Adding Type-Bound Actions to Action-Oberon

Ralph Back, Martin Büchi, Emil Sekerinski

Department of Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, 20520 Turku, Finland
{backrj, mbuechi, esekerin}@abo.fi

Turku Centre for Computer Science
TUCS Technical Report No 66
November 1996
ISBN 951-650-891-X
ISSN 1239-1891

Abstract

We extend the Action-Oberon language for executing action systems with type-bound actions. Type-bound actions combine the concepts of type-bound procedures (methods) and actions, bringing object orientation to action systems. Type-bound actions are created at runtime along with the objects of their bound types. They permit the encapsulation of data and code in objects. Allowing an action to have more than one participant gives us a mechanism for expressing *n*-ary communication between objects. By showing how type-bound actions can logically be reduced to plain actions, we give our extension a firm foundation in the Refinement Calculus.

Keywords: action systems, Action-Oberon, objects, concurrency, synchronization, type-bound actions

TUCS Research Group
Programming Methodology Research Group

1 Introduction

Action-Oberon extends Oberon-2 [21] with actions for modeling parallel and distributed computations. The extension is based on the theory of action systems [5] and was proposed by Back and Sere [7] and implemented by Hedman [13]. An action system is a parallel or distributed program where parallel activity is described in terms of guarded actions. Enabled actions are executed atomically in a nondeterministic order to model parallelism. Atomicity of actions guarantees that a parallel execution of an action system gives the same results as a sequential nondeterministic execution in Action-Oberon (serializability).

Action-Oberon supports only plain actions, which may optionally be replicated over a constant range of integers. Plain actions describe updates to the variables visible in the module in which they are declared. The new type-bound actions combine the principles of type-bound procedures (methods) and actions. Bound to one or more types, they are created dynamically whenever an object of a bound type is created. They describe updates to the objects to which they are bound, as well as to the variables visible in their declaration module. Järvinen and Kurki-Suonio first proposed the marriage of object-oriented concepts and action systems in the DisCo language [15]. Their basic idea is the same as ours, but the actual definitions differ greatly due to the form of object orientation, the base language, the underlying logic, and the interpretation.

We have built an environment, in form of an Action-Oberon to Oberon-2 compiler and an associated runtime/simulation system under Oberon/F [20], which allows extended Action-Oberon programs to be executed. The environment helps to debug specifications and isolate critical properties worth formal proofs. Our environment is a play field for action systems and not an attempt to add concurrency to the Oberon language and/or system.

Section 2 presents the Action-Oberon base language, section 3 explains the type-bound actions, section 4 elaborates on the deactivation of type-bound actions and the deallocation of objects, section 5 discusses inheritance of type-bound actions, section 6 provides a foundation for type-bound action in the Refinement Calculus, and section 7 draws a conclusion and gives an overview of related and further work.

2 Action-Oberon base language

Oberon-2 [21] is the successor of Pascal and Modula-2. Modula-2 adds modularization to Pascal. Oberon-2 extends Modula-2 with object-oriented concepts in form of type extension on record types (subtyping/inheritance) as well as type-bound procedures (methods). Oberon-2 has been chosen as a base language because of its simplicity and its similarity to previously used ad-hoc notations of action systems.

Action-Oberon [7] adds actions and guarded procedures to Oberon-2. Action

```
MODULE OneFish;

  CONST
    height=10;
    width=20;

  VAR
    x, y: INTEGER;
    right, up: BOOLEAN;

  ACTION MoveRight
    WHEN right & (x#width);
  BEGIN INC(x)
  END MoveRight;

  ACTION MoveLeft
    WHEN ~right & (x#0);
  BEGIN DEC(x)
  END MoveLeft;

  ACTION Bounce Right
    WHEN right & (x=width);
  BEGIN right:=FALSE
  END BounceRight;

  ACTION BounceLeft
    WHEN ~right & (x=0);
  BEGIN right:=TRUE
  END BounceLeft;

  ACTION MoveUp (* code *)
  ACTION Move Down (* code *)
  ACTION BounceUp (* code *)
  ACTION BounceDown (* code *)

BEGIN
  x:=0; y:=0; right:=TRUE; up:=TRUE
END OneFish.
```

Figure 1: Screen saver OneFish

systems are represented by Oberon modules. All actions are executed repeatedly in a loop until all actions are disabled. Selection of enabled actions is nondeterministic and is not bound to a fairness pledge. The nondeterminism is demonic, in the sense that there is no way of influencing which action is chosen. The simulation environment provides, however, the possibility to install one's own scheduler or to manually select actions. The module body contains the initialization. Parallel composition of action systems corresponds to loading several modules into memory at once. Actions from all loaded modules are executed in one big loop; that is, they may be interleaved in any order. The combined action system can only terminate when none of the loaded modules contains an enabled action.

Actions are declared like procedures without parameters. The guard of an action is given as a (side-effect free) boolean expression. Omitting the guard corresponds to an always enabled guard.

Throughout the paper we use the example of a fish screen saver. In our first version OneFish (Figure 1), a single fish swims around the screen. The fish's current position is given by Cartesian coordinates x (horizontal axis) and y (vertical axis). The fish is either moving right (right = TRUE) or left and either up (up = TRUE) or down. When it reaches a border it changes direction. Note that the lack of a fairness assumption means that the fish might only move along one axis, although the guard for moving along the other axis is infinitely often true.

Our screen saver is an example of an action system which never terminates. Hence, our interest does not lie in its input/output behavior, but in its possible traces (sequences of states).

```
MODULE ManyFishes;

  CONST
    many=5;
    height=10;
    width=20;

  VAR
    x, y: ARRAY many OF INTEGER;
    right, up: ARRAY many OF BOOLEAN;
    k: INTEGER;

  ACTION MoveRight(i: 0..many-1)
    WHEN right[i] & (x[i]#width);
  BEGIN INC(x[i])
  END MoveRight;

  ACTION MoveLeft(i: 0..many-1)
    WHEN ~right[i] & (x[i]#0);
  BEGIN DEC(x[i])
  END MoveLeft;

  ACTION Bounce Right(i: 0..many-1)
    (* code *)
  ACTION BounceLeft(i: 0..many-1)
    (* code *)
  ACTION MoveUp(i: 0..many-1)
    (* code *)
  ACTION Move Down(i: 0..many-1)
    (* code *)
  ACTION BounceUp(i: 0..many-1)
    (* code *)
  ACTION BounceDown(i: 0..many-1)
    (* code *)

BEGIN
  FOR k:=0 TO many-1 DO
    x[k]:=k; y[k]:=k;
    right[k]:=TRUE; up[k]:=TRUE
  END
END ManyFishes.
```

Figure 2: Screen saver ManyFishes

Actions may optionally be replicated over one or more constant ranges of integers, generating a number of similar actions. We use this mechanism to add more fishes to our screen saver in the next version ManyFishes (Figure 2). The declaration ACTION MoveRight(i: 0..many-1) generates an action for each i between 0 and many-1. The replicator i can be used like a constant in the guard and body of the action.

Like actions, procedures may be protected by a guard [6]. If the evaluation of the guard of an action or the execution of its body would lead to a call of a disabled procedure, the action is considered to be disabled. Note that no waiting for the guard to become true takes place, as is common with monitors or semaphores.

3 Type-bound actions

Being useful in certain cases, replication is awkward at best when we have to replicate several actions over the same range, as in figure 2. It provides no encapsulation of data and code within a single entity, (pseudo-) dynamic creation of new entities is cumbersome and error-prone — even if we added dynamically extendible arrays and variable replication ranges.

Thus, borrowing from the concept of object orientation of type-bound procedures, we add type-bound actions to Action-Oberon. The declaration ACTION (f: Fish) MoveRight leads to the dynamic creation of an action for each object of type

```
MODULE OOFishes;

  CONST
    height=10;
    width=20;
    many=5;

  TYPE
    Fish=POINTER TO FishDesc;
    FishDesc=RECORD
        x, y: INTEGER;
        right, up: BOOLEAN
    END;

  VAR
    fi: Fish;
    k: INTEGER;

  ACTION (f: Fish) MoveRight
    WHEN f.right & (f.x#width);
  BEGIN INC(f.x)
  END MoveRight;

  ACTION (f: Fish) MoveLeft
    WHEN ~f.right & (f.x#0);
  BEGIN DEC(f.x)
  END MoveLeft;

  ACTION (f: Fish) Bounce Right (* code *)
  ACTION (f: Fish) BounceLeft (* code *)
  ACTION (f: Fish) MoveUp (* code *)
  ACTION (f: Fish) Move Down (* code *)
  ACTION (f: Fish) BounceUp (* code *)
  ACTION (f: Fish) BounceDown (* code *)

  PROCEDURE NewFish(VAR nf: Fish;
      x, y: INTEGER; right, up: BOOLEAN);
  BEGIN
    NEW(nf); nf.x:=x; nf.y:=y;
    nf.right:=right; nf.up:=up
  END NewFish;

BEGIN
  FOR k:=0 TO many-1 DO
    NewFish(fi, k, k, TRUE, TRUE)
  END
END OOFishes.
```

Figure 3: Screen saver OOFishes

Fish that we create. The bound variable f is called participant and may be used like a variable in the action. It corresponds to the receiver (self) of a type-bound procedure. Figure 3 gives our screen saver using type-bound actions.

Suppose we want to program some special behavior if two fishes meet. We can do this with ACTION (f1, f2: Fish) Meet (Figure 4). We allow an action to have several participants, i.e. f1 and f2, of various types. An instance of meet will be created at runtime for each tuple of fishes, including double instantiations of the same fish. Hence, we have to explicitly strengthen the guard of Meet if we do not desire fishes to meet themselves (no aliasing). Actions with n participants lend themselves to symmetrically express n-ary communication, which is difficult in most other formalisms for $n > 2$.

Action names are treated as global identifiers of their modules. The complete EBNF for actions is given in figure 5.

If we add action Meet to OOFishes (Figure 3), it is not guaranteed that Meet will be executed whenever two fishes are at the same coordinates because the fishes' move actions are also enabled; their guards would have to be strengthened if desired.

```
ACTION (f1, f2: Fish) Meet
  WHEN (f1.x=f2.x) & (f1.y=f2.y) & (f1#f2);
  VAR baby: Fish;
BEGIN
  (* do something: i.e.
    - change direction
    - create new fish
    - remove one of the fishes *)
END Meet;
```

Figure 4: Type-bound action Meet

We could imagine several behaviors if two fishes meet. We could for example change the direction of one fish or we could have them produce a baby fish by invoking NEW. Without object-orientation, but only plain replication, we would have to extend our data arrays and ranges separately.

4 Deactivation and deallocation

Consider the case where we would want one fish to eat the other. How do we remove the dead fish from our system, that is how do we prevent it from participating in actions and how do we recycle its allocated memory? In Oberon-2, objects may be garbage collected if they are no longer referenced from one of the loaded modules. Having introduced type-bound actions, we cannot simply adopt this condition. Consider the case where we remove the last reference to an object. Should this object still be able to have one of its type-bound actions executed until it is garbage collected? If so, this action could again set a pointer to the object and, herewith, revive it. On the other hand, if the object looses its eligibility to participate in actions with the removal of the last reference, we unnecessarily restrict the independence of our active objects and — in an extendible system where we often don't know the number of references to an object — loose control over the duration of an object's active life cycle. Given the undesirable properties of the 'naturally' extended conditions for garbage collection, we enumerate the possible solutions which preserve pointer safety (no dangling pointers) and summarize their properties in figure 6:

```
Action        =  ACTION [Participants] IdentDef [Replicators] [Guard] ";"
                 DeclSeq [BEGIN StatementSeq] END identifier.
Participants  =  "(" VarDecl {";" VarDecl} ")".
Replicators   =  "(" Repl {";" Repl} ")".
Repl          =  identifier ":" ConstExpr ".." ConstExpr.
Guard         =  WHEN Expr.
```

Figure 5: EBNF of extended action declaration

1. An object may be collected after the last reference to it vanishes. Until then, it is eligible to participate in actions (as above).

2. An object may be collected after the last reference to it vanishes. An unreachable object cannot have one of its bound actions executed (as above).

3. An object may only be garbage collected if it is no longer referenced and none of its bound actions can ever be enabled again. Clearly, the second condition can in practice not be verified; hence, no automatic garbage collection can be implemented.

4. An object o is deallocated with a special command KILL(o). The precondition of KILL(o) is that o is the only reference to the object. As the declaration of type-bound actions is not restricted to their participants' declaration modules (see below), we stand the danger in an extendible system of prematurely killing an object.

5. The eligibility of an object o to participate in actions is removed with a special command DEACTIVATE(o). Meanwhile, all references are kept. An object can be garbage collected, if it has been deactivated and it is no longer referenced. We can interpret this as a special case of 3 where each object has a flag alive which is initially true, added as an implicit conjunct to each action guard, and can only be set to false by invoking DEACTIVATE. Creation and deactivation are not duals, as the latter only revokes an object's active behavior.

We can model any of the above choices in the Refinement Calculus (Section 6). However, the computation model is simpler if an object's eligibility to participate in actions does not depend on it being referenced and the model must not include a reference count.

Going back to our consumed fish example, solutions 4 and 5 let us solve the problem without introducing a liveness flag and the corresponding guards in all actions. To keep the theory simple, make recycling of memory feasible, avoid cluttering of code by explicit flags, and prevent the introduction of aborts, we choose solution 5. If o has already been deactivated DEACTIVATE(o) is skip; DEACTIVATE(NIL) is abort. So far, the loss of duality between creation and destruction and the premature disabling of actions have not caused any problems in our examples.

The existence of both modules and classes (types and associated type-bound procedures/actions) in Action-Oberon provides for more compositionality. Modules are compile-time abstractions which provide for scoping and may contain several classes, the latter providing for extensibility and being a run-time abstractions that defines the structucture and behavior of objects. This separation of concerns allows objects to be bundled to components [22, 20]. In Action-Oberon this gives us more compositionality on the module level by restricting the outside visibility of

Property	1	2	3	4	5
pointer-safety	yes	yes	yes	yes	yes
recycling of memory feasible	yes	yes	no	yes	yes
duality of constructor and destructor	yes	yes	yes	yes	no
manual disabling of actions without explicit flag	no	no	no	yes	yes
revival impossible	no	yes	yes	yes	yes
active lifespan independent of references	no	no	yes	yes	yes
execution model without reference count	no	no	yes	yes	yes
safe deallocation	yes	yes	yes	no	yes
safe disabling of actions	no[1]	no[1]	yes	no[2]	no[2]

[1]Due to dependency on references.
[2]Due to explicit termination with KILL, respectively DEACTIVATE.

Figure 6: Properties of different deallocation schemes for objects with type-bound actions

attributes and methods and still allows for privileged access between more closely related classes.

Unlike type-bound procedures, type-bound actions may be declared in any module where the participant types are visible, with access to the fields (instance variables) according to the Oberon-2 export/import visibility rules. This is needed for defining actions with participants stemming from different modules.

5 Inheritance of type-bound actions

We can add some variety to our aquarium by defining special kinds of fishes. If we create a type Shark as subtype of Fish (Figure 7), sharks have all actions of normal fishes bound to them plus possibly additional ones, i.e. ShowTeeth.

We might also want to override (redefine, extend) some actions for sharks, i.e. have sharks become hungrier whenever they move and eat another fish they meet when they are hungry enough. If we permitted overriding, we could create an action ACTION (s: Shark; f: Fish) Meet. This would immediately raise two problems. Consider a fish φ and a shark σ. Should we now have two actions Meet, the original one for (φ, σ) and the redefined for the reversed tuple (σ, φ) (Figure 8 a)?

Secondly, this would require multiple dispatch, as actions can have several participants. Assume that we also defined a subtype Piranha of Fish and ACTION (f: Fish; p: Piranha) Meet. Which action body would we choose for a tuple of a shark

```
TYPE
  Shark=POINTER TO SharkDesc;
  SharkDesc=RECORD (Fish)
    hunger: INTEGER
  END;

  ACTION (s: Shark) ShowTeeth;
  BEGIN (* show teeth *)
  END ShowTeeth;
```

Figure 7: Subtype Shark

and a piranha (Figure 8 b)? Requiring each combination of (normal) fishes, sharks and piranhas (Figure 8 c) to be defined is not practical in a modular system where the different subtypes can be defined in different modules and where modules are not statically linked. The solution of Chambers and Leavens for multiple dispatch of methods [10], which requires a designated topmost module and flags errors of other modules when compiling this module, is against the spirit of open systems and separate extension. While the first problem could be partly solved by introducing a special notation for symmetrical participants, the second one has no solution which is orthogonal to separate extension. Hence, we do not permit overriding of actions.

We could allow overriding for actions with only one participant as this does not require multiple dispatch and, therefore, does not create the problems described here. For simplicity's and orthogonality's sake we do not. Instead, we simulate overriding of type-bound actions by using overriding of type-bound procedures. Figure 9 shows how we override MoveRight by replacing the body with a single call to a type-bound action. In the implementation of MoveRight for Shark, s.MoveRightˆ is a super-call to the overridden type-bound procedure which in our case is a call to MoveRight for Fish. Instead of replacing the complete guard with a call to a type-bound procedure, we choose in this example to explicitly state the

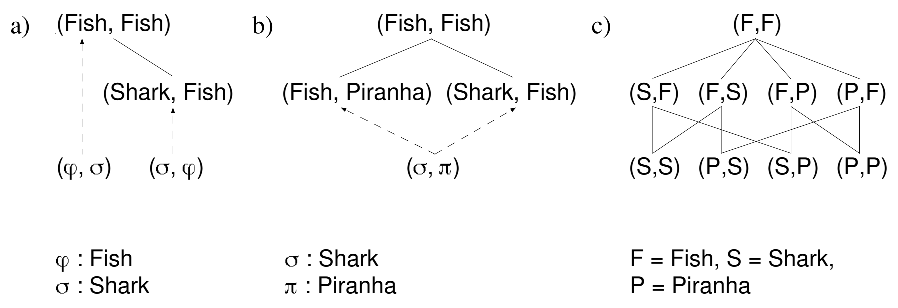

Figure 8: Problems of overriding actions with more than one participant

```
PROCEDURE (f: Fish) MoveRight;
BEGIN INC(f.x)
END MoveRight;

PROCEDURE (s: Shark) MoveRight;
BEGIN s.MoveRight^; INC(s.hunger)
END MoveRight;

PROCEDURE (f: Fish) WantToMove(): BOOLEAN;
BEGIN RETURN TRUE
END WantToMove;

PROCEDURE (s: Shark) WantToMove(): BOOLEAN;
BEGIN RETURN s.hunger<10
END WantToMove;

ACTION (me: Fish) MoveRight
  WHEN me.right & (me.x#width) & me.WantToMove;
BEGIN me.MoveRight
END MoveRight;
```

Figure 9: Simulating overriding for actions with only one participant

conjuncts common to all extensions. This form of overriding requires explicit provisions to be made, i.e. introducing the constant function WantToMove which we override for our lazy sharks which don't move if they are too hungry. However, there is also the argument that unless the designer has arranged for it, reuse and overriding never work in practice anyhow [16]. Overriding of type-bound procedures and dynamic type tests give all that is needed.

Another approach would be not to inherit actions, i.e. a Shark would not automatically have all actions defined for Fish. This would be orthogonal to polymorphism as actions are not called explicitly; it would not create the danger of invoking undefined actions as the method deletion mechanism of Smalltalk does. Because inheritance of type-bound actions has proved to be desirable in practice, we have adopted it in Action-Oberon.

6 A semantics for type-bound actions

In this section we give a formal semantics to type-bound actions by reducing them to plain actions. We have four syntactical levels to express an action system: Action-Oberon with type-bound actions, Action-Oberon without type-bound actions, action systems, and the Refinement Calculus. The translation from Action-Oberon without type-bound actions to action systems is given by Back and Sere [7], the translation from action systems to the Refinement Calculus and the mathematical treatment of action systems is due to Back, Kurki-Suonio and von Wright [5, 3, 9, 8]. Figure 10 shows these three levels for a sample program. A plain

MODULE M	*init*;	*init*;
	do	while $G1 \vee G2$ do
ACTION A1	$G1 \to S1$	$[G1];S1 \sqcap [G2];S2$
WHEN G1;	$[]\ G2 \to S2$	od
BEGIN S1	od	
END A1;		Sugared Refinement Calculus
ACTION A2		
WHEN G2;		*init*;
BEGIN S2		$(\mu X \bullet ([G1];S1 \sqcap [G2];S2);X \sqcap \texttt{skip});$
END A2;		$[\neg(G1 \vee G2)]$
BEGIN init		
END M.		
Action-Oberon	action system	Refinement Calculus

Figure 10: Translation of an Action-Oberon module

action ACTION A WHEN G; BEGIN S END A; translates to the guarded statement $G \to S$ which is only enabled if G holds. An Action-Oberon module with several actions translates to a do loop with a demonic choice between the actions. On the Refinement Calculus level, a predicate in square brackets denotes a guard, which is equivalent to `skip` if the predicate holds and `magic` otherwise. The meet ($\sqcap$) denotes the demonic choice, and μ stands for the least fixpoint. The refinement calculus level only applies to the input/output, but not to the trace semantics.

A type-bound action is defined as follows: There is only one instance of each type-bound action. The guard implicitly stands for 'there exists a tuple that satisfies the stated guard' and the first statement demonically (nondeterministically) chooses one such tuple. We first use this intuition to sketch a simulation of type-bound actions in Action-Oberon without type-bound actions. Thereafter, we also provide a direct translation to action systems by formalizing this idea.

We keep a data structure of all types and their inheritance relation as well as a set of all active objects of each type. We turn each type-bound action into a plain action of the same name by replacing the guard by a traversal function which returns true, iff it finds a tuple of possible participants from the respective sets of active participants. We add an additional first statement, which demonically chooses one possible tuple and assigns it to the participant variables. This is, up to optimizations, how the current implementation works.

Alternatively, we can map type-bound actions directly to their action system equivalent. Using the Refinement Calculus typed higher-order logic, we denote the record types by $R_1, \ldots, R_n$ and the corresponding pointer types by $P_1, \ldots, P_n$. Records are represented as tuples and type extension ([width] subtyping/inheritance) corresponds to tuple extension. We model the heap as a partial function (functional relation) from pointers to records and use a boolean flag for each record to indicate whether an object is active:

$$heap : P_1 + \ldots + P_n + \mathtt{NIL} \nrightarrow (R_1 + \ldots + R_n) \times Bool$$

Initially, *heap* is empty. To manipulate the sum types, we define families of injection functions $\mathtt{in_i} : P_i \to P_1 + \ldots + P_n + \mathtt{NIL}$ (also for records), projection functions $\mathtt{out_i} : P_1 + \ldots + P_n + \mathtt{NIL} \to P_i$, and discriminator functions $\mathtt{is_i} : R_1 + \ldots + R_n \to Bool$. $\mathtt{NIL}$ is the one-element type $\{\mathtt{nil}\}$ for modeling NIL pointers, for which we get the following invariant: $\mathtt{in_{NIL}}\,\mathtt{nil} \notin \mathtt{dom}\, heap$. We also define $\mathtt{is}^*_\mathtt{i}$ to be the transitive closure of $\mathtt{is_i}$ with respect to the direct subtype/inheritance relation ($R_j <: R_i \stackrel{\text{def}}{=} R_j$=RECORD (R_i) ... END):

$$\mathtt{is}^*_\mathtt{i}\, r \quad \stackrel{\text{def}}{=} \quad (\mathtt{is_i}\, r) \vee (\exists j . R_j <: R_i \wedge \mathtt{is}^*_\mathtt{j}\, r)$$

In fact, $\mathtt{is}^*_\mathtt{i}$ corresponds to Oberon's IS statement: r IS $R_i \equiv \mathtt{is}^*_\mathtt{i}\, r$. Furthermore, we define $\mathtt{dom}$ to return the domain of a partial function. We define $\mathtt{fst}$ and $\mathtt{snd}$ as the first and second projections of a tuple, and $\mathtt{prj_k}$ as the kth projection.

We express NEW as a family of predicate transformers. For pointer p of Action-Oberon type P_i we get:

$$\begin{aligned} NEW_i(p) \quad \stackrel{\text{def}}{=} \quad & (\sqcap x : P_i | \mathtt{in_i}\, x \notin \mathtt{dom}\, heap \bullet p := \mathtt{in_i}\, x); \\ & heap := heap \cup \{(p \mapsto (\mathtt{in_i}\, 0_{R_i}, T))\} \end{aligned}$$

The only change to NEW with respect to Oberon-2 is the initialization of the boolean flag indicating that the object is active: First we demonically — indicated by the meet — choose a free location in the heap and assign it to p and then augment the partial heap function by the mapping from p to the 0-record with unspecified values of the referenced type. To exhibit the fact that DEACTIVATE only changes the activity flag of the referenced record, we give it in terms of a relational override $\lhd\!\!\!\!+$ defined for relations r, s:

$$r \lhd\!\!\!\!+ s \quad \stackrel{\text{def}}{=} \quad \{(x \mapsto y) \in r | x \notin \mathtt{dom}\, s\} \cup s$$

$$\begin{aligned} DEACTIVATE(p) \quad \stackrel{\text{def}}{=} \quad & \{p \in \mathtt{dom}\, heap\}; \\ & heap := heap \lhd\!\!\!\!+ \{(p \mapsto (\mathtt{fst}\ (heap\, p), F))\} \end{aligned}$$

DEACTIVATE asserts — indicated by the braces — that p is a valid pointer and then sets the activity flag of the referenced record to false. We can now give the translation for a type-bound action:

ACTION (p: P_i) A WHEN G; BEGIN S END A;

$$\begin{aligned} \stackrel{\text{def}}{=} \quad & (\sqcap q \in \mathtt{dom}\, heap | \mathtt{is}^*_\mathtt{i}\ (\mathtt{fst}\ (heap\ q)) \wedge \mathtt{snd}\ (heap\ q) \bullet \\ & \mathtt{begin}\ \mathtt{var}\ p : P_1 + \ldots + P_n + \mathtt{NIL}; p := q; G \to S\ \mathtt{end}) \end{aligned}$$

The quantified variable q is a logical variable, whereas p, which is only visible within the block bracketed by `begin` and `end`, is a program variable which may also appear on the left hand side of assignments. The action is enabled if there is at least one possible participant for which G holds. More formally, the guard and the body of a predicate transformer S are defined as $gA = \neg \mathtt{wp}(A, \mathit{false})$ and $sA = \{gA\};A$. This allows us to view a type-bound action as a guarded statement. The generalization to actions with more than one participant is straightforward.

A statement containing a record field access is the sum of predicate transformers over all record, respectively pointer types. Let for example p be a pointer to a record whose kth field is x: INTEGER and let h: INTEGER. The assignment h:=p.x in Action-Oberon then corresponds to the following predicate transformer:

$$
\begin{array}{ll}
 & \mathsf{h{:=}p.x} \\
\stackrel{\text{def}}{=} & \{p \in \mathtt{dom}\, \mathit{heap}\}; \\
 & (h := \mathtt{prj_k}\,(\mathtt{out_1}\,(\mathtt{fst}\,(\mathit{heap}\, p)))) + \\
 & \quad \ldots + \\
 & \quad (h := \mathtt{prj_k}\,(\mathtt{out_n}\,(\mathtt{fst}\,(\mathit{heap}\, p)))) + \\
 & \quad \mathtt{abort}
\end{array}
$$

The properties of the summation operator for predicated transformers was explored by Back and Butler [4], based on Nauman's [19] and Martin's [18] category theoretical considerations. Late binding of type-bound procedures is modeled analogously.

Hence, we can use type-bound actions in our extended Action-Oberon programs and use their unsugared form for reasoning in the Refinement Calculus or use the above correspondence to give a sugared form of the relevant inference rules.

Interestingly, there are two other 'natural' explanations for type-bound actions which give identical semantics:

1. Whenever an object is created, the set of actions is augmented by the corresponding bound actions. Dually, when an object's eligibility to participate in actions is revoked, the associated actions are removed. This model is possible as the action system formalism does not require the set of actions to be constant or finite.

2. We start with an infinite number of objects in a special not-yet-created state and an infinite number of corresponding actions the guards of which test that all participant are in the created state. Creating an object corresponds to changing its state to created.

7 Conclusion

We have extended Action-Oberon with type-bound actions encapsulating both data and actions in objects by combining the principles of object-orientation and action systems. Actions with n participants provide a symmetrical mechanism to express

n-ary communication between objects. We found the explicit DEACTIVATE command to have the most desirable properties of the solutions for disabling an object's bound actions and recycling its allocated memory. Due to the conflict between multiple dispatch and independent extensibility, overriding of type-bound actions is prohibited. Overriding of type-bound procedures and dynamic type tests provide for selective overriding. By reducing type-bound actions to plain actions they are given a formal semantics in the Refinement Calculus framework which allows for concise mathematical reasoning.

Our work on Action-Oberon was inspired by the original Action-Oberon and DisCo. Other related work includes Unity, its successor Seuss, and a number of frameworks for active objects.

DisCo [15] first introduced type-bound actions. DisCo's concept of 'inheritance' corresponds to having a field of the inherited type in our terminology; hence, there is no overriding. DisCo does not have any type-bound procedures, making it lack any form of dynamic binding. The language contains no loops or recursion. Guarded procedures do not exist.

In Unity [11] 'actions' are restricted to (quantified) multiple deterministic assignments, and the set of actions must be finite and constant; on the other hand, it has fairness and progress properties. We are not aware of anything like type-bound actions in Unity. Due to the lack of nondeterminism in assignments and the restriction to a finite constant set of 'actions' none of our explanations for type-bound actions could be applied in Unity.

Seuss [12] gives the notion of boxes which correspond to our object types and clones which are instantiations thereof. Boxes have local variables (non-exported instance variables), actions (type-bound actions), and procedures (type-bound procedures) which may also be partial (guarded). The set of clones is static. By also requiring all actions to terminate, Seuss can provide fairness. As all actions reside within one clone, there is no possibility to create an action with more than one participant. As in our model, an action calling a disabled procedure fails. However, the disabled procedure can still change the state of the callee, by executing the code associated to a 'negative alternative'.

Formalisms and languages for active objects are characterized by different objects executing in parallel. New objects can be created dynamically. Objects communicate by message passing, which is the only way to have an object do something. Generally, objects do not contain any actions. Triggered procedures in object-oriented databases constitute a notable exception to this rule; however, the triggered (guarded) procedures are usually executed as part of the transaction setting off the trigger. Due to the lack of actions, the condition for garbage collection is simple reachability, respectively knowledge of an object's mail address. N-ary communication between objects can generally not be expressed in a symmetric fashion. Hewitt's actor model [14, 1] was the first formal model of active objects. More recently, CCS and the π-calculus have been used to give a semantics to members of the POOL family [2, 17, 23].

We are interested in increasing our collection of examples manifesting the usefulness of type-bound actions. Simulation of mechanical systems is a very promising application. We are also interested in adding fairness and/or priorities to Action-Oberon. Additionally, we are trying to show the value of object-encapsulation for atomicity refinement of actions.

We would like to thank Kaisa Sere, Philipp Heuberger and Wolfgang Weck for a number of clarifying and fruitful discussions on the topic of this paper.

References

[1] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[3] R. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings*. LNCS 430, Springer Verlag, 1990.

[4] R. Back and M. Butler. Exploring summation and product operators in the refinement calculus. Technical Report on Computer Science & Mathematics, Ser. A. No 152, Åbo Akademi, 1994.

[5] R. Back and R. Kurki-Suonio. Distributed co-operation with action systems. *ACM Transactions on Programming Languages and Systems 10:513–554*, 1988.

[6] R. Back and K. Sere. Action systems with synchronous communication. In *IFIP TC 2 Working Conference on Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 107–126. Elsevier, 1994.

[7] R. Back and K. Sere. From action systems to modular systems. In *Proceeding of Formal Methods Europe '94*. LNCS 873, Springer Verlag, 1994.

[8] R. Back and J. von Wright. Trace refinement of action systems. In *CONCUR 94*, pages 367–384. LNCS 836, Springer Verlag, 1994.

[9] R. J. R. Back. Refinement of parallel and reactive programs. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series, pages 73–92. Springer-Verlag, 1993.

[10] Craig Chambers and Gary T. Leavens. Type checking and modules for multi-methods. Technical Report #95-19, Iowa State University, August 1995.

[11] K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison Wesley, 1988.

[12] K.M. Chandy. A discipline of multiprogramming. Available from ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z, or from the PSP group's homepage http://www.cs.utexas.edu/users/psp/, June 1996.

[13] Eric J. Hedman. Action-Oberon. Master's thesis, Åbo Akademi University, 1995.

[14] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3), 1977.

[15] H.-M. Järvinen and R. Kurki-Suonio. DisCo specification language: Marriage of action and objects. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 142–151, Arlington, Texas, 1991. IEEE Computer Society Press.

[16] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June 1:2 1988.

[17] Cliff B. Jones. A π-calculus semantics for an object-based design notation. In *Proceedings of CONCUR 93*, pages 158–172. LNCS 715, Springer Verlag, 1993.

[18] C.E. Martin. *Preordered Categories and Predicate Transformers*. PhD thesis, Programming Research Group, Oxford University, 1991.

[19] D.A. Naumann. *Two-Categories and Program Structure: Data Types, Refinement Calculi, and Predicate Transformers*. PhD thesis, University of Texas at Austin, 1992.

[20] Oberon microsystems, Inc. *Oberon/F*. http://www.oberon.ch, 1995.

[21] P. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming 12:179–195*, 1991.

[22] Clemens A. Szyperski. Import is not inheritance — Why we need both: Modules and classes. In *Proceedings of ECOOP 92*, pages 19–32. LNCS 615, Springer Verlag, 1992.

[23] D.J. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**