Towards a Semantics of Emerald Expressed in Map Theory

Klaus Grue

December 5, 1994

Abstract

This paper presents a semantics for a subset of an object oriented, concurrent and distributed language with Emerald as the chosen example. The concurrent and distributed aspects posses particular problems, and the paper focuses on these aspects. In particular the paper covers parallelism in which a created process can continue to run after the creating process halts. The semantics is presented in a continuation passing style in map theory.

Map theory is a foundation of mathematics with the same expressive power (w.r.t. consistency) as set theory, but it builds upon λ-calculus and functions instead of logic and sets. It allows unlimited recursion which, combined with its expressive power, makes it ideal for expressing semantics of programming languages.

The paper serves three purposes: First, it gives a semantics for a subset of Emerald, which can be scaled up to cover all of Emerald. Second, it shows how map theory can be used in definitions of semantics. Third, it shows how distribution and parallelism can be modelled.

Contents

1 Introduction

Emerald [6,7,16,17,15,18,19,20,25] is a novel object oriented, concurrent and distributed language. It has been adopted for teaching of first year students at the University of Copenhagen. This paper presents a semantics of a subset of Emerald. The concurrency and distribution aspects of Emerald are difficult to express in traditional frameworks, so the present paper focuses on these aspects.

The reader is assumed to know Emerald as described in [15], but Section 5 will be a sufficient introduction to many readers.

The semantics of programming languages are traditionally defined axiomatically [10,11,9], in lambda calculus [3,5,21], in set theory [1,2,4,23] or in other suitable theories [14,24]. The axiomatic approach has the benefit that it highlights a number of basic properties of the language from which all other properties can be deduced. A drawback of this method is that consistency of the basic properties has to be established somehow. Another drawback is that it is necessary to develop a new style of proof and a new collection of metatheorems for each new axiomatisation.

In the denotational approach, the semantics is defined within some, previously established theory. That theory can be a general purpose one like set theory, it can be tailored to computer science like lambda calculus, or it can be even further tailored to semantics like CCS and CSP [14,24]. This paper uses the denotational approach, but expresses the semantics in map theory [8] which is a general purpose competitor to set theory. The semantics makes heavy use of the unlimited recursion available in map theory.

A semantics of a programming language is by nature a definition. For that reason, the present paper contains a definition and no theorems or proofs. For some of the above mentioned approaches, it is necessary or at least convenient to accompany each semantics by proofs such as consistency, soundness, completeness or congruence proofs. This is not necessary or relevant in the present

setting, where the only question is whether or not the semantics expresses the intentions of the language designers.

Emerald supports parallelism in that objects can create new objects and the processes of the new objects will run in parallel with the creating object. Furthermore, the processes of the new objects may continue to run after the creating object halts or ceases to exist like in the fork calculus [10,11,9]. This is difficult to describe in traditional calculi of parallelism like CCS [24] and CSP [14].

The semantics presented in this paper is intended as an aid in discussions of the semantics of Emerald. Such discussions may be between language designers, language implementors and language users. Furthermore, semantics can be useful in language standardisation.

A completely different use of semantics is in automated proof systems. A semantics written for such a system should be stated in a style that suits that particular system. However, the present paper aims at a human audience. Readers who know Emerald in advance may use the semantics for sharpening his or her intuitive understanding of Emerald. Other readers may use it as an example of how one can model Emeralds kind of parallelism and distribution in a continuation passing style.

2 Overview of the paper

Section 3 describes the problems that Emerald and thereby the semantics have to tackle and Section 4 gives a brief introduction to map theory. Section 5 gives a brief introduction to Emerald and at the same time defines the subset of Emerald that will be described in the semantics.

Sections 6–13 define the semantics. The semantics is defined by a function $\mathcal{E}(p, n, d)$ which, given a program p, input n and "decision list" d returns a trace, i.e. a list of all states that the program traverses when executed. The list is finite if p terminates in finite time. The output from the program is recorded in the trace. It is possible to define process congruences based on this output, but that is deferred to further work (Section 14). Emerald has non-determinism, so a program may result in different traces in different runs. All possible traces of program p for input n are generated by $\mathcal{E}(p, n, d)$ when d ranges over the set D of all possible "decision lists".

Section 6 represents the abstract syntax of the chosen subset of Emerald in map theory. Sections 7 and 8 introduce two auxiliary concepts. Section 9 describes how program states are represented in map theory. A trace is a list of program states. Section 10 defines $\mathcal{E}(p, n, d)$ on basis of two functions "trace" and "initstate" that are defined in Sections 11 and 12, respectively. The function "initstate" is far more complex than "trace" and is described last. The function "initstate" computes the initial state of the program, and that initial state contains all information about all future behaviour of the program represented as

"continuations". The function "trace" merely has to apply these continuations iteratively. The function "initstate" uses the function tr to translate abstract syntax into continuations. That function is defined in Section 13

The function $\mathcal{E}$ is not computable by machine, and no attempts have been made to make the computable parts of $\mathcal{E}$ "efficient" in any sense.

3 Overview of the semantics

Emerald is a language whose programs execute on networks of processors. Each processor in the network is called a 'node'. At any given time, each node may be 'up' (available) or 'down' (crashed or off duty). Each node has an input and an output byte stream which are the only means for input to and output from the program (at least in the semantics to be defined).

At any given time, an Emerald program consists of a collection of objects. Objects may contain local data, may export operations, may contain a local process, and may contain a local monitor [13]. Objects may move between nodes, either under program control or controlled by the system.

Emerald supports distribution in that objects may float between nodes (processors) in a network. The nodes may go down and come up unpredictably. Objects running on a node are lost when the node goes down, but objects may 'checkpoint' themselves. A checkpoint is a copy of the object on disk, and if an object is lost due to node crash, then the object may be restarted by loading its latest checkpoint.

In the semantics, each object will be identified by a 'token', i.e. a unique identifier whose internal structure is irrelevant (the given semantics uses natural numbers as tokens). Token allocation will be modelled by 'token trees' as described in Section 7

Emerald allows concurrency in that local processes of distinct objects run in parallel. The parallelism may be genuine on systems with more than one node. In Emerald, concurrency leads to non-determinism in that the order of execution of parallel processes may affect the end result. In the semantics, this non-determinism will be modelled by 'decision lists' as described in Section 8.

The operations of an object may be monitored or non-monitored. At most one monitored operation of an object may execute at a time whereas any number of non-monitored operations may execute in parallel. The monitors of the objects are the only means for synchronisation. In particular, there are no means for locking internal variables.

4 Overview of map theory

Map theory [8] is an extension of untyped lambda calculus [3]. It has lambda-abstraction $\lambda x.\mathcal{A}$ and functional application $f(x)$. Like Lisp [22,26] it has atoms, but contrary to Lisp it only has one atom. That atom is called T and, by

convention, represents as diverse objects as truth, the empty set and the empty list. In the semantics it will also be used to represent absence. Map theory has an if–then–else construct and allows unlimited use of recursion. Non-termination is denoted $\perp$.

Map theory has several equal signs $=$, $\hat{=}$, $\doteq$ etc. They all denote equality and the distinctions are irrelevant for the present paper.

In addition, map theory has an epsilon operator [12]. As an example, $\varepsilon x{\in}\mathbf{R}.x^2{\hat{=}}2$ denotes a real number x such that $x^2 = 2$. Hence, $\varepsilon x{\in}\mathbf{R}.x^2{\hat{=}}2$ could denote $\sqrt{2}$ or $-\sqrt{2}$. The epsilon operator makes a deterministic but unspecified choice, so $\varepsilon x{\in}\mathbf{R}.x^2{\hat{=}}2$ is either always $\sqrt{2}$ or always $-\sqrt{2}$. $x^2{\hat{=}}-1$ holds for no real numbers x, so $\varepsilon x{\in}\mathbf{R}.x^2{\hat{=}}-1$ cannot find a proper x. In this case, $y = \varepsilon x{\in}\mathbf{R}.x^2{\hat{=}}-1$ is still a real number, e.g. 0, but that real number y of course cannot satisfy $y^2 = -1$. The epsilon operator is not computable by machine.

The universal quantifier $\forall$, existential quantifier $\exists$, set membership $\in$ and logical connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ and $\neg$ are all definable in map theory. In addition, all concepts definable in set theory are also definable in map theory. Examples could be the set $\mathbf{N}$ of natural numbers, the set $\mathbf{R}$ of real numbers, and the class On of ordinals. All theorems of set theory are provable in map theory.

A pairing construct (x, y) and head and tail operations z^h and z^t are definable such that $(x, y)^h = x$ and $(x, y)^t = y$ hold for all x and y. As examples, $(\mathsf{T}, \perp)^h = \mathsf{T}$ and $(\mathsf{T}, \perp)^t = \perp$. The let–construct let $x = \mathcal{A}$ in $\mathcal{B}$ is defined as usual to stand for $(\lambda x.\mathcal{B})(\mathcal{A})$. For more information on map theory, consult [8].

5 Limitation of task

This paper will define the semantics of a small but interesting subset of Emerald. The emphasis will be on parallelism and distribution. The strong typing aspect of Emerald is left out since typing and static semantics is well understood. The subset is rather useless for actual programming. Rather, the subset is chosen to show the interesting aspects of the semantics in such a way that the semantics can be scaled up to cover all of Emerald.

The rest of this section lists the facilities of Emerald and specifies what is included in and excluded from the Emerald subset to be described. For further information on Emerald consult [15].

Assignment is included in order to describe how state changes occur. Only assignment to simple variables (variables that hold objects) and operation return parameters is included. Assignment to record fields and array elements is not described (and such assignments are actually syntactic sugar in Emerald). Multiple assignments are excluded from the subset for simplicity. The abstract syntax for assignments in the subset is

Assignment ::= Id Exp

where Exp is an expression to evaluate and Id is the destination of the assignment. Constants are excluded.

As mentioned, static semantics and types will not be covered here. In the semantics, all variables can hold objects of arbitrary type and variables need not be declared.

Sequence (begin–end) is included, and the abstract syntax is

Sequence ::= Stmt_1 Stmt_2

During execution, Stmt_1 is executed first and Stmt_2 is executed when Stmt_1 completes. Sequences with more than two elements can be obtained by nested sequencing. Selection (if) and iteration (loop, exit, for) are excluded because they just are more complex examples of sequencing. Local variables in begin–end blocks are excluded because handling of local variables is covered by operation return variables.

In the semantics, objects can float freely between nodes which is the hard part of distribution to describe. The facilities fix, unfix, refix, move and visit for controlling object movement are excluded.

Operation invocation is included. For simplicity, however, all operations take exactly one argument and return exactly one return value. The abstract syntax of an invocation is

Call ::= Exp_1 Id Exp_2

where the value of Exp_1 identifies the object, Id identifies the operation and Exp_2 is the argument.

Assert statements are excluded.

Monitors are included in order to describe synchronisation. Conditions (Hoare monitor conditions) and the wait, signal and awaiting facilities are excluded since they just add complexity.

Checkpoints are supported. For simplicity, checkpoints are parameterless and checkpoints are stored on the local node. If the node goes down, then the checkpoint will be recovered when the node comes up again. For simplicity, any object is checkpointed when it is created so that objects are never lost completely, even if their node crashes before they can make a checkpoint themselves. If an object checkpoints itself and then floats to another node, then the checkpoint stays behind in the semantics to be given. Since checkpoints are parameterless, their abstract syntax is

Checkpoint ::=

Return is included, return–and–fail is not. The abstract syntax of return is

Return ::=

The Primitive Statement (loophole to lower levels) is excluded.
Object creation is included. The abstract syntax of an object creation is

Create ::= OpList_1 OpList_2 Stmt_1 Stmt_2 Stmt_3

where OpList_1 lists the monitored operations, OpList_2 lists the non-monitored operations, Stmt_1 is the initialisation code, Stmt_2 is the recovery code that is executed when the object is recovered from a checkpoint, and Stmt_3 is the process of the object. Note that the initialisation code is never executed in case the local node goes down immediately after object creation. For simplicity, the OpList's and Stmt's of the object creation cannot refer to variables of the creating object. Lots of Emerald type declaration facilities and compilation hints are excluded from the object creation construct above.

Built-in objects like **integer** and **Boolean** will not be described in the semantics.

The abstract syntax of expressions are

Exp ::= Var | Call | Create

where

Var ::= Id

so an expression is either a variable reference, an operation invocation or an object creation. The abstract syntax of statements is

Stmt ::= Assignment | Sequence | Checkpoint | Return

The abstract syntax of operation definitions is

Op ::= Id_1 Id_2 Id_3 Stmt

where Id_1 is the name of the operation, Id_2 is the name of the argument, Id_3 is the name of the return variable and Stmt is the statement to be executed on invocation. An OpList is a list of Op's.

An Emerald program defines a list of objects whose processes start up in parallel when the program starts. For simplicity, a program in the semantics will be a single object specified by a Create construct (but this single object may of course start arbitraryly many objects to run in parallel). Emerald facilities for separate compilation are excluded.

6 Abstract syntax

The abstract syntax of the chosen subset is represented as follows in map theory:

$$
\begin{array}{lcl}
\text{Var(id)} & \doteq & (0, \text{id}) \\
\text{Call}(\text{exp}_1, \text{id}, \text{exp}_2) & \doteq & (1, \text{exp}_1, \text{id}, \text{exp}_2) \\
\text{Create}(p, q, r, s, t) & \doteq & (2, p, q, r, s, t) \\
\text{Assign}(\text{id}, \text{exp}) & \doteq & (3, \text{id}, \text{exp}) \\
\text{Seq}(\text{stmt}_1, \text{stmt}_2) & \doteq & (4, \text{stmt}_1, \text{stmt}_2) \\
\text{Check} & \doteq & (5, 0) \\
\text{Return} & \doteq & (6, 0) \\
\text{Op}(\text{id}_1, \text{id}_2, \text{id}_3, \text{stmt}) & \doteq & (\text{id}_1, \text{id}_2, \text{id}_3, \text{stmt})
\end{array}
$$

An OpList is represented as a list of Op's in map theory.

As an example, if x, y and z are variables represented by Var(1), Var(2) and Var(3), respectively, and if f is an operation represented by the number 4, then $x := y.f(z)$ in Emerald is represented by the abstract syntax

$$\text{Assign}(1, \text{Call}(\text{Var}(2), 4, \text{Var}(3)))$$

which equals

$$(3, 1, (1, (0, 2), 4, (0, 3)))$$

7 Token trees

In the semantics, allocation of tokens is modelled by 'token trees'. A token tree is an infinite, binary, root labelled tree whose labels are distinct objects. Figure 1 shows a token tree. The operations for token trees in map theory read tt.create, tt.root, tt.head, tt.tail and tt.default.

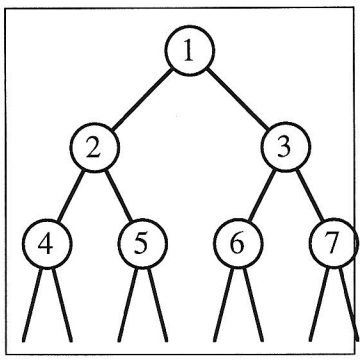

Figure 1: Token tree

$$
\begin{array}{lcl}
\text{tt.create(root, head, tail)} & \doteq & \text{(root, head, tail)} \\
\text{tt.root(tree)} & \doteq & \text{tree}^{h} \\
\text{tt.head(tree)} & \doteq & \text{tree}^{th} \\
\text{tt.tail(tree)} & \doteq & \text{tree}^{tt} \\
\text{tt.default} & \doteq & \text{d}(1) \\
\text{d}(n) & \doteq & \text{tt.create}(n, \text{d}(2 \cdot n), \text{d}(2 \cdot n + 1))
\end{array}
$$

Map theory is not object oriented and has no particular syntax in that it allows notational freedom (which, however, is slightly restricted to allow machine manipulation, but the details are left out here where map theory is used in the same way that mathematicians use set theory). Above, 'tt.root' is just a function name that contains a dot. The 'tt' in the name abbreviates 'transition tree'. Here are some theorems that follow from the definitions:

$$
\begin{array}{lcl}
\text{tt.root(tt.create}(r, h, t)) & = & r \\
\text{tt.head(tt.create}(r, h, t)) & = & h \\
\text{tt.tail(tt.create}(r, h, t)) & = & t \\
\text{tt.root(tt.default)} & = & 1 \\
\text{tt.root(tt.head(tt.tail(tt.default)))} & = & 6
\end{array}
$$

Figure 1 shows the transition tree tt.default. This and any other token tree can be used for allocating unique tokens. If an object P has a token tree T, needs a unique token and then forks a new object Q, then the object may use tt.root(T) as the unique token, it may use tt.tail(T) for further allocations of unique tokens, and it may pass tt.head(T) to Q so that Q may also allocate unique tokens. Token trees are easy and efficient to implement in practice, but are mainly used here as a theoretical convenience.

8 Decision lists

During execution of a program there will at most stages be several possible next actions. If a number of processes inside objects are running, then any one of the processes may perform the next step (unless queued in a monitor). Furthermore, each node may go down or come up, and the system may decide to move any object to any node.

This non-determinism is modelled by 'decision lists' in the semantics. A decision list is an infinite list of tokens with the fairness property that any finite sequence of tokens occurs somewhere in the sequence. The set D of all decision lists is given by

$$D = \{x \in \mathbf{N}^{\omega} | \forall n \in \mathbf{N} \forall y \in \mathbf{N}^{n} \exists m \in \mathbf{N}.\text{subseq}(x, m, n) \hat{=} y\}$$

The definition says that x is a decision list if x is an infinite list of tokens (i.e. natural numbers) and any finite sequence y of tokens occurs somewhere in

x. The natural number n is the length of y and the natural number m is the position of y in x.

The semantics of Emerald will be a function $\mathcal{E}$, where $\mathcal{E}(p, n, d)$ denotes the 'trace' of the Emerald program p when executed on the network n using decision list d. The meaning and encoding of programs, networks and traces are given later. The decision list d controls which alternative of all possible alternatives is chosen at any stage in the computation. Each alternative is identified by a unique token, and an alternative is only chosen if it occurs next in the decision list. If no alternative is chosen, then the next step will be a 'no operation' step.

As an example of use, the expression

$$\forall d{\in}D.\text{finite}(\mathcal{E}(p, n, d))$$

is true if the program p is guaranteed to terminate in finite time on the network n. As another example,

$$\forall n{\in}\mathbf{N}\exists d{\in}D.\neg\text{finite}(\mathcal{E}(p, n, d))$$

states that the program p might loop indefinitely on any network of processors.

9 System states

At any time during execution, an Emerald program p will have a state s. This state will be modelled by a function in the semantics, and that function will have a somewhat complex structure, which is described in the following. The state s will not only describe the state of the program itself, but also the state of the processors on which it is running and s will contain a decision list that will control the behaviour of the program whenever there is a choice between several possibilities.

The integers between -1 and -21 (inclusive) will be denoted Nodestate, Up, Down, Input, Output, Object, Location, Val, Cont, Done, Fail, MonState, TokenTree, Ops, Recover, CheckLoc, CheckVal Decision, Parm, Result, and Env, respectively.

If an Emerald program has state s at any given time, then s is going to be interpreted as follows:

$$s(\text{Nodestate}, n)$$

denotes the state of processor number n on the network. The state is either Up or Down (i.e. $s(\text{Nodestate}, n) = \text{Up}$ or $s(\text{Nodestate}, n) = \text{Down}$). Processors are numbered from 1 and up, and there may by finitely or infinitely many processors. The number of processors is constant during execution of a program (but each processor may go down and come up arbitrarily). If there is no processor number n on the network, then $s(\text{Nodestate}, n) = \mathsf{T}$ where T is an object of map theory that by convention represents as diverse things as truth, the empty set, the empty list and absence. Here and below, T represents absence.

$s(\text{Input}, n)$

denotes the yet unread bytes of the input stream of processor n. The chosen subset of Emerald has no means for reading or writing, but such means are easy to model. ($s(\text{Input}, n) = \mathsf{T}$ if n is not a node since T represents absence).

$s(\text{Output}, n)$

accumulates the bytes sent to the output stream of processor n.

$s(\text{Object}, b)(\text{Location})$

denotes the node at which object b is located. The node is identified by an integer.

$s(\text{Object}, b)(\text{Val})(i)$

contains the value of variable i in object b. A 'value' is always an object, and objects are represented by integers.

$s(\text{Object}, b)(\text{Cont})$

contains a function that describes how this object would like to change the state next time it gets the opportunity. Implicitly, this 'continuation' defines the behaviour of the object in all future.

If $c = s(\text{Object}, b)(\text{Cont})$, if m is a natural number, and if $(v, e, s') = c(b, m, s)$, then s' will be the state of the system after object b has performed one atomic action. The object may have a non-deterministic choice between several possibilities, and these possibilities are enumerated by m.

If $v = \text{Done}$, then the process of b has completed its execution. If $v = \text{Fail}$, then b has experienced a failure (such as division by zero). In all other cases, $v(e)$ is a new continuation to be installed in $s(\text{Object}, b)(\text{Cont})$.

The value of e is a 'local environment', and v is a 'parameterised continuation' which is parameterised by this local environment. The two values v and e are returned separately in order to be able to express sequencing of statements.

Continuations and parameterised continuations will be defined recursively and apply to expressions as well as statements. Statements are treated as expressions that can loop indefinitely or have values Done and Fail. Expressions, in comparison, can loop indefinitely, have the value Fail or have a natural number as value. In the latter case, that natural number identifies the object which is the value of the expression.

The local environment e is quite simple. When executing an operation, it contains an argument value $a = e(\text{Parm})$, a return value $r = e(\text{Result})$, and the local environment $e' = e(\text{Env})$ of the calling operation.

$s(\text{Object}, b)(\text{MonState})$

contains the state of the monitor of object b. If $f = s(\text{Object}, b)(\text{MonState})$ and if $f(i) = b'$ for some natural number i, then object b' is number i in the queue to get access to the monitor. If $f(i) = \mathsf{T}$, then there is no-one in queue position i. Whenever an object is queued by the monitor, it immediately gets a position in the queue, but it needs not get the first empty position, so objects that are queued later may come in front of b'. However, scheduling will be fair because each object gets a finite-numbered position in the queue to start with and moves up in the queue each time another object is dequeued.

$s(\text{Object}, b)(\text{TokenTree})$

contains the token tree from which the object allocates unique tokens. The root of the token tree may be equal to the number that identifies the object itself, so the root cannot be used as a fresh token. Token trees of distinct objects have no tokens in common.

$s(\text{Object}, b)(\text{Ops}, p)$

contains the operation p in object b represented as a continuation.

$s(\text{Object}, b)(\text{Recover})$

contains the continuation to be installed at recovery from a checkpoint.

$s(\text{Object}, b)(\text{CheckLoc})$

identifies the node at which the latest checkpoint of the object resides (identified by a number).

$s(\text{Object}, b)(\text{CheckVal})$

contains the value of $s(\text{Object}, b)(\text{Val})$ at the time of the last checkpoint. Hence, $s(\text{Object}, b)(\text{CheckVal})$ contains all variable values to be used at recovery.

$s(\text{Decision}, 0)$

contains the yet unused portion of the decision list that controls the program whenever there is a choice between several possibilities.

10 Definition of $\mathcal{E}$

As mentioned in Section ??, the semantics of the chosen subset of Emerald is described by the function $\mathcal{E}$. It is defined as follows:

$\mathcal{E}(p, n, d) \doteq \text{trace}(\text{initstate}(p, n, \text{tt.default}, d))$

The arguments p, n and d stand for program, node description and decision list, respectively. The program p has to be the abstract syntax of a Create construct. The decision list d has to be a decision list. When d varies over all possible decision lists, $\mathcal{E}(p, n, d)$ will vary over all possible behaviours of p.

The node description n has to be a list of input streams, i.e. a list of byte streams. The first byte stream in the list will become the input byte stream of node number one and so on. The number of byte streams (which may be infinite) determines the number of nodes on the network. Each byte stream may be finite or infinite.

The function "initstate" constructs the initial state of the system on basis of p, n and d, and uses tt.default as token tree (any token tree will do).

The function "trace" returns the list of all states that the system will traverse. That list will be finite if all objects halt or fail eventually.

Among other, the initial state will contain continuations that describe the behaviour of the system in all future, so the function trace merely has to apply these continuations suitably. Hence, "trace" is much simpler than "initstate" and will be described first.

11 Evolution

The value of trace(s) is a list whose first element is the state s and whose remaining elements are all future states. If s represents a halted system, then there are no future states:

$$\text{trace}(s) \doteq (s, \text{if halted}(s) \text{ then } \mathsf{T} \text{ else trace}(\text{step}(s)))$$

halted(s) is true if s denotes a state where all processes are done or failed or absent:

$$\text{halted}(s) \doteq \forall b{\in}\mathbf{N}.s(\text{Object}, b)(\text{Cont}) \in \{\text{Done}, \text{Fail}, \mathsf{T}\}$$

The function step(s) takes a state s and returns the next state the system will be in. step(s) consumes three elements of the decision list of s and uses the first one of them to decide whether the step is going to be used for an object movement, a node crash/recovery or a program step.

$$\begin{array}{l}\text{step}(s) \doteq \\ \quad \text{let } d = s(\text{Decision}); s' = (s\langle\text{Decision}\rangle := d^{ttt}) \text{ in} \\ \quad\quad \text{if } d^h \hat{=} 1 \text{ then move}(d^{th}, d^{tth}, s') \text{ else} \\ \quad\quad\quad \text{if } d^h \hat{=} 2 \text{ then crash}(d^{th}, s') \text{ else exec}(d^{th}, d^{tth}, s')\end{array}$$

This definition locally defines d to be the decision list of s. The terms d^h, d^{th} and d^{tth} denote the first, second, and third element of this list, and d^{ttt} denotes the remainder of the list when the first three elements are removed.

The term $(f\langle x\rangle := y)$ denotes a function $g = \lambda z.\text{if } z \hat{=} x \text{ then } y \text{ else } f(z)$ for which $g(z) = f(z)$ for all z except that $g(x) = y$. Hence, s' above denotes the state s in which the three first elements of the decision list of s is removed. The definition of $f\langle x\rangle := y$ above would be inefficient if it were executed on a computer, but that is irrelevant here.

If b and n are integers that identify an object and a node, respectively, and if s is a state, then $s' = \text{move}(b, n, s)$ is a new state in which object b has moved to node n provided that both node n and the current residence of b are up. In all other cases, move returns s unmodified.

$$
\begin{array}{l}
\text{move}(b, n, s) \doteq \\
\quad \text{if } s(\text{Nodestate}, s(\text{Object}, b)(\text{Location})) \hat{=} \text{Up} \wedge s(\text{Nodestate}, n) \hat{=} \text{Up} \\
\quad\quad \text{then } s\langle \text{Object}, b\rangle\langle \text{Location}\rangle := n \\
\quad\quad \text{else } s
\end{array}
$$

The function first tests that both nodes exist and are up. If not, then s is returned unmodified. If they are, then a state is returned that merely differs from s in that the location of object b is set to n. According to this semantics, an object may move even if it is queued in a monitor and even after its process has terminated.

If b and m are integers, if b identifies an object and if s is a state, then $\text{exec}(b, m, s)$ is the state that is reached when b performs one atomic action as specified by the continuation of object b. If that action results in the object being queued in a monitor, then m is used to determine the position in the queue.

$$
\begin{array}{l}
\text{exec}(b, m, s) \doteq \\
\quad \text{if } s(\text{Nodestate}, s(\text{Object}, b)(\text{Location})) \hat{=} \text{Down then } s \text{ else} \\
\quad\quad \text{let } c = s(\text{Object}, b)(\text{Cont}); (v, e, s') = c(b, m, s) \text{ in} \\
\quad\quad\quad \text{let } c' = \text{if } v \in \{\text{Done}, \mathit{Fail}\} \text{ then } v \text{ else } v(e) \text{ in} \\
\quad\quad\quad\quad s'\langle \text{Object}, b\rangle\langle \mathit{Cont}\rangle := c'
\end{array}
$$

As can be seen, this definition does not say much about the behaviour of the object. Rather, exec simply applies the continuation c to the current state s, so the behaviour of the object is entirely hidden in c. In the first place, the continuation c is set up by initstate which is why initstate is complicated. See the description of $s(\text{Object}, b)(\text{Cont})$ in Section 9 for a description of the parameterised continuation c and the continuation c' above.

If n identifies a node and s is a state, then $\text{crash}(n, s)$ is the state after n has changed status. If n is up, then it goes down and if n is down then it comes up. When n goes down, all objects on n are lost. The semantics specifies that all lost objects are recovered immediately, but the recovery will not take effect before the residence node of the checkpoint is up. No recovery actions are made when nodes come up since the recovery has been handled already. This is of course not a faithful description of what really happens, but has the same effect.

$$\begin{array}{l}\text{crash}(n,s) \doteq \\ \quad \text{if } s(\text{Nodestate},n) = \text{Down then } s\langle \mathit{Notedate}, n\rangle := \text{Up else} \\ \quad\quad \text{if } s(\text{Nodestate},n) = \text{Up then GoDown}(n,s) \text{ else } s\end{array}$$

GoDown(n,s) lets node n go down and recovers all objects on it.

$$\begin{array}{l}\text{GoDown}(n,s) \doteq \\ \quad \text{let } s' = s\langle \text{Nodestate}, n\rangle := \text{Down in} \\ \quad\quad \text{let } B = \{b{\in}\mathbf{N} | s(\text{Object},b)(\text{Location}){\hat{=}}n\} \text{ in} \\ \quad\quad\quad \text{RecoverSet}(B,s')\end{array}$$

RecoverSet(B,s') recovers the set B of objects. It continually picks out an object of B and recovers it until B is empty. RecoverSet uses the ε-operator of map theory to pick out elements of B.

$$\begin{array}{l}\text{RecoverSet}(B,s) \doteq \\ \quad \text{if } B{\hat{=}}\emptyset \text{ then } s \text{ else} \\ \quad\quad \text{let } b = \varepsilon x.x \in B, B' = B \setminus \{b\} \text{ in} \\ \quad\quad\quad \text{RecoverSet}(B', \text{RecoverOne}(b,s))\end{array}$$

RecoverOne(b,s) recovers object b.

$$\begin{array}{l}\text{RecoverOne}(b,s) \doteq \\ \quad \text{let } s' = s\langle \text{Object}, b\rangle\langle \text{Location}\rangle := s(\text{Object},b)(\text{CheckLoc}) \text{ in} \\ \quad\quad \text{let } s'' = s'\langle \text{Object}, b\rangle\langle \text{Val}\rangle := s(\text{Object},b)(\text{CheckVal}) \text{ in} \\ \quad\quad\quad \text{let } s''' = s''\langle \text{Object}, b\rangle\langle \text{Cont}\rangle := s(\text{Object},b)(\text{Recover}) \text{ in } s'''\end{array}$$

12 The initial state

The function initstate(p,n,t,d) takes a program p expressed in abstract syntax, a list n of input byte streams, a token tree t, and a decision list d as input and returns an initial state. It uses initdecision to install the decision list d in the state, initnode to install the nodes of the system, and initobj to install the object expressed by p.

$$\begin{array}{l}\text{initstate}(p,n,t,d) \doteq \\ \quad \text{initobj}(p,t,\text{initnode}(n,1,\text{initdecision}(d)))\end{array}$$

initdecision(d) returns a state containing the decision list d. It does so by adding d to the empty state T.

$$\text{initdecision}(d) \doteq \mathsf{T}\langle \text{Decision}, 0\rangle := d$$

initnode(n,n',s) creates nodes with numbers $n', n'+1, n'+2, \ldots$ with input byte streams taken from n. It creates finitely many nodes if n is finite and infinitely many otherwise. All created nodes are up, which is of no importance since the decision list can specify that some (possibly all) nodes go down before execution starts.

$$
\begin{array}{l}
\text{initnode}(n, n', s) \doteq \\
\quad \text{if } n \hat{=} \mathsf{T} \text{ then } s \text{ else} \\
\quad\quad \text{let } s' = s\langle \text{Nodestate}, n' \rangle := \text{Up in} \\
\quad\quad\quad \text{let } s'' = s'\langle \text{Input}, n' \rangle := n^h \text{ in} \\
\quad\quad\quad\quad \text{let } s''' = s''\langle \text{Output}, n' \rangle := \mathsf{T} \text{ in} \\
\quad\quad\quad\quad\quad \text{initnode}(n^t, n' + 1, s''')
\end{array}
$$

Programs expressed in abstract syntax are translated to continuations by the function tr. The program to be executed is an object expressed by an object creation construct p. The function initobj creates a dummy object identified by the number 0, translates p to a parameterised continuation using tr and executes the continuation in the environment of the dummy object. This creates the object as a side effect, and the resulting state is returned by initobj. The only thing needed from the dummy object is the token tree t.

$$
\begin{array}{l}
\text{initobj}(p, t, s) \doteq \\
\quad \text{let } s' = s\langle \text{Object}, 0 \rangle \langle \text{TokenTree} \rangle := t \text{ in} \\
\quad\quad \text{let } (v, e, s'') = \text{tr}(p, 0, \mathsf{T}, \mathsf{T})(\mathsf{T})(0, \mathsf{T}, s') \text{ in } s''
\end{array}
$$

In the definition, the two occurrences of 0 refers to the dummy object. The first 0 states that the creation occurs lexically inside dummy object. The second 0 states that the create construct is called by the process of the dummy object. The three occurrences of T are not used. The first and second T identify the argument and return parameters of the operation in which the create construct is located, the third T is the local environment of the create construct and the fourth T determines how the create construct would be queued if it entered a monitor.

13 Translation

The function $\text{tr}(p, b, a, r)$ takes a program p and returns a parameterised continuation c. The parameter b identifies the object that lexically contains p. If p occurs inside an operation definition, then a and r identify the argument and return parameter, respectively. The value of r affects assignment because assignment to r affects the local environment (i.e. the stack) whereas other assignments affect the global state (the local environment merely contains argument and return parameters since local declarations are omited from the chosen subset of Emerald). The value of a affects the translation of variable references since a has to be looked up in the local environment whereas all other variables are looked up in the global state (for simplicity, the return parameter cannot be referenced).

The parameterised continuation c takes an environment e, an object b', a monitor control parameter m and a state s as arguments and returns a new parameterised continuation v, a new local environment e' and a new state s'. More precisely, suppose that

$$(c', e', s') = \mathrm{tr}(p, b, a, r)(e)(b', m, s)$$

holds. Suppose p is a program fragment that occurs lexically inside object b and inside an operation with argument a and return parameter r. Further suppose this program fragment is executed in local environment e and global state s and that the execution is part of execution of the process of object b'. In this case, (c', e', s') represents the state after execution of one atomic action of p. If that atomic action involves entering a monitor, then m controls where the process is placed in the monitor queue.

e' is the new local environment and s' is the new global state after execution of one atomic action. The value of v is somewhat more complicated. If p is a statement, then v equals Done if the atomic action leads to normal termination of p. In this case, e' is going to be passed on to whatever follows p. v equals Fail if a failure (such as division by zero) is experienced. In this case, Fail has to propagate back to object b', whose process halts. Otherwise, v is a new parameterised continuation which describes what the process of b' will do next time it gets the opportunity to do something. If p is an expression, then the above description applies except that if p terminates normally then v equals the value of the expression rather than the value Done. The value of an expression is always an object and v identifies this object.

The program fragment p can be a variable reference, an operation invocation, an object creation, an assignment, a sequence, a checkpoint and a return. The definition of tr simply determines the type of p and calls a function that can translate that kind of program fragment.

$$\begin{array}{l}
\mathrm{tr}(p, b, a, r) \doteq \\
\quad \text{if } p^h \hat{=} 0 \text{ then } \mathrm{var}(p^t, b, a, r) \text{ else} \\
\quad \text{if } p^h \hat{=} 1 \text{ then } \mathrm{call}(\mathrm{tr}(p^{th}, b, a, r), p^{tth}, \mathrm{tr}(p^{ttt}, b, a, r), b, a, r) \text{ else} \\
\quad \text{if } p^h \hat{=} 2 \text{ then } \mathrm{create}(p^{th}, p^{tth}, p^{ttth}, p^{tttth}, p^{ttttt}, b, a, r) \text{ else} \\
\quad \text{if } p^h \hat{=} 3 \text{ then } \mathrm{assign}(p^{th}, \mathrm{tr}(p^{tt}, b, a, r), b, a, r) \text{ else} \\
\quad \text{if } p^h \hat{=} 4 \text{ then } \mathrm{seq}(\mathrm{tr}(p^{th}, b, a, r), \mathrm{tr}(p^{tt}, b, a, r), b, a, r) \text{ else} \\
\quad \text{if } p^h \hat{=} 5 \text{ then } \mathrm{check}(b, a, r) \text{ else } \mathrm{return}(b, a, r)
\end{array}$$

$\mathrm{var}(i, b, a, r)$ returns a parameterised continuation which fetches either an operation argument from the environment or a variable value from the state. For simplicity, access to uninitialised variables gives a failure.

$$\begin{array}{l}
\mathrm{var}(i, b, a, r) \doteq \lambda e.\lambda(b', m, s). \\
\quad \text{if } i \hat{=} a \text{ then } (e(\mathrm{Parm}), e, s) \text{ else} \\
\quad\quad \text{let } v = s(\mathrm{Object}, b)(\mathrm{Val})(i) \text{ in} \\
\quad\quad\quad \text{if } v \hat{=} \mathsf{T} \text{ then } (\mathrm{Fail}, e, s) \text{ else } (v, e, s)
\end{array}$$

$\mathrm{seq}(c, c', b, a, r)$ puts the two continuations c and c' in sequence. This is done by executing one atomic action of c and returning c' if c terminates.

$$
\begin{array}{l}
\mathrm{seq}(c, c', b, a, r) \doteq \lambda e.\lambda(b', m, s). \\
\quad \text{let } (v, e', s') = c(e)(b', m, s) \text{ in} \\
\quad\quad \text{if } v = \mathrm{Fail} \text{ then } (\mathrm{Fail}, e', s') \text{ else} \\
\quad\quad\quad \text{if } v \hat{=} \mathrm{Done} \\
\quad\quad\quad\quad \text{then } (c', e', s) \\
\quad\quad\quad\quad \text{else } (\mathrm{seq}(v, c', b, a, r), e', s')
\end{array}
$$

check(b, a, r) makes a checkpoint of object b' whose process executes the checkpoint construct. The checkpoint is placed on the node where b' currently resides. The object b that lexically contains the checkpoint construct is not checkpointed.

$$
\begin{array}{l}
\mathrm{check}(b, a, r) \doteq \lambda e.\lambda(b', m, s). \\
\quad \text{let } s' = s\langle \mathrm{Object}, b' \rangle \langle \mathrm{CheckLoc} \rangle := s(\mathrm{Object}, b')(\mathrm{Location}) \text{ in} \\
\quad\quad \text{let } s'' = s'\langle \mathrm{Object}, b' \rangle \langle \mathrm{CheckVal} \rangle := s(\mathrm{Object}, b')(\mathrm{Val}) \text{ in} \\
\quad\quad\quad (\mathrm{Done}, e, s'')
\end{array}
$$

return(b, a, r) returns the return value stored in the local environment (and fails if the return value is unassigned). It also returns the local environment of the calling operation.

$$
\begin{array}{l}
\mathrm{return}(b, a, r) \doteq \lambda e.\lambda(b', m, s). \\
\quad \text{let } v = e(\mathrm{Result}); e' = e(\mathrm{Env}) \text{ in} \\
\quad\quad \text{if } v \hat{=} \mathsf{T} \\
\quad\quad\quad \text{then } (\mathrm{Fail}, e', s) \\
\quad\quad\quad \text{else } (v, e', s)
\end{array}
$$

Execution of an assignment consists of all those atomic actions that are necessary to evaluate the expression followed by a single atomic action which performs the assignment. The expression is evaluated by assign and the assignment is performed by assign$'$. The assignment affects the local environment if the assignment variable is the return parameter and affects a variable of the lexically enclosing object otherwise.

$$
\begin{array}{l}
\mathrm{assign}(i, c, b, a, r) = \lambda e.\lambda(b', m, s). \\
\quad \text{let } (v, e', s') = c(e)(b', m, s) \text{ in} \\
\quad\quad \text{if } v \hat{=} \mathrm{Fail} \text{ then } (\mathrm{Fail}, e', s') \text{ else} \\
\quad\quad\quad \text{if } v \in \mathbf{N} \\
\quad\quad\quad\quad \text{then } (\mathrm{assign}'(i, v, b, a, r), e', s') \\
\quad\quad\quad\quad \text{else } (\mathrm{assign}(i, v, b, a, r), e', s')
\end{array}
$$

$$
\begin{array}{l}
\mathrm{assign}'(i, v, b, a, r) = \lambda e.\lambda(b', m, s). \\
\quad \text{if } i \hat{=} r \\
\quad\quad \text{then } (\mathrm{Done}, e\langle \mathrm{Result} \rangle := v, s) \\
\quad\quad \text{else } (\mathrm{Done}, e, s\langle \mathrm{Object}, b \rangle \langle \mathrm{Val} \rangle \langle i \rangle := v)
\end{array}
$$

Invocation of an operation consists of all those atomic actions involved in computing which object to call followed by a single atomic action for looking up the operation followed by all those atomic actions involved in computing the argument followed by the actual invocation. call computes the object, call$'$ looks up the operation and call$''$ takes care of the rest.

$$
\begin{array}{l}
\mathrm{call}(c,i,c',b,a,r) \doteq \lambda e.\lambda(b',m,s). \\
\quad \text{let } (v,e',s') = c(e)(b',m,s) \text{ in} \\
\quad\quad \text{if } v \hat{=} \mathrm{Fail} \text{ then } (\mathrm{Fail},e',s') \text{ else} \\
\quad\quad\quad \text{if } v \in \mathbf{N} \\
\quad\quad\quad\quad \text{then } (\mathit{call}'(v,i,c,b,a,r),e',s') \\
\quad\quad\quad\quad \text{else } (\mathit{call}(v,i,c,b,a,r),e',s')
\end{array}
$$

$$
\begin{array}{l}
\mathrm{call}'(v,i,c,b,a,r) \doteq \lambda e.\lambda(b',m,s). \\
\quad \text{let } p = s(\mathrm{Object},v)(\mathrm{Ops})(i) \text{ in} \\
\quad\quad \text{if } p \hat{=} \mathsf{T} \text{ then } (\mathrm{Fail},e,s) \text{ else} \\
\quad\quad\quad (\mathrm{call}''(p,c,b,a,r),e,s)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{call}''(p,c,b,a,r) \doteq \lambda e.\lambda(b',m,s). \\
\quad \text{let } (v,e',s') = c(e)(b',m,s) \text{ in} \\
\quad\quad \text{if } v \hat{=} \mathsf{T} \text{ then } (\mathrm{Fail},e',s') \text{ else} \\
\quad\quad\quad \text{if } v \notin \mathbf{N} \\
\quad\quad\quad\quad \text{then } (\mathit{call}''(p,v,b,a,r),e',s') \\
\quad\quad\quad\quad \text{else let } e'' = \mathsf{T}\langle\mathrm{Parm}\rangle := v; e''' = e''\langle\mathrm{Env}\rangle := e' \text{ in} \\
\quad\quad\quad\quad\quad (p,e'',s')
\end{array}
$$

Only object creation remains to be described. Object creation is by far the most complicated operation in the abstract syntax. create$(q,q',\mathrm{init},\mathrm{rec},\mathrm{proc})$ takes as arguments a list q of monitored operations, a list q' of non-monitored operations, and the abstract syntax for the initialisation code init, the recovery code rec, and the object process proc.

The object b' whose process executes the create construct has a token tree t which is used for allocating an id b'' for the new object. The new object gets tt.head(t) as token tree, and the object that donates the token tree keeps tt.tail(t). The root of the donated token tree is used as id for the created object.

The continuation of the created object b'' is set to the initialisation code followed by the process. At recovery, the object executes the recovery code followed by the process. Initialisation and recovery code is executed inside the monitor before any monitored operations can be started. This is ensured by setting the first element in the monitor queue to b''.

The created object is placed on the node of the lexically enclosing object b which is where the creation construct is executed.

The created object has no variables to begin with. Variables are created by assignment statements.

For simplicity, the created object is checkpointed immediately so that all objects have a checkpoint.

$$
\begin{array}{l}
\mathrm{create}(q, q', \mathrm{init}, \mathrm{rec}, \mathrm{proc}, b, a, r) \doteq \lambda e.\lambda(b', m, s). \\
\quad \mathrm{let}\ \mathrm{TokenTree}_1 = s(\mathrm{Object}, b')(\mathrm{TokenTree})\ \mathrm{in} \\
\quad \mathrm{let}\ \mathrm{TokenTree}_2 = \mathrm{tt.head}(\mathrm{TokenTree}_1)\ \mathrm{in} \\
\quad \mathrm{let}\ b'' = \mathrm{tt.root}(\mathrm{TokenTree}_2)\ \mathrm{in} \\
\quad \mathrm{let}\ c = \mathrm{tr}(\mathrm{init}, b'', \mathsf{T}, \mathsf{T})\ \mathrm{in} \\
\quad \mathrm{let}\ c' = \mathrm{tr}(\mathrm{rec}, b'', \mathsf{T}, \mathsf{T})\ \mathrm{in} \\
\quad \mathrm{let}\ c'' = \mathrm{tr}(\mathrm{proc}, b'', \mathsf{T}, \mathsf{T})\ \mathrm{in} \\
\quad \mathrm{let}\ n = s(\mathrm{Object}, b)(\mathrm{Location})\ \mathrm{in} \\
\quad \mathrm{let}\ s_1 = s\langle \mathrm{Object}, b'' \rangle \langle \mathrm{Location} \rangle := n\ \mathrm{in} \\
\quad \mathrm{let}\ s_2 = s_1\langle \mathrm{Object}, b'' \rangle \langle \mathrm{Val} \rangle := \mathsf{T}\ \mathrm{in} \\
\quad \mathrm{let}\ s_3 = s_2\langle \mathrm{Object}, b'' \rangle \langle \mathrm{Cont} \rangle := \mathrm{seq}(\mathrm{seq}(c, \mathrm{release}(b'')), c'')\ \mathrm{in} \\
\quad \mathrm{let}\ s_4 = s_3\langle \mathrm{Object}, b'' \rangle \langle \mathrm{MonState} \rangle := (\mathsf{T}\langle 1 \rangle := b'')\ \mathrm{in} \\
\quad \mathrm{let}\ s_5 = s_4\langle \mathrm{Object}, b'' \rangle \langle \mathrm{TokenTree} \rangle := \mathrm{TokenTree}_2\ \mathrm{in} \\
\quad \mathrm{let}\ s_6 = \text{tr-mop}(q, b, s_5)\ \mathrm{in} \\
\quad \mathrm{let}\ s_7 = \text{tr-uop}(q', b, s_6)\ \mathrm{in} \\
\quad \mathrm{let}\ s_8 = s_7\langle \mathrm{Object}, b'' \rangle \langle \mathrm{Recover} \rangle := \mathrm{seq}(\mathrm{seq}(c', \mathrm{release}(b'')), c'')\ \mathrm{in} \\
\quad \mathrm{let}\ s_9 = s_8\langle \mathrm{Object}, b'' \rangle \langle \mathrm{CheckLoc} \rangle := n\ \mathrm{in} \\
\quad \mathrm{let}\ s_{10} = s_9\langle \mathrm{Object}, b'' \rangle \langle \mathrm{CheckVal} \rangle := \mathsf{T}\ \mathrm{in} \\
\quad \mathrm{let}\ s_{11} = s_{10}\langle \mathrm{Object}, b' \rangle \langle \mathrm{TokenTree} \rangle := \mathrm{tt.tail}(\mathrm{TokenTree}_1)\ \mathrm{in} \\
\quad\quad (\mathrm{Done}, e, s_{11})
\end{array}
$$

release(b) is an atomic action which releases the monitor of object b.

$$
\begin{array}{l}
\mathrm{release}(b) \doteq \lambda e.\lambda(b', m, s). \\
\quad \mathrm{let}\ \mathrm{mon} = s(\mathrm{Object}, b)(\mathrm{MonState})\ \mathrm{in} \\
\quad \mathrm{let}\ \mathrm{mon}' = \lambda i.\mathrm{mon}(i+1)\ \mathrm{in} \\
\quad \mathrm{let}\ s' = s\langle \mathrm{Object}, b \rangle \langle \mathrm{MonState} \rangle := \mathrm{mon}'\ \mathrm{in} \\
\quad\quad (\mathrm{Done}, e, s')
\end{array}
$$

Object b' asks for entering the monitor of object b by acquire(b). b' is placed in position m in the queue (if that position is vacant, and somewhere else otherwise). Then i is set to the position of b' in the queue, and if there are no objects before b', then b' enters the monitor. Note that positions in the queue may be empty.

$$
\begin{array}{l}
\text{acquire}(b) \doteq \lambda e.\lambda(b', m, s). \\
\quad \text{let mon} = s(\text{Object}, b)(\text{MonState}) \text{ in} \\
\quad \text{let mon}' = \text{addQueue}(\text{mon}, b', m) \text{ in} \\
\quad \text{let } i = \varepsilon j{\in}\mathbf{N}.\text{mon}'(j){\hat{=}}b' \text{ in} \\
\quad\quad \text{if } \exists j{\in}\mathbf{N}.j < i \wedge \text{mon}'(j) \neq \mathsf{T} \\
\quad\quad\quad (\text{acquire}(b), e, s) \\
\quad\quad \text{else} \\
\quad\quad\quad \text{let mon}'' = \lambda j.\text{mon}'(i + j - 1) \text{ in} \\
\quad\quad\quad \text{let } s' = s\langle \text{Object}, b\rangle\langle \text{MonState}\rangle := \text{mon}'' \text{ in} \\
\quad\quad\quad\quad (\text{Done}, e, s')
\end{array}
$$

If b is not already queued then addQueue(mon, b, m) adds b to the monitor queue mon at position m if that position is vacant and somewhere else otherwise.

$$
\begin{array}{l}
\text{addQueue}(\text{mon}, b, m) \doteq \\
\quad \text{if } \exists i \in \mathbf{N}.\text{mon}(i){\hat{=}}b \text{ then mon else} \\
\quad\quad \text{if mon}(m){\hat{=}}\mathsf{T} \text{ then mon}\langle m\rangle := b \text{ else} \\
\quad\quad\quad \text{mon}\langle \varepsilon i{\in}\mathbf{N}.\text{mon}(i){\hat{=}}\mathsf{T}\rangle := b
\end{array}
$$

tr-uop(q, b, s) translates the sequence q of non-monitored operations of object b and install them in s as operations of b. In the definition, op is the abstract syntax of an operation definition (cf. Section 6), i identifies the operation, p is the body of the operation, a is the argument variable of the operation, and r is the return parameter.

$$
\begin{array}{l}
\text{tr-uop}(q, b, s) \doteq \\
\quad \text{if } q{\hat{=}}\mathsf{T} \text{ then } s \text{ else} \\
\quad\quad \text{let } s' = \text{tr-uop}(q^t, b, s) \text{ in} \\
\quad\quad \text{let op} = q^h; i = \text{op}^h; a = \text{op}^{th}; r = \text{op}^{tth}; p = \text{op}^{ttt} \text{ in} \\
\quad\quad \text{let } c = \text{tr}(p, b, a, r) \text{ in} \\
\quad\quad \text{let } s'' = s'\langle \text{Object}, b\rangle\langle \text{Ops}, i\rangle := c \text{ in } s''
\end{array}
$$

tr-mop is like tr-uop except that each operation acquires the monitor lock before starting and releases the lock at completion.

$$
\begin{array}{l}
\text{tr-mop}(q, b, s) \doteq \\
\quad \text{if } q{\hat{=}}\mathsf{T} \text{ then } s \text{ else} \\
\quad\quad \text{let } s' = \text{tr-mop}(q^t, b, s) \text{ in} \\
\quad\quad \text{let op} = q^h; i = \text{op}^h; a = \text{op}^{th}; r = \text{op}^{tth}; p = \text{op}^{ttt} \text{ in} \\
\quad\quad \text{let } c = \text{tr}(p, b, a, r) \text{ in} \\
\quad\quad \text{let } c' = \text{seq}(\text{seq}(\text{acquire}(b), c), \text{release}(b)) \text{ in} \\
\quad\quad \text{let } s'' = s'\langle \text{Object}, b\rangle\langle \text{Ops}, i\rangle := c' \text{ in } s''
\end{array}
$$

14 Conclusion and further work

A semantics for a subset of Emerald has been given. The subset focuses on parallelism and distribution. Possible future work includes to scale up the semantics to cover full Emerald and to discuss the semantics with the language designers. The semantics of the given subset has been constructed with such scale-up in mind.

Another obvious task would be to define a reasonable process equivalence relation in map theory, e.g. based on bi-simulation, and to show such an equivalence to be a congruence with respect to the abstract syntax in Section 6. This would allow to construct a normal model of Emerald such that two programs were equal in the model if they had the same behaviour.

Acknowledgement

My thanks are due to Eric Jul for useful comments and suggestions to the manuscript.

References

[1] J-R. Abrial. The specification language Z: Syntax and "semantics". Technical report, Programming Research Group, Oxford, April 1980. (out of print).

[2] P. Aczel. *Non-Well-Founded Sets*, volume 14 of *Lecture Notes*. CSLI, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, 1988.

[3] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundation of Mathematics*. North-Holland, 1984.

[4] M.J. Beeson. Towards a computational system based on set theory. *Theoretical Computer Science*, 60:297–340, 1988.

[5] D. Bjørner and C.B. Jones. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[6] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. Object Structure in the Emerald System. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, October 1986.

[7] Andrew P. Black, Norman C. Hutchinson, Eric Jul, Henry M. Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[8] K. Grue. Map theory. *Theoretical Computer Science*, 102(1):1–133, July 1992.

[9] K. Havelund. The fork calculus. Ph.D. Thesis Diku Report 94/4, The University of Copenhagen, Dept. Comp. Sci., Universitetsparken 1, DK-2100 Copenhagen, Denmark, January 1994.

[10] K. Havelund and K. G. Larsen. The fork calculus. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *20th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 544–557. LNCS 700, 1993.

[11] K. Havelund and K. G. Larsen. The fork calculus. In S. Meldal and M. Haveraaen, editors, *4th Nordic Workshop on Program Correctness, Report no 78*, pages 153–164. Department of Informatics, University of Bergen, Norway, 1993.

[12] D. Hilbert and P. Bernays. *Grundlagen der Mathematic*, volume 2. Springer-Verlag, 1939.

[13] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[14] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.

[15] N. C. Hutchinson, R. K. Raj, A. P. Black, H. M. Levy, and E. Jul. The emerald programming language. Diku Report 87/22, The University of Copenhagen, Dept. Comp. Sci., Universitetsparken 1, DK-2100 Copenhagen, Denmark, October 1993.

[16] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, TR 87-01-01, Department of Computer Science, University of Washington, Seattle, January 1987.

[17] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald Programming Language Report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, October 1987. (Revised August 1988).

[18] Eric Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, TR 88-12-06, Department of Computer Science, University of Washington, Seattle, December 1988.

[19] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. Fine-grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[20] Eric Jul, Rajendra K. Raj, and Norman Hutchinson. The Emerald System: User's Guide. Emerald Project Internal Memorandum (Revised November 1988), July 1988.

[21] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*. North-Holland, 1975. Volume 80 of Studies in Logic and The Foundation of Mathematics.

[22] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195, 1960.

[23] E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth and Brooks, 3. edition, 1987.

[24] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[25] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software—Practice and Experience*, 21(1):91–118, January 1991.

[26] Guy L. Steele. *Common Lisp—The Language.* Digital Press, second edition, 1990.