Algebraic Proofs of Properties of Objects

David Walker

Department of Computer Science
University of Warwick
Coventry CV4 7AL, U.K.

Abstract

A semantics by translation to a process calculus is used as the basis for an investigation of transformations to programs expressed in a parallel object-oriented language. Two concrete examples are studied. In one it is shown that transformations introducing concurrency into the design of a priority queue class do not alter the observable behaviour. In the other, a more delicate relationship between two symbol table classes is established.

1 Introduction

In [5] a development method for parallel programs is proposed. Central to it are the application of program transformations to control the introduction of concurrency into designs and the use of concepts from object-oriented programming to constrain interference. Characteristic of the object-oriented style of programming is the description of a computational system as a collection of *objects* each of which is a self-contained entity possessing data and procedures, or *methods*. A parallel object-oriented program typically describes a highly mobile concurrent system in which new objects are created as computation proceeds and the linkage between components changes as references to objects are passed in communications. Providing adequate, tractable models for such systems is challenging. But the challenge is one which must be addressed if program transformations are to be subject to rigorous justification.

Among the most well-developed work on semantics of parallel object-oriented languages is that on the POOL family [3]. For example in [1] an operational semantics for a member of the family is given while [2] offers a denotational semantics based on metric spaces. An alternative technique was introduced in [13] where semantics for two small parallel object-oriented languages were given by translation to the π-calculus [9], a process calculus in which one can naturally express systems which have changing structure. This work was extended in [14] where a semantics for POOL was given by translation to the polyadic π-calculus [8] and a close correspondence between it and a two-level operational semantics, a reformulation of the semantics of [1], was established. The examples in the present paper are expressed in the language $\pi o\beta\lambda$ of [5]. Semantics for it have been given by translation to the polyadic π-calculus [6] and to the higher-order π-calculus [15] introduced in [12].

The basic entities of the π-calculus and its descendants are *names*. In the (polyadic) π-calculus processes communicate by using names to pass (tuples of) names to one another. In the higher-order π-calculus processes may also pass to one another process abstractions of arbitrarily high order. In the semantics by translation each entity – class, object, method, variable, statement, expression – is encoded as a process (or higher-order abstraction), with the representation of a composite phrase being defined directly in terms of those of its

constituents. Execution of a program is represented by reduction of the process encoding it. In particular, the passing of references between objects is captured naturally as the passing of names between the processes representing them with the treatment of *private* names afforded by the restriction operator of the π-calculus playing a central rôle. An important contribution to the perspicuity of the translations is made by the imposition of a sort discipline [8] which constrains the use of names; and in the semantics using the higher-order π-calculus, each program constructor is represented as a higher-order process abstraction. These refinements help to retain some of the high-level structure of programs.

One advantage of giving programming language semantics by translation to a process calculus whose semantic basis is established is that we are thereby freed from the task of constructing an adequate semantic domain and establishing that it supports constructions necessary to give a faithful interpretation of the language. In the case of parallel object-oriented languages, accomplishing this task can require something of a *tour de force*; see e.g. [2]. But the use of this method of semantic definition offers a second potential benefit and it is this that the present paper begins to explore. A topic of central concern in process calculus has been to give abstract descriptions of the behaviour of processes and to develop techniques for reasoning about this behaviour. By encoding language phrases, and in particular programs, as processes, we can use process calculus apparatus to derive abstract descriptions of the behaviour they express and to reason about it. In this paper we investigate this possibility by studying two concrete examples from [5]. Principal aims of this study are to illustrate the utility of process calculi in a rigorous treatment of such problems, and to contribute to the clarification of the difficulties involved in finding and proving sound general transformation rules for parallel object-oriented programs of the kind proposed in [5].

The process calculus used in this paper is an extension of the polyadic π-calculus with value expressions and conditional agents. We do not carry through in detail here the amalgamation of the well-established theories of calculi in which processes exchange simple data values and the π-calculus. Rather we state the results which we assume of the process calculus; they are all well-known in other settings. As mentioned earlier, data can be encoded as π-calculus processes or, more cleanly, as abstractions in the higher-order π-calculus. The reason for working in the less parsimonious setting is that our aim is to give proofs which are both simple and natural, and coding data such as integers and booleans as π-calculus processes is not helpful in achieving this.

It is not possible in the space available to give full accounts of the programming language, the process calculus, the translational semantics, the examples and the analysis of them. In the following section we introduce the examples. Section 3 contains a condensed acçount of the calculus and an illustration of the semantics. In Section 4 we describe the analysis of the first example and in Section 5 very briefly sketch that of the second. A fuller account of this work is given in [16].

Acknowledgement. I am grateful to Matthew Hennessy, Cliff Jones and Davide Sangiorgi for comments on this work.

2 The Examples

The first example concerns two integer priority queue classes. The first is sequential while the second can be viewed as being obtained from it by transformations which are intended to introduce some concurrency. The first class definition is as follows:

```
class Q
var V:NAT, P:Q
```

```
method Add(X:NAT)
   if V=nil then (V:=X ; P:=new(Q))
   else if V<X then P!Add(X)
        else (P!Add(V) ; V:=X) ;
   return
method Rem():NAT
   var T:NAT
   T:=V ;
   if not(V=nil) then (V:=P!Rem() ; if V=nil then P:=nil) ;
   return T
```

A class definition provides a template for its *instances*, the objects of that class. Each instance of Q represents a cell which stores a nonnegative integer in the variable V and a pointer to another cell in the variable P. A priority queue is composed of a chain of cells in which integers are stored in ascending order. The value of the expression new(Q) is a reference to a new object of the class. On creation of a cell the values of V and P are undefined (nil). New cells are created and appended to the chain as integers are added via the method Add which takes an integer parameter and returns no result. The smallest integer held in the queue (or nil if the queue is empty) is returned and removed by the Rem method. Evaluation of the expression P!Rem() involves the invocation in the object to which the value of P is a reference of the method Rem. Execution of the invoking object is suspended until an integer is returned to it, this being the value of the expression; a cell returns its value in the Rem method by executing the statement return T. Similarly, the statement P!Add(X) represents an invocation of the method Add in the object to which the value of P is a reference with the value of X as parameter. Again the activity of the invoking object is suspended until it is released from the rendezvous; in this case no value is returned and the appropriate releasing statement is return. An invariant of a queue of cells is that only the first cell is accessible to other objects. When an Add method is invoked in the first cell its activity is suspended until the integer is inserted into the queue at the appropriate point and a return signal ripples back along the queue to the first cell which then releases the caller from the rendezvous. A similar discipline constrains the Rem method. In $\pi o \beta \lambda$, at most one method may be active in an object at any time: a cell may not accept another method invocation until it has completed execution of the active method body.

The second class definition is as follows:

```
class Q′
var V:NAT, P:Q′
method Add(X:NAT)
    return ;
    if V=nil then (V:=X ; P:=new(Q′))
    else if V<X then P!Add(X)
         else (P!Add(V) ; V:=X)
method Rem():NAT
    return V ;
    if not(V=nil) then (V:=P!Rem() ; if V=nil then P:=nil)
```

Q′ differs from Q in that when the Add method is invoked in a cell it executes its return statement immediately, thus freeing the calling object from the rendezvous; the two may thus proceed in parallel. The Rem method acts in a similar way; in this case the local variable T is no longer required.

A $\pi o \beta \lambda$ program consists of a sequence of class definitions together with an indication of which class furnishes the *root object* which alone exists at the beginning of a computation.

Suppose *prog* is a program in which the class Q is defined, and suppose *prog'* is obtained from it by replacing the definition of Q by that of Q' and each mention of Q by mention of Q'. The functions of this transformation are to increase the scope for concurrency in execution of the program, and to do so without altering its observable (input and output) behaviour. That the first intention is fulfilled will not be argued for here, though evidence that it is is discernible in the argument which we give to show that the observable behaviour of the program is not altered. The translational semantics associates with the programs *prog* and *prog'* agents P and P'. We may express the property of interest as the assertion that $P \simeq P'$ where $\simeq$ is an appropriate one of the many notions of behavioural equivalence which have been studied in process calculus. We show in Section 4 that $P \simeq P'$ holds when for $\simeq$ we take *weak bisimulation equivalence* [7, 9].

The second example concerns two symbol table classes in which references to objects of some class A are associated with nonnegative integer keys. Again the first class is sequential while the second can be viewed as being obtained from it by transformations which are intended to introduce some concurrency. The first class definition is as follows:

```
class T
var K:NAT, V:A, L:T, R:T
method Insert(X:NAT, W:A)
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W) ;
    return
method Search(X:NAT):A
    if K=nil then return nil
    else if X=K then return V
         else if X<K then L!Search(X)
              else R!Search(X)
```

A table is structured as a binary search tree in which each key occurs at most once. Each instance of T represents a node of the tree. Only the root node is accessible to other objects. A node has a key K, a value V which is a pointer to an object of the class A, and pointers L and R to the left and right subtrees. On creation all have undefined values; this it true also of the variables of leaf nodes. The methods of T are Insert for inserting a key-value pair and Search which returns the reference associated with the key supplied as parameter (if it occurs in the table). Analogously to the behaviour of the sequential priority queue Q, when a method is invoked in the root node of a tree its activity is suspended until a return ripples back through the tree to it when it releases the caller from the rendezvous.

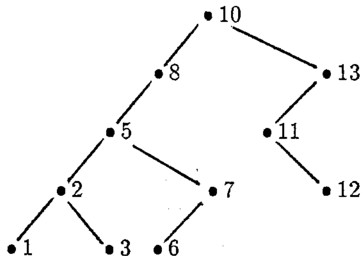
The second class definition is as follows:

```
class T′
var K:NAT, V:A, L: T′ , R: T′
method Insert(X:NAT, W:A)
    return ;
    if K=nil then (K:=X ; V:=W ; L:=new(T′) ; R:=new(T′))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W)
method Search(X:NAT):A
    if K=nil then return nil
    else if X=K then return V
```

```
else if X<K then commit L!Search(X)
     else commit R!Search(X)
```

Two transformations are applied to obtain T′ from T. The first involves moving the `return` statement in the method `Insert` to the beginning of the body. The second transformation involves the replacement in the `Search` method of the invocations of `Search` methods in the left and right subtrees by `commit` statements (in [5] '`yield`' is used instead of '`commit`'). When an object α executes a `commit` statement by invoking a method in an object β, it is implicit (i) that β should return its result not to α but to the object γ to which α should return a result, and (ii) that α is freed from the task of returning a result to γ. In particular, execution of α may continue in parallel with that of β. In T′, if the `Search` method is invoked in a node with a key smaller (resp. larger) than that stored there, the node will commit that search to its left (resp. right) child. We may think of the node as passing to the child a return address to which the result of the search should be sent. This address will have been received by the node either directly from the initiator of the search or from its parent in the tree.

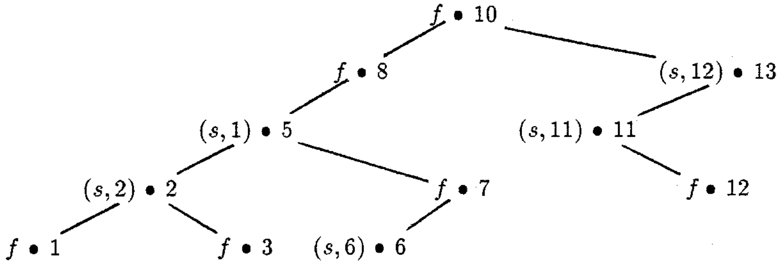
The effect of moving the `return` statement to the beginning of the `Insert` method is similar to that resulting from the analogous transformation to the `Add` method of the class `Q`. More interesting are the transformations which introduce the `commit` statements. A symbol table constructed from nodes of class T′ may support a number of concurrent searches. The order in which the results of these searches may be returned is intimately related to the tree structure of the table. For example consider the following tree structure in which no method is active in any of the nodes (the nature of the references associated with the keys is irrelevant to the discussion):



The root node may accept the sequence of invocations

```
Search(7) ; Search (6) ; Search (2) ; Search(11) ; Search(12) ; Search(1)
```

without the result of any of them being returned. Thereafter the results of the searches with keys 7 and 11 may be returned but the results of the searches with the other four keys may not as the progress of each of them is blocked by uncompleted searches in relevant parts of the tree structure. An important point here is that since only one method may be active in an object, one search cannot 'overtake' another. If the result of the search with key 7 is returned and each of the searches makes as much progress as is then possible, the state of the table may be pictured thus:



Here we annotate each node with a *status* recording if the node is active and if so how it is engaged. The status is f ('free') if the node is inactive and (s, k) if it is executing a search with parameter k. In this state the results of the searches with keys 2, 6 and 11 may be returned, but the search with key 1 is blocked until that with key 2 returns and the search with key 12 is blocked by the unfinished search with key 11.

It is thus clear that the correspondence between the classes T and T′ is not a simple one. Account must also be taken of the progress of invocations of `Insert`. Further, in the case of infinite computations issues related to fairness arise. The analysis we give involves finding an abstract description of the behaviour of the process representing a newly-created table, and showing how the finite parts of this behaviour are related to those of a process describing the behaviour of a newly-created table of class T. Briefly, we show that if μ is a finite sequence of method invocations then the possible computations of the T′-table generated by μ are the parallelizations allowed by the tree structure of the computation of the T-table generated by μ.

3 The Language, the Calculus and the Semantics

The language $\pi o\beta\lambda$ in which the examples are expressed is described in [5, 6]. Semantics by translation for it appear in [15], where the higher-order π-calculus is used, and in [6], which employs the polyadic π-calculus. An important development here is the use of an enrichment of the π-calculus with value expressions of simple types and conditional agents. This section contains a condensed description of the calculus and an illustration of the translation of $\pi o\beta\lambda$ to it using fragments of the priority queue class Q.

3.1 The Calculus

We assume a set $\mathcal{S}$ of *subject sorts* among which are two distinguished sorts N and B for the simple types of natural numbers and booleans. The set $\mathcal{O}$ of *object sorts* is $\mathcal{S}^*$, the set of finite tuples of subject sorts. We write $(s_1, \ldots, s_n)$ for the object sort of length n consisting of $s_1, \ldots, s_n \in \mathcal{S}$. A *sorting* is a function $\Sigma : \mathcal{S}_0 \to \mathcal{O}$ where $\mathcal{S}_0 = \mathcal{S} - \{\mathsf{N}, \mathsf{B}\}$. The basic entities of the calculus are link names and simple value expressions. The names may be thought of as names of channels via which processes may interact. Each name is assigned a subject sort and the sorting determines that only entities with the object sort $\Sigma(s)$ may be communicated using a name of sort $s \in \mathcal{S}_0$; these entities are tuples of names and values of the simple types (sorts). The subject sorts N and B have no associated object sorts: entities of these sorts are not names of links. This kind of sorting discipline was introduced in [8]. It was refined in [11] with the adoption of structural matching, rather than name matching, of sorts, and the introduction of input/output tags allowing a natural form of subsorting. Here we use structural matching of sorts but do not require input/output tags.

We assume an infinite set of names $x, y, \ldots$ of each subject sort and write $x : s$ to indicate that x is of sort s. We assume that N contains constant names $0, 1, \ldots, \bot$ and B constant names tt, ff, $\bot$, and that both have in addition an infinite number of variable names. A suitably expressive language of value expressions is assumed. We assume that each closed expression (i.e. expression containing no variable names) of sort N or B has a value which is a closed name of that sort (with $\bot$ representing an undefined value). We assume also an infinite set of agent constants K of each object sort θ, written $K : \theta$. The set of *agent expressions* P, Q is given by

$$P \ ::= \ \Sigma_{j \in J} \pi_j . P_j \ \mid \ P \mid Q \ \mid \ (\nu\, x)P \ \mid \ K\langle \widetilde{w} \rangle \ \mid \ \mathsf{if}(b : P, Q)$$

where: in $K\langle \widetilde{w} \rangle$, $\widetilde{w}$ ranges over tuples of link names and value expressions; in the conditional agent expression $\mathsf{if}(b : P, Q)$ ("if b then P else Q"), b ranges over value expressions of sort B; a *prefix* π is an output of the form $\overline{x}(\widetilde{w})$ or an input of the form $x(\widetilde{y})$; in a restriction $(\nu\, x)$, x has a sort in $\mathcal{S}_0$; the indexing set J in the guarded summation is finite; and each constant K has a defining equation $K \stackrel{\text{def}}{=} F$ where F is an *abstraction* of the form $(\widetilde{y})P$ where $\widetilde{y}$ is a tuple of distinct variable names containing all those occurring free in P (see below). We write $\mathbf{0}$ for an empty summation and π for $\pi.\mathbf{0}$. In $x(\widetilde{y}).P$ and $(\widetilde{y})P$ the occurrences of $\widetilde{y}$ are binding with scope P; the restriction operator $(\nu\, x)$ is also a binding operator. We assume the standard notions of free names, substitution (of link names for link names and value expressions for variable names of sort N or B), alpha-conversion etc. and identify agent expressions which differ only by change of bound names. We write $\mathrm{fn}(P)$ for the set of free names of P. We consider only *sort-respecting* substitutions, i.e. those σ such that x and $\sigma(x)$ have the same sort for every (variable) name x. We say a substitution σ is *closed* if for each variable name x of sort N or B, $\sigma(x)$ is a constant name. An agent expression containing no free variable names of sort N or B is called a *process*.

We consider only agent expressions which *respect* the sorting Σ. The essence of respecting Σ, which is defined formally by a family of rules [8], is that each well-sorted agent expression is assigned an object sort, with $P : ()$ for each process P, so that definitions, applications and prefixes are consistent with Σ. Thus extending ':' componentwise to tuples we have e.g. that if $K \stackrel{\text{def}}{=} (\widetilde{y})P$ and $K : \theta$ then $\widetilde{y} : \theta$ and in $K\langle \widetilde{w} \rangle$ we must have $\widetilde{w} : \theta$; and in $\pi = \overline{x}(\widetilde{w})$, if $x : s$ then $s \in \mathcal{S}_0$ and $\widetilde{w} : \Sigma(s)$. To allow a simple presentation of the dynamics of the calculus a relation of *structural congruence* on agent expressions, written $\equiv$, is defined; see [8]. The behaviour of processes is given by a family of labelled transition relations. They are obtained from the transition relations for the polyadic π-calculus by incorporating an appropriate treatment of value expressions. We adopt an 'early' instantiation regime [10] in which variable names of the sorts N and B are instantiated by constant names when an input action is inferred. The relations are labelled by actions α of which there are three kinds: the silent action τ, output actions $\overline{x}((\nu\, \widetilde{z})\widetilde{v})$ and input actions $x(\widetilde{v})$. Here $\widetilde{v}$ is a tuple each of whose entries is a link name (of a sort in $\mathcal{S}_0$) or a constant name (of sort N or B), and $\widetilde{z}$ is a subset of the set of link names occurring in $\widetilde{v}$; $\widetilde{z}$ is the set $\mathrm{bn}(\alpha)$ of bound names of the action α; also $\mathrm{bn}(\tau) = \emptyset$ and $\mathrm{bn}(x(\widetilde{v})) = \emptyset$. A *variant* of a transition $P \xrightarrow{\alpha} Q$ is a transition which differs only in that P and Q have been replaced by structurally-congruent processes and α has been alpha-converted, where a bound name of α includes Q in its scope. The relations are defined by the following rules. In each rule the conclusion stands for all variants of the transition:

1. $\overline{x}(\widetilde{w}).P + \ldots \xrightarrow{\overline{x}(\widetilde{v})} P$ where $\widetilde{v}$ is identical to $\widetilde{w}$ except that each closed expression of sort N or B is replaced by its value,

2. $x(\widetilde{y}).P + \ldots \xrightarrow{x(\widetilde{v})} P\{\widetilde{v}/\widetilde{y}\}$ provided the names of sort N or B in $\widetilde{v}$ are constant names;

here $P\{\widetilde{v}/\widetilde{y}\}$ is the result of substituting the components of $\widetilde{v}$ for the corresponding components of $\widetilde{y}$ in P,

3. If $P \xrightarrow{\alpha} P'$ then $P \mid Q \xrightarrow{\alpha} P' \mid Q$ provided $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$,
4. If $P \xrightarrow{\overline{x}((\nu\widetilde{z})\widetilde{v})} P'$ and $Q \xrightarrow{x(\widetilde{v})} Q'$ then $P \mid Q \xrightarrow{\tau} (\nu\widetilde{z})(P' \mid Q')$ provided $\widetilde{z} \cap \mathrm{fn}(Q) = \emptyset$,
5. If $P \xrightarrow{\alpha} P'$ then $(\nu x)P \xrightarrow{\alpha} (\nu x)P'$ provided x does not occur in α,
6. If $P \xrightarrow{\overline{x}((\nu\widetilde{z})\widetilde{v})} P'$ then $(\nu y)P \xrightarrow{\overline{x}((\nu\widetilde{z}\,y)\widetilde{v})} P'$ provided y occurs in $\widetilde{v} - (\widetilde{z} \cup \{x\})$,
7. If $P \xrightarrow{\alpha} P'$ then $\mathsf{if}(b : P, Q) \xrightarrow{\alpha} P'$ provided the value of b is tt,
8. If $Q \xrightarrow{\alpha} Q'$ then $\mathsf{if}(b : P, Q) \xrightarrow{\alpha} Q'$ provided the value of b is ff.

Following a standard line of development in process theory we write $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, for each α set $\overset{\alpha}{\Longrightarrow}$ to be $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$, and set $\overset{\widehat{\tau}}{\Longrightarrow}$ to be $\Longrightarrow$ and $\overset{\widehat{\alpha}}{\Longrightarrow}$ to be $\overset{\alpha}{\Longrightarrow}$ if $\alpha \neq \tau$. Then *weak bisimilarity* is the largest symmetric relation $\dot{\approx}$ on processes such that if $P \dot{\approx} Q$ then for all α, if $P \xrightarrow{\alpha} P'$ then for some Q', $Q \overset{\widehat{\alpha}}{\Longrightarrow} Q'$ and $P' \dot{\approx} Q'$. We extend $\dot{\approx}$ to agent expressions by setting $P \dot{\approx} Q$ if $P\sigma \dot{\approx} Q\sigma$ for each σ which is the identity on link names. Similarly for abstractions, $(\widetilde{y})P \dot{\approx} (\widetilde{y})Q$ if $P\{\widetilde{v}/\widetilde{y}\} \dot{\approx} Q\{\widetilde{v}/\widetilde{y}\}$ for all $\widetilde{v}$. The relation $\dot{\approx}$ is an equivalence relation preserved by most of the process constructors, but it is not preserved by instantiation of link names and hence not by input prefix. In carrying out the analysis of the programming examples we use the natural congruence relation on agent expressions determined by $\dot{\approx}$. Since we work only with guarded summation no special treatment of initial τ-actions is required to characterize the congruence. We say that agent expressions P and Q are *weakly equivalent*, $P \approx Q$, if for every substitution σ, $P\sigma \dot{\approx} Q\sigma$. We assume that $\approx$ is a congruence relation on agent expressions, that it is preserved by recursive definition, and that simple equations have unique solutions up to $\approx$. We use also an appropriate form of expansion theorem, some straightforward algebraic laws involving conditional agent expressions, and standard τ-laws.

3.2 The Translation

To give the translation we take as subject sorts of the calculus N, B, NIL, and for each type name T and tuple $\widetilde{T}$ of type names, $\mathsf{LINK}[T]$, $\mathsf{METHOD}[\widetilde{T};T]$ and $\mathsf{METHOD}[\widetilde{T};]$. Here $\widetilde{T};T$ (resp. $\widetilde{T};$) indicates a method taking arguments of types $\widetilde{T}$ and returning a result of type T (resp. no result type). To give the sorting it is convenient to introduce some synonyms. We set $\mathsf{OBJECT}[\mathsf{NAT}] \equiv \mathsf{N}$ and $\mathsf{OBJECT}[\mathsf{BOOL}] \equiv \mathsf{B}$. If A has methods $M_1, \ldots, M_q$ and M_i has type $U_i = \widetilde{T}_i;T_i$ or $\widetilde{T}_i;$ then we set $\mathsf{OBJECT}[A] \equiv \mathsf{METHOD}[U_1], \ldots, \mathsf{METHOD}[U_q]$. Finally, if $\widetilde{T} = T_1, \ldots, T_r$ then we set $\mathsf{OBJECTS}[\widetilde{T}] \equiv \mathsf{OBJECT}[T_1], \ldots, \mathsf{OBJECT}[T_r]$. The sorting Σ is then as follows:

$$\begin{aligned}
\Sigma(\mathsf{NIL}) &= () \\
\Sigma(\mathsf{LINK}[T]) &= (\mathsf{OBJECT}[T]) \\
\Sigma(\mathsf{METHOD}[\widetilde{T};T]) &= (\mathsf{OBJECTS}[\widetilde{T}],\ \mathsf{LINK}[T]) \\
\Sigma(\mathsf{METHOD}[\widetilde{T};]) &= (\mathsf{OBJECTS}[\widetilde{T}],\ \mathsf{NIL})
\end{aligned}$$

As an example, for the priority queue class $\mathtt{Q}$ we have

$$\begin{aligned}
\mathsf{OBJECT}[\mathtt{Q}] &\equiv \mathsf{METHOD}[\mathsf{NAT};],\ \mathsf{METHOD}[;\mathsf{NAT}] \\
\Sigma(\mathsf{METHOD}[\mathsf{NAT};]) &= (\mathsf{N},\ \mathsf{NIL}) \\
\Sigma(\mathsf{METHOD}[;\mathsf{NAT}]) &= (\mathsf{LINK}[\mathsf{NAT}])
\end{aligned}$$

Since we adopt structural matching of sorts, we have e.g. that LINK[Q] = LINK[Q'].

The translation function $[\![\cdot]\!]$ associates with each phrase of the language an agent constant whose object sort conveys some static information about the phrase. The sorting will be explained in detail as we proceed. Briefly, the 'object identifier' of an instance of the class Q will be a pair of names (add, rem) : (METHOD[NAT;], METHOD[; NAT]). The invocation of the Add method in such an instance will be represented as the sending to the process encoding it, via its *add* link, of a pair (x, r) : (N, NIL) with x being the parameter and r a link to be used for the return of the invocation. For the Rem method, along the *rem* link is sent a return link of sort LINK[NAT] for the return of the value. Channels of sort LINK[Q] are used to pass identifiers of objects of the class Q. The declaration of the class Q, for instance, is encoded as a replicator which may produce at a link of sort LINK[Q] an indefinite number of copies of the agent encoding an object of the class.

As a first illustration, abbreviating LINK[NAT], LINK[NAT] to LINK[NAT]^2 we have

$$[\![\texttt{var V:NAT}]\!] : (\mathsf{LINK[NAT]}^2) \stackrel{\text{def}}{=} (get\, put)\, \mathsf{REG_{NAT}}\langle get, put, \bot\rangle$$

where a natural number register is defined by

$$\mathsf{REG_{NAT}} : (\mathsf{LINK[NAT]}^2, \mathsf{N}) \stackrel{\text{def}}{=} (get\, put\, x) \\ (\overline{get}(x).\, \mathsf{REG_{NAT}}\langle get, put, x\rangle + put(y).\, \mathsf{REG_{NAT}}\langle get, put, y\rangle)$$

Here x represents the value stored, which is undefined on declaration, and *get* (*put*) the name used for reading (writing) the register. The agent $[\![\texttt{var P:Q}]\!] : (\mathsf{LINK[Q]}^2)$ is defined by

$$[\![\texttt{var P:Q}]\!] : (\mathsf{LINK[Q]}^2) \stackrel{\text{def}}{=} (get\, put)\, (\nu\, a, r)\, \mathsf{REG_Q}\langle get, put, a, r\rangle$$

where $\mathsf{REG_Q} : (\mathsf{LINK[Q]}^2, \mathsf{OBJECT[Q]})$ is similar to $\mathsf{REG_{NAT}}$. The nil reference is captured by the restriction. A sequence of variable declarations is encoded as the composition of the agents representing the individual declarations.

An expression E of type T is encoded as an agent constant $[\![E]\!] : \theta^\frown(\mathsf{LINK}[T])$ where θ consists of sorts corresponding to the distinct variables and new expressions occurring in E, and the last parameter is used for delivering the value of the expression. For constants and variables of type NAT we set

$$\begin{aligned} [\![\texttt{k}]\!] : (\mathsf{LINK[NAT]}) &\stackrel{\text{def}}{=} (val)\, \overline{val}(k) \\ [\![\texttt{nil}]\!] : (\mathsf{LINK[NAT]}) &\stackrel{\text{def}}{=} (val)\, \overline{val}(\bot) \\ [\![\texttt{X}]\!] : (\mathsf{LINK[NAT]}^2) &\stackrel{\text{def}}{=} (get\, val)\, get(x).\, \overline{val}(x) \end{aligned}$$

For the creation of a new object of class Q,

$$[\![\texttt{new(Q)}]\!] : (\mathsf{LINK[Q]}^2) \stackrel{\text{def}}{=} (new\, val)\, new(a, r).\, \overline{val}(a, r)$$

This agent is intended to receive at *new* from the replicator representing the declaration of the class Q a pair (a, r) representing the object identifier of a new instance of the class and to yield that pair as its value at *val*. Then we have

$$[\![\texttt{P:=new(Q)}]\!] : (\mathsf{LINK[Q]}^2, \mathsf{NIL}) \stackrel{\text{def}}{=} (new\, put\, d)(\nu\, val : \mathsf{LINK[Q]}) \\ ([\![\texttt{new(Q)}]\!]\langle new, val\rangle \mid val(a, r).\, \overline{put}(a, r).\, \overline{d})$$

Here *put* is a link to the register agent corresponding to the variable P and d a link on which the agent encoding the assignment statement signals termination. In general, a statement S is encoded as an agent $[\![S]\!] : \theta^\frown(\mathsf{NIL})$ where θ consists of sorts corresponding to the distinct variables and new expressions occurring in S together with names associated with method calls.

Continuing the illustration we have $[\![\texttt{P!Add(X)}]\!] : (\mathsf{LINK[Q]}, \mathsf{LINK[NAT]}, \mathsf{NIL})$ defined by

$$[\![\texttt{P!Add(X)}]\!] \stackrel{\text{def}}{=} (get_P\, get_X\, d) \quad (\nu\, val_0 : \mathsf{LINK[Q]},\, val_1 : \mathsf{LINK[NAT]},\, w : \mathsf{NIL}) \\ ([\![\texttt{P}]\!]\langle get_P, val_0\rangle \mid w.\, [\![\texttt{X}]\!]\langle get_X, val_1\rangle \\ \mid val_0(a, r).\, \overline{w}.\, val_1(y).\, (\nu\, ret)\overline{a}(y, ret).\, ret.\, \overline{d})$$

The third component receives the pair of links representing the reference stored in P, activates the second component and receives from it the value of the parameter, then sends that with a private return link along the Add-component of the reference. It then waits for a return along *ret* before signalling termination at d. The boolean expression V<X is encoded as an agent of sort $(\mathsf{LINK[NAT]}^2, \mathsf{LINK[BOOL]})$ thus:

$$[\![\texttt{V<X}]\!] \stackrel{\text{def}}{=} (get_V\, get_X\, val) \quad (\nu\, val_0 : \mathsf{LINK[NAT]}, val_1 : \mathsf{LINK[NAT]}, w : \mathsf{NIL}) \\ ([\![\texttt{V}]\!]\langle get_V, val_0\rangle \mid w.\, [\![\texttt{X}]\!]\langle get_X, val_1\rangle \mid \mathsf{LESS}\langle val_0, val_1, w, val\rangle)$$

where $\mathsf{LESS} : (\mathsf{LINK[NAT]}^2, \mathsf{NIL}, \mathsf{LINK[BOOL]})$ is defined by

$$\mathsf{LESS} \stackrel{\text{def}}{=} (val_0\, val_1\, w\, val)\, val_0(x_0).\, \overline{w}.\, val_1(x_1).\, \overline{val}(x_0 < x_1)$$

Continuing further, setting $\theta = (\mathsf{LINK[NAT]}^2, \mathsf{LINK[Q]}, \mathsf{LINK[NAT]}, \mathsf{NIL})$ we have

$$[\![\texttt{if V<X then P!Add(X) else (P!Add(V) ; V:=X)}]\!] : \theta \stackrel{\text{def}}{=} \\ (get_V\, put_V\, get_P\, get_X\, d) \quad (\nu\, val : \mathsf{LINK[BOOL]}, d_0 : \mathsf{NIL}, d_1 : \mathsf{NIL}) \\ ([\![\texttt{V<X}]\!]\langle get_V, get_X, val\rangle \\ \mid val(b).\, \mathsf{if}(b : \overline{d_0}, \overline{d_1}) \\ \mid d_0.\, [\![\texttt{P!Add(X)}]\!]\langle get_P, get_X, d\rangle \\ \mid d_1.\, [\![\texttt{P!Add(V) ; V:=X}]\!]\langle get_P, get_V, put_V, get_X, d\rangle)$$

Continuing with the illustration we have that for the body $\mathsf{S_{Add}}$ of the Add method of Q, $\mathsf{S_{Add}} : (\mathsf{LINK[Q]}, \mathsf{LINK[NAT]}^2, \mathsf{LINK[Q]}^2, \mathsf{LINK[NAT]}, \mathsf{NIL}^2)$ where the penultimate abstracted name is used in encoding the `return` statement in $\mathsf{S_{Add}}$. We then have $[\![\texttt{method Add(X:NAT)}, \mathsf{S_{Add}}]\!] : (\mathsf{LINK[Q]}, \mathsf{LINK[NAT]}^2, \mathsf{LINK[Q]}^2, \mathsf{METHOD[NAT;]}, \mathsf{NIL})$ defined by

$$[\![\texttt{method Add(X:NAT)}, \mathsf{S_{Add}}]\!] \stackrel{\text{def}}{=} (new\, get_V\, put_V\, get_P\, put_P\, add\, d) \\ (\nu\, get_X, put_X)\, ([\![\texttt{var X:NAT}]\!]\langle get_X, put_X\rangle \\ \mid add(y, ret).\, \overline{put_X}(y).\, [\![\mathsf{S_{Add}}]\!]\langle new, get_V, put_V, get_P, put_P, get_X, ret, d\rangle)$$

The encoding of the Rem method is along similar lines. Then an instance of class Q in the quiescent state is represented by the agent $\mathrm{Obj_Q} : (\mathsf{LINK[Q]}, \mathsf{OBJECT[Q]})$ defined as follows where $\widetilde{p} = get_V, put_V, get_P, put_P$:

$$\mathrm{Obj_Q} \stackrel{\text{def}}{=} (new\, add\, rem)(\nu\, \widetilde{p})([\![\texttt{var V:NAT, P:Q}]\!]\langle \widetilde{p}\rangle \mid \mathrm{Body_Q}\langle new, \widetilde{p}, add, rem\rangle)$$

where

$$\mathrm{Body_Q} \stackrel{\text{def}}{=} (new\, add\, rem)\, (\nu\, d) \quad (([\![\texttt{method Add}]\!]\langle ..., add, d\rangle + [\![\texttt{method Rem}]\!]\langle ..., rem, d\rangle) \\ \mid d.\, \mathrm{Body_Q}\langle new, add, rem\rangle)$$

This captures that in the quiescent state the object offers its two methods, and that it returns to the quiescent state when execution of a method invocation finishes. As mentioned earlier the declaration of the class Q is encoded as a replicator:

$$[\![\texttt{class Q}]\!] \stackrel{\text{def}}{=} (new)\, (\nu\, add, rem)\, !\, \overline{new}(add, rem).\, \mathrm{Obj_Q}\langle new, add, rem\rangle$$

Here '!' is the replication operator from [8] which may be eliminated in favour of an agent constant. We can think of $!P$ as $P \mid P \mid P \mid \cdots$. Finally the encoding of a program is obtained by composing the representations of the class declarations with the agent corresponding to the root object, and restricting on all link names except those which correspond to the interface of the program.

4 Priority Queues

In this section we describe the analysis of the priority queue classes Q and Q′ outlined in the Introduction. Due to the limitation of space, very many details are omitted. Suppose *prog* is a program in which the definition of Q appears. Then as outlined in the preceding section its encoding $[\![prog]\!]$ is an abstraction of the form

$$(\ldots)\,(\nu \ldots)\,(\text{Classes}\langle\ldots\rangle \mid \text{RootObject}\langle\ldots\rangle)$$

where Classes is the composition of the replicators which represent the class definitions and RootObject is an agent representing the root object of the system. $[\![prog]\!]$ is abstracted on the link names which correspond to the interface of the program; all other link names are restricted. Consider the context

$$\mathcal{C}[\bullet] \equiv (\ldots)\,(\nu\, new, \ldots)(\bullet\langle new\rangle \mid \text{OtherClasses}\langle\ldots\rangle \mid \text{RootObject}\langle\ldots\rangle)$$

derived by encoding the incomplete program obtained by omitting the definition of Q; the '•' indicates where the abstraction $[\![\texttt{class Q}]\!]$ should be placed to recover $[\![prog]\!]$. We assume that the name *new* on which the encoding of $[\![\texttt{class Q}]\!]$ is abstracted is not part of the program's interface and is thus restricted. Let *prog′* be obtained from *prog* by replacing the definition of Q with that of Q′ and each mention of Q by mention of Q′. The behavioural equivalence of *prog* and *prog′* may be expressed as follows:

Theorem 1 $\mathcal{C}[[\![\texttt{class Q}]\!]] \approx \mathcal{C}[[\![\texttt{class Q}']\!]]$.

A sketch of the proof follows. If the root object is an instance of one of the priority queue classes then since the only class new instances of which may be created by the root object is the class of which it is an instance, the result follows easily. To prove the theorem in the more interesting case, two principal results are required. The first is an addition to the collection of useful properties of replicators presented in [8].

Theorem 2 Suppose that $R \equiv !(\nu\tilde{y})\overline{x}(\tilde{y}).P$, $R' \equiv !(\nu\tilde{y})\overline{x}(\tilde{y}).P'$ and $(\nu x)(R \mid P) \approx (\nu x)(R' \mid P')$ where x occurs in P and P' only in positive subject position. Then for any S in which x occurs only in positive subject position, $(\nu x)(R \mid S) \approx (\nu x)(R' \mid S)$.

Proof: Assume $(\nu x)(R \mid P) \approx (\nu x)(R' \mid P')$ where R, P, R', P' are as above. Set $E\mathcal{B}E'$ if $E \equiv (\nu x\tilde{z})(R \mid S)$ and $E' \equiv (\nu x\tilde{z})(R' \mid S')$ for some $\tilde{z}$ and $S \stackrel{\cdot}{\approx} S'$ in both of which x occurs only in positive subject position. We claim that $\mathcal{B}$ is a weak bisimulation up to strong bisimilarity $\stackrel{\cdot}{\sim}$. To prove this we first recall from [8] that $(\nu x)(R \mid P) \sim (\nu x)(R \mid (\nu x)(R \mid P))$ and similarly for R' where $\sim$ is strong equivalence. Suppose $E\mathcal{B}E'$ where E and E' are as above and $E \stackrel{\alpha}{\longrightarrow} F$. Then either $F \equiv (\nu x\tilde{w})(R \mid T)$ where $S \stackrel{\alpha}{\longrightarrow} T$, or $F \equiv (\nu x\tilde{z}\tilde{y})(R \mid P \mid T)$ where $S \stackrel{x(\tilde{y})}{\longrightarrow} T$. In the first case $S' \stackrel{\widehat{\alpha}}{\Longrightarrow} T' \stackrel{\cdot}{\approx} T$ so $E' \stackrel{\widehat{\alpha}}{\Longrightarrow} F' \equiv (\nu x\tilde{w})(R' \mid T')$ and $F\mathcal{B}F'$. In the second case $S' \stackrel{x(\tilde{y})}{\Longrightarrow} T' \stackrel{\cdot}{\approx} T$ so $E' \Longrightarrow F' \equiv (\nu x\tilde{z}\tilde{y})(R' \mid P' \mid T')$. Now by the fact noted above, $F \sim G \equiv (\nu x\tilde{z}\tilde{y})(R \mid (\nu x)(R \mid P) \mid T)$ and $F' \sim G' \equiv (\nu x\tilde{z}\tilde{y})(R' \mid (\nu x)(R' \mid P') \mid T')$. Moreover as $(\nu x)(R \mid P) \stackrel{\cdot}{\approx} (\nu x)(R' \mid P')$ and $T \stackrel{\cdot}{\approx} T'$, $G\mathcal{B}G'$. Hence $E' \Longrightarrow F'$ with $F \stackrel{\cdot}{\sim} \mathcal{B} \stackrel{\cdot}{\sim} F'$. So $\mathcal{B}$ is a weak bisimulation up to $\stackrel{\cdot}{\sim}$ and hence if x occurs in S only in positive subject position, $(\nu x)(R \mid S) \stackrel{\cdot}{\approx} (\nu x)(R' \mid S)$. Now if σ is a substitution in which x does not occur then repeating the above argument replacing R, P, R', P' by $R\sigma, P\sigma, R'\sigma, P'\sigma$ we have that if x occurs in T only in positive subject position, $(\nu x)(R\sigma \mid T) \stackrel{\cdot}{\approx} (\nu x)(R'\sigma \mid T)$. Hence if x occurs in S only in positive subject position then for any σ, $((\nu x)(R \mid S))\sigma \equiv (\nu x)(R\sigma' \mid S\sigma') \stackrel{\cdot}{\approx} (\nu x)(R'\sigma' \mid S\sigma') \equiv ((\nu x)(R' \mid S))\sigma$ for some σ' in which x does not occur. Hence $(\nu x)(R \mid S) \approx (\nu x)(R' \mid S)$. □

The second result describes a close relationship between the agents encoding the classes.

Theorem 3 We have

$$(\nu\, new)([\![\texttt{class Q}]\!]\langle new\rangle \mid \mathrm{Obj}_{\texttt{Q}}\langle new,...\rangle) \approx (\nu\, new)([\![\texttt{class Q}']\!]\langle new\rangle \mid \mathrm{Obj}_{\texttt{Q}'}\langle new,...\rangle).$$

To prove Theorem 1, in Theorem 2 we take $R \equiv [\![\texttt{class Q}]\!]\langle new\rangle$, $R' \equiv [\![\texttt{class Q}']\!]\langle new\rangle$ and S the context $\mathcal{C}[\bullet]$ without the restriction on *new* so that $(\nu\, new)(R \mid S) \equiv \mathcal{C}[[\![\texttt{class Q}]\!]]$ and $(\nu\, new)(R' \mid S) \equiv \mathcal{C}[[\![\texttt{class Q}']\!]]$. The condition that *new* occur in $\mathrm{Obj}_{\texttt{Q}}$, $\mathrm{Obj}_{\texttt{Q}'}$ and S only in positive subject position is met: it is straightforward to check from the full definition of the translation $[\![\cdot]\!]$. Furthermore, by Theorem 3 the other condition of Theorem 2 is met in this case. Hence an appeal to Theorem 2 gives the desired result, that $\mathcal{C}[[\![\texttt{class Q}]\!]] \approx \mathcal{C}[[\![\texttt{class Q}']\!]]$.

To prove Theorem 3 is less straightforward than it might appear at first glance. It is fairly easy to *describe* a relation containing the relevant pair of agents which one would expect to be a bisimulation up to $\approx$. But to *prove* that it is so would require an argument taking account of the fact that an indefinite number of new cell-agents may be created, and that each time a new cell is added to the chain, new behaviour becomes possible. To overcome this difficulty we distil from an understanding of the abstract behaviour of the classes a system of equations in agent variables which by virtue of its form is known to have a unique solution up to $\approx$. We then derive abstract descriptions of the behaviour of the agents $\mathrm{Obj}_{\texttt{Q}}$ and $\mathrm{Obj}_{\texttt{Q}'}$ and use these to give inductive proofs that the processes in Theorem 3 are corresponding members of two families which are solutions to this system. See [16] for the proofs and an account of the treatment of run-time errors, an issue which must be addressed in order for the above discussion to be fully accurate. This topic is addressed briefly in the case of the symbol tables at the beginning of the next section.

5 Symbol Tables

In this section we outline the analysis of the relationship between the symbol table classes T and T′ described in the Introduction. As explained there the correspondence between the classes is not a simple one. In particular there is no simple behavioural equivalence between the agents encoding them. Thus the analysis differs markedly from that for the priority queue classes. As mentioned at the end of the previous section we must take account of the possibility of run-time errors. We do this by introducing an empty statement λ and inserting an extra conditional at the head of each method body of class `T`:

```
method Insert(X:NAT, W:A)
    if X=nil then λ
    else if K=nil then ...

method Search(X:NAT)
    if X=nil then λ
    else if K=nil then ...
```

For the `Search` method of class `T′` the alteration is identical to that above while for the `Insert` method we have

```
method Insert(X:NAT, W:A)
    return ;
    if X=nil then λ
    else if K=nil then ...
```

The treatment of run-time errors in the priority queue example is similar. The reason for making these modifications to the class definitions is that in their absence, due to the possibility of a method being invoked with a parameter having an undefined value, the precise relationship between the class definitions would be much less simple. When, for instance, the `Insert` method of a T$'$-node is invoked, the caller is released from the rendezvous before the parameter is examined. In contrast, if a T-node examines the parameter, it does so before returning to the caller. Since, according to the semantics by translation, the process encoding a node may deadlock during examination of an undefined value – an agent of the form $\mathsf{if}(\bot : P, Q)$ has no transitions – it is clear that as a tree structure evolves, significantly different behaviour may be manifest in the two cases. A related issue is the treatment of searches for keys absent from a table. We assume that the result of such a search is a nil reference and use the expression $\overline{r}(\mathsf{nil})$ to represent the return on a link r of a tuple of names corresponding to a nil reference.

We first obtain an abstract description of the behaviour generated by the sequential class T. To describe symbol tables we use the tree expressions given by

$$t \ ::= \ \varepsilon \ \mid \ t_l \lhd \langle k, a\rangle \rhd t_r$$

Here ε represents a tree consisting of a single node with key $\bot$, and in the tree described by $t_l \lhd \langle k, a\rangle \rhd t_r$, the root has key k and value a, if h is a key of the left subtree t_l then $h < k$, and if h is a key of the right subtree t_r then $h > k$. We write $\widehat{tx}$ for the value associated with key x in t (if it exists).

To represent trees of nodes of the class T we introduce agents T_t as follows. Recalling that $\mathrm{Obj}_{\mathtt{T}} : (\mathsf{LINK[T]}, \mathsf{OBJECT[T]})$ where $\mathsf{OBJECT[T]} \equiv \mathsf{METHOD[NAT,A;]}, \mathsf{METHOD[NAT;A]}$ we have $T_t : (\mathsf{LINK[T]}, \mathsf{OBJECT[T]}, (\mathsf{N}, \mathsf{OBJECT[A]})^h, \mathsf{N}^l)$ if t represents a tree with l leaves and h internal nodes, where the linear order of the parameters is related in some natural way to the corresponding tree structure. Let $E \equiv \mathrm{Obj}_{\mathtt{T}}\langle new, ins, srch\rangle$ be the process representing an empty node of class T. Let also $C\langle k, a\rangle \equiv C\langle new, ins, srch, k, a, insl, srchl, insr, srchr\rangle$ be the representation of a node of class T storing an integer key k and with $a : \mathsf{OBJECT[A]}$ representing the reference stored in the variable V and $(insl, srchl)$, resp. $(insr, srchr)$, the reference stored in the variable L, resp. R. Then $T_\varepsilon\langle\widetilde{p}, \bot\rangle \equiv E$ where $\widetilde{p} = new, ins, srch$, and for $t = t_l \lhd \langle x, a\rangle \rhd t_r$,

$$T_t \stackrel{\text{def}}{=} (\widetilde{p}\, x\, a \ldots)(\nu\, \widetilde{q})(T_{t_l}\langle \widetilde{p}_l, \ldots\rangle \mid C\langle x, a\rangle \mid T_{t_r}\langle \widetilde{p}_r, \ldots\rangle)$$

where $\widetilde{q} = insl, srchl, insr, srchr$, $\widetilde{p}_l = new, insl, srchl$ and $\widetilde{p}_r = new, insr, srchr$. We define also $T_t^{\leadsto} : (\mathsf{N}, \mathsf{OBJECT[A]}, \mathsf{LINK[T]}, \mathsf{OBJECT[T]}, (\mathsf{N}, \mathsf{OBJECT[A]})^h, \mathsf{N}^l)$ as follows:

$$T_\varepsilon^{\leadsto} \stackrel{\text{def}}{=} (y\, b\, \widetilde{p} \ldots)\mathsf{if}(y = \bot : T_\varepsilon\langle \widetilde{p}, \bot\rangle, T_{t_0}\langle \widetilde{p}, y, b, \bot, \bot\rangle)$$

where $t_0 = \varepsilon \lhd \langle y, b\rangle \rhd \varepsilon$, and for $t = t_l \lhd \langle x, a\rangle \rhd t_r$,

$$\begin{array}{rll} T_t^{\leadsto} \stackrel{\text{def}}{=} (y\, b\, \widetilde{p} \ldots)\,\mathsf{cond}(& y = \bot & : T_t\langle \widetilde{p}, x, a, \ldots\rangle, \\ & y = x & : T_{t'}\langle \widetilde{p}, x, b, \ldots\rangle, \\ & y < x & : (\nu\, \widetilde{q})(T_{t_l}^{\leadsto}\langle y, b, \widetilde{p}_l, \ldots\rangle \mid C\langle x, a\rangle \mid T_{t_r}\langle \widetilde{p}_r, \ldots\rangle), \\ & \mathsf{else} & : (\nu\, \widetilde{q})(T_{t_l}\langle \widetilde{p}_l, \ldots\rangle \mid C\langle x, a\rangle \mid T_{t_r}^{\leadsto}\langle y, b, \widetilde{p}_r, \ldots\rangle)) \end{array}$$

where $t' = t_l \lhd \langle x, b\rangle \rhd t_r$ and cond abbreviates a nested conditional. Finally define

$$\widetilde{E} \equiv (\nu\, new)(E \mid [\![\mathtt{class\ T}]\!]\langle new\rangle)$$

and define $\widetilde{C}\langle k, a\rangle$, $\widetilde{T}_t$ and $\widetilde{T}_t^{\leadsto}$ similarly as restricted compositions. Then omitting some parameters to improve readability we have the following.

Theorem 4 For any t, $\widetilde{T}_t\langle ...\rangle \approx ins(y,b,r).\,\overline{r}.\,\widetilde{T}_t^{\smile}\langle y,b,...\rangle + srch(x,r).\,\overline{r}(\hat{t}x).\,\widetilde{T}_t\langle ...\rangle$.

This result is proved by finding abstract descriptions of $\widetilde{E}$ and $\widetilde{C}$ and then establishing, using an inductive argument, that the family $\{\widetilde{T}_t, \widetilde{T}_t^{\smile}\}_t$ satisfies the system of equations implicit in the theorem. Note that $\widetilde{T}_\varepsilon\langle \widetilde{p}, \bot\rangle \equiv (\nu\, new)(\mathrm{Obj}_{\mathrm{T}}\langle \widetilde{p}\rangle \mid [\![\texttt{class T}]\!]\langle new\rangle)$ so the theorem gives a convenient description of the behaviour of a newly-created object of class T (in restricted correspondence with a private copy of the replicator representing the class). We now wish to give a corresponding abstract description of the behaviour of $(\nu\, new)(\mathrm{Obj}_{\mathrm{T}'}\langle \widetilde{p}\rangle \mid [\![\texttt{class T}']\!]\langle new\rangle)$.

To describe the behaviour of a tree of T′-nodes we consider augmented tree expressions (t,σ) where t is a tree expression and σ a function which assigns to each node n of the tree described by t a *status* $\sigma(n)$. If n is a leaf then $\sigma(n)$ may be f or of the form (i_r, m, b, r) or the form (s, m, r). The first indicates that the node in inactive ('free'), the second that the `Insert` method with parameters (m,b) and return link r has been invoked but has not yet returned, and the third that the `Search` method has been invoked with parameter m and return link r but that the search has not yet been returned or committed. If n is an internal node with key k then $\sigma(n)$ may additionally be of the form (i,m,b) where $m \neq \bot, k$. This indicates that the `Insert` method with parameters (m,b) has been invoked and returned but not yet passed on to the appropriate child node.

The abstract description of the behaviour of trees of T′-nodes is obtained by focussing on augmented tree expressions (t,σ) in which each method invocation has progressed as far as it can through the tree structure, subject to the constraints imposed by other outstanding invocations. To do this we use the function *fall* on augmented tree expressions which is such that $fall(t,\sigma)$ describes a tree in which the method invocation outstanding at the root node, if there is one, is propagated as far through the tree structure as possible without any other method invocation making further progress.

For n a node of the tree described by an augmented tree expression (t,σ) we let $fall(t,\sigma,n)$ be the augmented tree expression describing the tree obtained by replacing the subtree (t_n,σ_n) rooted at n by $fall(t_n,\sigma_n)$. Now suppose n is a node of the tree described by (t,σ) and let $n = n_1,\ldots,n_p = root(t)$ be the nodes on the path from n to the root of t. Define $(t_1,\sigma_1) = fall(t,\sigma,n_1)$ and for $2 \leq i \leq p$, $(t_i,\sigma_i) = fall(t_{i-1},\sigma_{i-1},n_i)$. Then define $cascade(t,\sigma,n) = (t_p,\sigma_p)$. The function *cascade* is useful in describing the overall progress of method invocations when a search is returned by a node. We then say that (t,σ) is *settled* if for each node n of t, $cascade(t,\sigma,n) = (t,\sigma)$. In giving the abstract description we stay within the domain of settled expressions. We say also that a node n of t is *ready* (to return a search) if for some m and r, $\sigma(n) = (s,m,r)$ and n has key m or key $\bot$. Finally we say (t,σ) is *blocked* if $\sigma(root(t)) \neq f$.

We can now define the agents which provide an abstract description of the behaviour of trees of T′-nodes. Suppose (t,σ) is settled. If (t,σ) is blocked we set $U_{t,\sigma} \stackrel{\mathrm{def}}{=} U^R_{t,\sigma}$ while if (t,σ) is not blocked we set

$$U_{t,\sigma} \stackrel{\mathrm{def}}{=} U^R_{t,\sigma} + ins(y,b,r).\,U_{t',\sigma'} + srch(x,r).\,U_{t'',\sigma''}$$

where $(t',\sigma') = (t, \sigma[(i_r,y,b,r)/root(t)])$ and $(t'',\sigma'') = fall(t,\sigma[(s,x,r)/root(t)])$, and where

$$\begin{aligned} U^R_{t,\sigma} \stackrel{\mathrm{def}}{=} \Sigma\{\overline{r}(a).\,U_{t',\sigma'} \mid\ & \text{for some ready node } n \text{ of } t,\ t(n) = \langle k,a\rangle \text{ and } \sigma(n) = (s,k,r), \\ & \text{and } (t',\sigma') = cascade(t,\sigma[f/n],n)\} \\ + \Sigma\{\overline{r}(\mathsf{nil}).\,U_{t',\sigma'} \mid\ & \text{for some leaf } n \text{ of } t,\ \sigma(n) = (s,k,r) \text{ and} \\ & (t',\sigma') = cascade(t,\sigma[f/n],n)\} \\ + \overline{r}.\,U_{t',\sigma'} & \end{aligned}$$

The last summand is present only if the root node has status (i_r, m, b, r), so that (t, σ) is blocked, and in it (t', σ') is the appropriate expression representing the *fall* of the insertion as far as possible through the tree.

The agent $U^R_{t,\sigma}$ describes the possible returns of method invocations from the tree of T′-nodes described by (t, σ). The first summand describes the returns of searches for keys present in the table, the second the return of searches for keys absent from the table, and the third the return from the root node of an `Insert` method. That this family of agents describes the behaviour of trees of T′-nodes is the substance of the following theorem.

Theorem 5 Setting $(t_0, \sigma_0) = (\varepsilon, [f/root])$ we have

$$(\nu\, new)(\mathrm{Obj}_{\mathrm{T}'}\langle new, ins, srch\rangle \mid [\![\texttt{class T'}]\!]\langle new\rangle) \approx U_{t_0,\sigma_0}.$$

The proof is again by induction and involves finding abstract descriptions of the behaviour of agents encoding nodes. Using these abstract descriptions of the behaviour of $(\nu\, new)(\mathrm{Obj}_{\mathrm{T}}\langle ...\rangle \mid [\![\texttt{class T}]\!]\langle new\rangle)$ and $(\nu\, new)(\mathrm{Obj}_{\mathrm{T}'}\langle ...\rangle \mid [\![\texttt{class T'}]\!]\langle new\rangle)$, we may now describe the relationship between the classes sketched in the Introduction. Let $\widetilde{\mu} = \mu_1, \ldots, \mu_p$ be any sequence of method invocations in which none of the parameters is $\perp$ and which is such that if μ_i is `Search(k)` then for some $j < i$, μ_j is `Insert(k, a)` for some reference `a`. (It is not essential to make these restrictions but doing so allows us to concentrate on the main case). Associated with $\widetilde{\mu}$ are a sequence $\widetilde{\alpha} = \alpha_1, \ldots, \alpha_{2p}$ of actions and a sequence $\widetilde{t} = t_0, \ldots, t_p$ of tree expressions, where $t_0 = \varepsilon$, if μ_i is `Insert(k, a)` then for some r, $\alpha_{2i-1} = ins(k, a, r)$ where a corresponds to `a`, $\alpha_{2i} = \overline{r}$, and $\widehat{t_i} = \widehat{t_{i-1}}[a/k]$, and if μ_i is `Search(k)` then for some r, $\alpha_{2i-1} = srch(k, r)$, $\alpha_{2i} = \overline{r}(\widehat{t_{i-1}}k)$ and $t_i = t_{i-1}$.

By Theorem 4, $(\nu\, new)(\mathrm{Obj}_{\mathrm{T}}\langle ...\rangle \mid [\![\texttt{class T}]\!]\langle new\rangle) \stackrel{\widetilde{\alpha}}{\Longrightarrow}\approx \widetilde{T}_{t_p}\langle ...\rangle$ is the unique computation of the T-node (with private replicator) determined by $\widetilde{\mu}$. Theorem 5 describes the possible computations of the T′-node (with private replicator) determined by $\widetilde{\mu}$. They are labelled by the permutations of the sequence $\widetilde{\alpha}$ allowed by the evolving tree structure. Note that in the encoding of a program in which T′ occurs, whenever an object-agent invokes a search in a T′-table it creates a private link for the return of the result and suspends its activity until a result is returned to it. This private name is handled (in the precise sense of [8]) by only one T′-node agent at any one time. Moreover the restriction operator ensures that the return links associated with distinct searches are not confused and that the result of a search is returned to the appropriate invoking object.

6 Conclusion

The process calculus apparatus employed to analyse the examples has been limited to purely equational reasoning. This approach is certainly appropriate for establishing the equivalence of the encodings of the priority queue classes and there are no doubt other instances of equivalence-preserving transformations whose soundness could be established by analogous reasoning. In the symbol tables example the equational techniques are useful in giving abstract descriptions of the behaviour generated by the two classes. From these the precise, intricate relationship between the classes may be seen more clearly. Remaining within the process calculus framework, one might also investigate the modal and temporal properties of the agents encoding the classes, and examine whether the use of a higher-order calculus would ease the analysis. It would also be of interest to examine alternative approaches to the question of the soundness of the kind of program transformations considered here. For instance, can the metric-space semantics of [2] be used to tackle this kind of question? And can the Hoare-style proof system for partial correctness of POOL programs presented in [4], or extensions of it, be used fruitfully is studying such problems? This paper has addressed

the issue of the soundness of program transformations through two concrete examples. It is hoped that it may contribute to the development of techniques for proving soundness of general transformation rules for parallel object-oriented programs.

References

[1] P. America, J. de Bakker, J. Kok and J. Rutten, Operational semantics of a parallel object-oriented language, in *Conference Record of the 13th Symposium on Principles of Programming Languages*, 194–208 (1986).

[2] P. America, J. de Bakker, J. Kok and J. Rutten, Denotational semantics of a parallel object-oriented language, *Information and Computation*, 83, 152–205 (1989).

[3] P. America, Issues in the design of a parallel object-oriented language, *Formal Aspects of Computing*, 1, 366–411 (1989).

[4] F. de Boer, Reasoning about dynamically evolving process structures, PhD thesis, Free University of Amsterdam (1991).

[5] C. Jones, Constraining interference in an object-based design method, in Proc. TAP-SOFT'93, Springer-Verlag LNCS vol. 668, 136–150 (1993).

[6] C. Jones, A pi-calculus semantics for an object-based design notation, in Proc. CONCUR'93, Springer-Verlag LNCS vol. 715, 158–172 (1993).

[7] R. Milner, **Communication and Concurrency**, Prentice Hall, 1989.

[8] R. Milner, The polyadic π-calculus: a tutorial, in **Logic and Algebra of Specification**, F. Bauer et al. (eds.), Springer-Verlag (1993).

[9] R. Milner, J. Parrow and D. Walker, A calculus of mobile processes, I and II, *Information and Computation* 100, 1–40 and 41–77 (1992).

[10] R. Milner, J. Parrow and D. Walker, Modal logics for mobile processes, in Proc. CONCUR'91, Springer-Verlag LNCS vol. 527, 45–60 (1991).

[11] B. Pierce and D. Sangiorgi, Typing and subtyping for mobile processes, in Proc. IEEE LICS'93, 376–385, Computer Society Press (1993).

[12] D. Sangiorgi, Expressing mobility in process algebras: first-order and higher-order paradigms, PhD thesis, University of Edinburgh (1992).

[13] D. Walker, π-calculus semantics for object-oriented programming languages, in Proc. TACS'91, Springer-Verlag LNCS 526, 532–547 (1991).

[14] D. Walker, Objects in the π-calculus, to appear in *Information and Computation* (1992).

[15] D. Walker, Process calculus and parallel object-oriented programming lanaguges, in Proc. International Summer Institute on Parallel Computer Architectures, Langauges and Algorithms, Prague, July 1993 (Computer Society Press, to appear).

[16] D. Walker, Algebraic proofs of properties of objects, technical report, University of Warwick (1994).