Predicative Programming — A Survey

Jim Grundy*

University of Cambridge, Computer Laboratory
New Museums Site, Pembroke Street, Cambridge CB2 3QG, England

Abstract. The idea of using a predicate to specify behaviour is a compelling one, and leads to a desire to refine specifications into implementations in languages whose semantics have also been specified with predicates. Many of us believe we share a common intuition about what a specification phrased as a predicate means. It may be surprising to learn that there are several ways to interpret a predicate as a specification. Under these interpretations, the same predicate can specify different behaviour. This paper examines three simple styles of specification using predicates.

1 Introduction

The idea of using a predicate to specify behaviour is a compelling one. The specification languages HP-SL [1] and Z [7] are based on it. Predicates in higher-order logic are used for the specification and verification of hardware [2]. The use of predicates to specify the required behaviour of programs leads to a desire to be able to refine specifications into implementations in programming languages whose semantics have also been specified with predicates. Various techniques for making such refinements have been proposed [4, 5, 8, 10, 12, 14, 18].

One reason for the popularity of specifying with predicates is clarity; predicates and the predicate calculus are well understood logical concepts. When a behaviour is specified with a predicate, those behaviours that satisfy the predicate meet the specification, all others do not.

Many of us believe we share a common intuition about what a specification phrased as a predicate means. It may be surprising to learn that there are several ways to interpret a predicate as a specification. Under these interpretations the same predicate can describe different behaviour. This paper examines three simple styles of specification using predicates. We compare these styles, highlighting their strengths and weaknesses. In particular, we examine how the semantics of an imperative programming language can be expressed in each style.

2 The Basic Concepts

Before we begin, we must introduce some terminology.

* The author is supported by DSTO Australia.
state: We use the term state to refer to the state of a computer. The state is a vector of program variables, for example \((w, x, y, z)\), which we refer to as \(\sigma\). The set of all states is denoted by the symbol \(\Sigma\).

initial state: The initial state of a computer is the state before a computation begins. We denote the value of program variable \(x\) in the initial state by \(\sigma\), and the entire initial state by \(\sigma\).

final state: The final state of a computer is the state that it is left in after a computation finishes. We denote the value of program variable \(x\) in the final state by \(x'\), and the entire final state by \(\sigma'\).

nontermination: When a computation finishes it is said to have terminated. The term nontermination is used to describe a computation that never finishes. This may happen either because the computation encounters an error and aborts, or because the computation enters an infinite loop. We do not distinguish between these two possibilities.

outcome: The outcome of a computation is either the final state in which it terminates, or nontermination. We introduce the symbol \(\bot\) to denote nontermination. The set of all outcomes, written \(\Sigma_\bot\), is the disjoint union of the set of all states and nontermination:

\[
\Sigma_\bot \equiv \Sigma + \{\bot\}
\]

behaviour: For any computation, the behaviour of a computer can be characterised by the initial state in which the computation was begun and the outcome of the computation. Behaviours are therefore elements of \(\Sigma \times \Sigma_\bot\).

specification: A specification is a description of the possible behaviours of a computation. A specification is a function from behaviours to booleans, that is of type \((\Sigma \times \Sigma_\bot) \rightarrow \mathbb{B}\). We say that a behaviour \(b\) meets a specification \(S\), if \(S\) applied to \(b\) is true.

nondeterminism: If for any initial state only one behaviour meets a specification, then the specification is said to be deterministic. If, however, several behaviours would meet a specification, then that specification is said to be nondeterministic.

If we are attempting to prove a property of some system, we should assume that it behaves demonically. If a specification is nondeterministic and only some of the possible behaviours are ones that we want, then a demonic interpretation of the specification always chooses one of the behaviours that we do not want.

The opposite of demonic nondeterminism is angelic nondeterminism. If a specification is nondeterministic and only some of the possible behaviours are ones that we want, then an angelic interpretation of the specification always chooses one of the behaviours that we do want.

refinement: We say that a specification \(T\) is a refinement of another specification \(S\), if any behaviour that meets \(T\) also meets \(S\). The relationship \(\text{"} S \text{ is refined by } T \text{"} \) is written \(\text{"} S \sqsubseteq T \text{"} \).

\[
\text{"} S \sqsubseteq T \text{"} = (\forall b. (T \ b) \supset (S \ b))
\]

From this definition we can see that refinement is the process of reducing nondeterminism in a specification.