A pi-calculus Semantics for an Object-Based Design Notation

C. B. Jones
Department of Computer Science
Manchester University, M13 9PL, UK
cbj@cs.man.ac.uk

Abstract. Companion papers give examples of the development of concurrent programs using a design notation which employs a number of concepts from object-oriented programming languages. This paper documents the semantics of the design language by providing a mapping to the pi-calculus.

1 Introduction

It has long been realized that restrictions on programming languages are key to the development of concurrent programs. Both [Jon93a, Jon93b] use object-based concepts to facilitate the development of concurrent programs: the former uses constraints on the object graphs to limit interference; the latter employs assertions to reason about about interference.

The design language used in the above referenced papers is known as $\pi o\beta \lambda$. The $\pi o\beta \lambda$ class presented in Figure 1 provides a sorting facility. Each instance of the class has two instance variables. The variable $v$ contains the natural number which is stored in one element of the class and the variable $l$ – if not nil – contains a reference to another instance of this class. By tracing along this series of references, one can collect the sorted sequence of values from the $v$ variables. The class – and thus each of its instances – has three methods. The $add$ method accepts a non-zero natural number and stores it at an appropriate place in the sequence of instances; the $remove$ method returns the first (i.e. lowest) element of the sequence; and the $test$ method determines whether a given natural number is or is not in the sorted vector (for simplicity, zero is used to mark a nil value). The semantics of $\pi o\beta \lambda$ is such that only one method can be active in any one instance of a class at a time. The code which invokes a method is held in a rendezvous until the method being executed reaches a return statement. Notice that the $add$ method contains a return as its first statement so, as soon as the parameter has been passed, the caller is released from the rendezvous. The remainder of the code of the $add$ method does what one might expect: the higher value is passed down a chain of method-calls linked by the $l$ instance variables. Because of the way the returns work, concurrency is possible within a sequence of instances of the $Sort$ class. The only other point in $add$ is to notice that when the first value is stored in an instance of the class, a new next item is created in the linked-list. The $remove$ method is similar, noting only that it destroys the final element of the linked-list of instances when a zero is passed back from further down the list. The idea of obtaining concurrency by making sure that returns are executed as soon as possible would also be useful in
the test method. But, here the invoking procedure is bound to be held up because a value is required. The effect of the yield statement in test is to delegate the task of returning a value but to terminate this instance of the test method so that other methods can be invoked on the same instance of the class.

Sort class
vars v: [N] ← 0; l: private ref(Sort) ← nil
add(x: N1) method
  return
  if v = 0 then (v ← x; l ← new Sort)
  elif v ≤ x then !!add(x)
  else (!!add(v); v ← x)
fi
rem() method r: N1
  return v
  if v ≠ 0 then v ← !!rem()
    if v = 0 then l ← nil fi
fi
test(x: N1) method r: B
  if x = v then return true
  elif v = 0 ∨ x ≤ v then return false
  else yield !!test(x)
fi

Fig. 1. Example program Sort

It is worth noting that the specification of the Sort class in Figure 1 is by no means trivial. It is not possible to write a conventional pre/post-condition specification because the initial state to which one might expect to relate the final state in a post-condition is in fact a composite of the values stored in the v variables and any activity of add and remove operations which is still rippling down the list. It would be necessary in order to write such a specification to use something like Lamport’s ‘prophecy variables’. The text presented in Figure 1 is actually developed in [Jon93a] via a sequential (i.e. non-concurrent) version of the same algorithm. The sequential version differs from that presented in Figure 1 by having the return statements placed at the end of add and remove methods and the yield statement in test written as a return. The sequential program is easy to specify and to develop by normal concepts of data reification and operation decomposition; the final concurrent code presented in Figure 1 is derived by a step of transformation. Of course, it is necessary to know that such transformation rules are correct in the sense that they preserve the intended behaviour of the methods of the class; in other words it is necessary to show under what circumstances it is valid to permute the return statements in methods and to substitute yield statements for return statements.

Although presented as a program it is not the intention that ποβλ be seen as a new concurrent object-oriented programming language. It would certainly be simpler to use ποβλ as a design notation and to implement the programs in some language