took place - in this way the papers also reflect the discussions that took place during the workshop. The editors moreover invited a few authors to contribute papers not based on work presented during the meeting. We were fortunate to enjoy the cooperation of such an excellent group of lecturers and further participants. We are grateful to all of them for contributing to the success of the event. Special thanks go to Jan Rutten for his help in preparing the scientific program of the workshop.

We gratefully acknowledge the financial support for the workshop from the NFI programme.

The CWI, Amsterdam, was responsible for the technical organization of the meeting. The local organization was in the capable hands of Mieke Bruné and Frans Snijders.

The REX project

The REX - Research and Education in Concurrent Systems - project investigates syntactic, semantic and proof-theoretic aspects of concurrency. In addition, its objectives are the education of young researchers and, in general, the dissemination of scientific results relating to these themes. REX is a collaborative effort of Leiden University (G. Rozenberg), the CWI in Amsterdam (J.W. de Bakker), and the Eindhoven University of Technology (W.P. de Roever), representing the areas of syntax, semantics and proof theory, respectively. The project is supported by the Netherlands National Facility for Informatics (NFI); its duration is approximately six years starting in 1988. The educational activities of REX include regular "concurrency days", consisting of tutorial introductions, presentations of research results, and lecture series of visiting professors. The research activities of the REX project include, more specifically:

a) Three subprojects devoted to the themes: syntax of concurrent systems; comparative semantics, metric transition systems and domain theory; and high-level specification and refinement of real-time distributed systems.
b) Collaboration with visiting professors and post-doctoral researchers.
c) Workshops and Schools. Aiming at a broad coverage of major themes in, or relating to, concurrency, REX has organized the following events:

1988 Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency
Proceedings published as Springer Lecture Notes in Computer Science 354
1989 Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness
LNCS 430
1990 Foundations of Object-Oriented Languages
LNCS 489
1991 Real-Time: Theory in Practice
LNCS 600
1992 Semantics: Foundations and Applications
These Proceedings.

The project closes in 1993 with the School/Symposium "A Decade of Concurrency - Reflections and Perspectives", where the accomplishments in the field of concurrency will be surveyed and a look into the future will be attempted as to (un)expected developments.

February 1993

J.W. de Bakker
W.P. de Roever
G. Rozenberg

Table of Contents

Predicate Transformers and Higher Order Logic

R.J.R. Back
Åbo Akademi University, Department ofComputer Science
Lemminkäinengatan 14, SF-20520 Åbo, Finland

J. von Wright
Swedish School of Economics and Business Education
Biblioteksgatan 16, SF-65100 Vasa, Finland

ABSTRACT Predicate transformers are formalized in higher order logic. This gives a basis for mechanized reasoning about total correctness and refinement of programs. The notions of program variables and logical variables are explicated in the formalization. We show how to describe common program constructs, such as assignment statements, sequential and conditional composition, iteration, recursion, blocks and procedures with parameters, are described as predicate transformers in this framework. We also describe some specification oriented constructs, such as assert statements, guards and nondeterministic assignments. The monotonicity of these constructs over the lattice of predicates is proved, as well as the monotonicity of the statement constructors with respect to the refinement ordering on predicate transformers.

Key words Stepwise refinement, weakest preconditions, total correctness, predicate transformers, higher order logic, HOL, semantics of programming languages, state spaces, nondeterminism, procedures

CONTENTS

1 Introduction

1. Statements of a programming language can be given a semantics by associating every statement with a predicate transformer, i.e., a function mapping predicates to predicates. The *weakest precondition* semantics associates a statement with a predicate transformer with the following property: each postcondition is mapped to the weakest precondition that guarantees that the execution of the statement will terminate in a final state that satisfies the postcondition. This semantic interpretation of statements is useful for reasoning about total correctness and refinement of programs and specifications [5, 1].

The proofs used in such reasoning are usually semi-formal, done in the tradition of classical mathematics. This proof method generally works well, but there are situations when a higher level of formality is desirable. For example, reasoning about blocks with local variables is often done without an exact definition of the status of the local variables.

In this paper we show how reasoning in the weakest precondition framework can be given a solid logical foundation by using higher order logic (simple type theory) as a basis. We describe a programming notation that covers basic programming constructs, as well as blocks with local variables, recursion and procedures with parameters. Statements are predicate transformers, defined as terms of higher order logic. This formalization captures the weakest precondition semantics of the corresponding traditional programming notations.

An important property of statements in the weakest precondition calculus is monotonicity. All reasonable statements of a programming notation should denote monotonic predicate transformers. Statement constructors should also be monotonic with respect to the refinement relation on statements [1]. We prove that all the statement constructors introduced here have both these monotonicity properties.

2. One of our main motivations for this work is the desire to mechanize reasoning about programs, using a theorem prover based on simple type theory. One such prover is HOL [6], and we have admittedly been inspired by the HOL logic when we developed this theory. Our aim is to overcome some of the problems encountered in formalizing the theory of imperative languages using theorem provers [7, 6, 4].

The formalization of predicate transformers and refinement calculus as described here has in fact been implemented in HOL as a mechanized theory. The monotonicity results stated here have also all been constructed and checked in HOL.

3. The paper is organized as follows. In the next section, we give a very brief overview of higher order logic. In Section 3 we describe the basic ideas underlying our formalization of predicate transformers. Section 4 shows how to define basic program statements within this framework. Section 5 introduces

some additional constructs, that are found useful in practice, and which can be defined in terms of the basic constructs. Section 6 proves that all the constructs introduced have the required monotonicity properties. Section 7 ends with a few comments and remarks.

2 Higher order logic

1. The logic assumed is a polymorphic higher order logic. We assume that there is a collection of basic types. Every type σ is interpreted as a set (also denoted σ). Examples of basic types are `bool` (the booleans), `num` (the natural numbers) and `int` (the integers). We adopt the convention that constants and type names are written in typewriter font.

We use traditional symbols for logical connectives. The boolean truth values are denoted `F` (falsity) and `T` (truth). The scope of binders ($\forall$, $\exists$ and λ) extends as far to the right as possible.

From the basic types we can form new types by repeatedly applying *type constructors*: we will need only product types $\sigma \times \tau$ and function types $\sigma \to \tau$, defined as usual. For a given type σ, the *predicate type* $\overline{\sigma}$ is defined by

$$\overline{\sigma} \stackrel{\text{def}}{=} \sigma \to \texttt{bool}.$$

This type is so common in our treatment that it is convenient to have it as an abbreviation.

2. The elements of $\overline{\sigma}$ can also be interpreted as sets, by identifying a set with its characteristic function. Thus p is identified with the set

$$\{s \mid p\, s\}$$

Then we can write e.g., $\emptyset$ for `false` and $p \cup q$ for $p \wedge q$. We also have that $v \in p$ is equivalent to $p\, v$. We will use the predicate and the set notation interchangeably, choosing whichever is more convenient for the moment.

We also generalize the set notation in the following way: for arbitrary $q : \alpha \to \beta$ and $p : \alpha \to \texttt{bool}$ the notation

$$\{q\, s \mid s : p\, s\}$$

(the set of all $q\, s$ where s ranges over all values such that $p\, s$ holds) stands for the corresponding characteristic function

$$\lambda s'.\, \exists s.\, p\, s \wedge (s' = q\, s)$$

3. In the HOL system, rigorous proofs are carried out within the framework of a sequent calculus. In order to make proofs shorter, we use an informal calculational proof style in this paper. However, all proofs are easily transformed into formal proofs.

Since the logic is higher-order, we permit quantification and lambda abstraction over arbitrary types. Functions can have arguments of any type. New constants can be introduced by simple definitions. When defining a function f we often write

$$f\, x \quad \stackrel{\text{def}}{=} \quad E$$

rather than the equivalent $f \stackrel{\text{def}}{=} \lambda x.\, E$. Note that in a definition such as (1), all free variables of E must occur free on the left hand side also.

4. We permit type variables α, β and γ in types. A type variable can be instantiated to any type (even to a type containing type variables). This means that we can define *polymorphic constants*. An example of a polymorphic constant is infix equality, with type

$$=: \alpha \to \alpha \to \texttt{bool}$$

(the fact that a term t has type σ is indicated by writing $t : \sigma$).