BRICS NS-95-4 Gurevich & Börger: Evolving Algebras

BRICS

Basic Research in Computer Science

Evolving Algebras

Mini-Course

Yuri Gurevich
Egon Börger

BRICS Notes Series **NS-95-4**

ISSN 0909-3206 **July 1995**

See back inner page for a list of recent publications in the BRICS Notes Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

BRICS

Mini-Course on

Evolving Algebras

Yuri Gurevich
University of Michigan

Egon Börger
University of Pisa

University of Aarhus
7–10 August, 1995

Contents

EVOLVING ALGEBRAS: AN ATTEMPT TO DISCOVER SEMANTICS*

YURI GUREVICH
Electrical Engineering and Computer Science Department
The University of Michigan
Ann Arbor, MI 48109-2122, USA

Fall 1992

Preface

This tutorial is based on lecture notes from the Fall 1990 course on Principles of Programming Languages at the University of Michigan. (My young friend Quisani did not attend the lectures.) The present version incorporates some changes provoked by the necessity to update the bibliography. The main part of the paper is still the same, however, and the examples are unchanged even though many things happened in the meantime. In particular, we (the collective we) have learned how to build evolving algebras by the method of successive refinements, and the current evolving algebra description of the C programming language in [GH] doesn't look much like the `strcpy` example anymore. Now, we understand better how to compose evolving algebras and how to prove things with evolving algebras. A typical misconception is that the operational approach is necessarily too detailed. Some people think that an approach suited for complexity analysis does not give a good high-level specification language. I believe in a high-level specification language based on evolving algebras; the successive refinement method is then one tool to prove implementation correctness. But this and various other issues (how to incorporate real time into evolving algebras for example) will have to be addressed elsewhere.

* *Current Trends in Theoretical Computer Science*, eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292. The first version of this tutorial was published in *Bull. EATCS*, No. 43, Feb. 1991, 264–284. Partially supported by NSF and ONR.

1. Another computation model

Quisani: Somebody told me that you are doing semantics these days.

Author: Somebody was right.

Q: Sure, somebody is usually right. Tell me about your semantics.

A: The original idea was to provide operational semantics for algorithms by elaborating upon what may be called the implicit Turing's thesis: every algorithm is simulated by an appropriate Turing machine [Gu1]. Turing did not claim this explicitly; his thesis was: every computable function is computable by some Turing machine. But his informal proof of the thesis [Tu] gives the stronger version. In the sense of the stronger thesis, Turing machines give operational semantics to algorithms. Unfortunately, this semantics is not very good (and of course I do not claim that semantics was Turing's goal). Turing machine simulation may be very clumsy. In particular, one step of the algorithm may require a long sequence of steps of the simulating Turing machine. I was looking for machines able to simulate algorithms much more closely. In particular, the simulation should be *lock-step* so that the simulating machine makes only a bounded number of steps to simulate one step of the given algorithm. Evolving algebras, or EAs, are supposed to be such machines.

Q: There are abstract machines in the literature better suited to simulate algorithms than Turing machines: Kolmogorov-Uspensky machines [KU], storage modification machines of Schönhage [Sch], various random access machines.

A: Each of these machine models has a fixed level of abstraction which may be very low for some algorithms. An evolving algebra, on the other hand, may be tailored to an arbitrary abstraction level. One may have a whole hierarchy of evolving algebras of various abstraction levels for the same algorithm. See [GH] for an example where you will find EAs for various abstraction levels of the C programming language. A hierarchy of evolving algebras is constructed by Egon Börger and Dean Rosenzweig in [BRo1] where the technique of successive refinements is used to reconstruct the Warren Abstract Machine (a virtual machine model which underlies most of the current Prolog implementations and incorporates crucial optimization techniques) starting from a more abstract EA for Prolog developed by Börger in [Bo1–Bo3].

Q: How do you tailor an EA machine to the abstraction level of an algorithm whose individual steps are complicated algorithms all by themselves? For example, the algorithm may be written in a high level language that allows, say, multiplying integer matrices in one step.

A: You model the given algorithm modulo those algorithms needed to perform single steps. In your case, matrix multiplication will be built in as an operation.

Q: Coming back to Turing, there could be a good reason for him to speak about computable functions rather than algorithms. We don't really know what algorithms are.

A: I agree. Notice, however, that there are different notions of algorithm. On the one hand, an algorithm is an intuitive idea which you have in your head before writing code. The code then implements the algorithm. The same algorithm may be coded in different programming languages. It may have many different abstraction levels. In particular, a source program and the result of its compilation implement versions of the same algorithm that differ in their abstraction levels. We can argue which, if any, of the abstraction levels is the natural one. The question when two such intuitive algorithms are the same may be hard.

On the other hand, there is a more naive notion according to which an algorithm essentially is a program (together with some computing environment which is often implicit). In particular, different programs give different algorithms. I do not want to completely equate programs and algorithms though; one speaks usually about programs in particular programming languages. The notion of algorithm is a little more general. A programming language itself can be seen as an algorithm – a universal algorithm that takes a given program as a part of its data.

Q: Do you want to capture the naive notion of algorithm by means of evolving algebras?

A: The goal is good operational semantics, but it would be great to formalize properly the notion of algorithms, wouldn't it?

Q: To what extent is this formalization goal achieved?

A: Well, I will explain to you the notion of sequential evolving algebras. It seems to me that it captures the notion of sequential algorithms, that every sequential algorithm can be closely simulated by an appropriate sequential EA.

Q: What do you mean "closely simulated"?

A: The simulating EA is on the same abstraction level and does not use much more resources than the given algorithm. In particular the simulation is lock-step.

Q: But who knows what kind of resources the given algorithm uses?

A: Well, give me a sequential algorithm and tell me which resources you care about. Then we will be able, I think, to construct a sequential EA simulating your algorithm closely with respect to the resources in question.

Q: What are your arguments?

A: Speculation and experimentation. In particular, EAs were constructed for a number of sequential programming languages, e.g., Modula-2 [GMr], Prolog [Bo1–Bo3], Prolog III [BS], Protos-L [BB] and C [GH]. In this modeling of programming languages, the goal was direct and natural operational semantics. But these models confirm the thesis as well.

Q: What about nonsequential, say distributed parallel, algorithms?

A: The notion of sequential EA has been generalized to the distributed parallel case [GMs]. In particular, EAs were constructed for Occam [GMs], Parlog [BRi1] and Concurrent Prolog [BRi2]. See also [GR]. I do not know any distributed parallel algorithm that presents a conceptual challenge for EA formalization. There is, however, a difference between the sequential and distributed parallel cases. An arbitrary sequential algorithm can be viewed as a sequential transition system; analyzing such systems, you discover sequential evolving algebras and may justify to an extent the thesis. The notion of distributed parallel algorithms seems open-ended at the moment.

Q: Even if one buys your thesis, he/she may not like your semantics.

A: I agree but hope that he/she will like it. EAs are relatively easy to understand and design. I use them in class. Even a very small program in an unusual language can be difficult to understand directly. Sketching an appropriate EA on a blackboard may help. You can define various machine models as special classes of evolving algebras.

Q: Do you have software to run an EA machine?

A: Yes. For this purpose, we use an EA interpreter written in C here at Michigan. You can run your EA for a specified number of steps and then examine the resulting state, you can run your EA until some predicate becomes true, and so on.

Q: Do you see any use for evolving algebras outside the classroom?

A: Yes. Here are some examples. ISO, the International Standards Organization, adapted Egon Börger's proposal [BD] of an EA based precise semantics of full Prolog [WG17]. (Actually, Egon Börger is not completely happy with the adaptation. In cooperation with Dean Rosenzweig, he prepared a simpler evolving algebra description of full Prolog [BRo2].) Georg Gottlob, Gerti Kappel, and Michael Schrefl used EAs to specify the semantics of characteristic features of object-oriented database models [GKS, KS]. An EA based manual for a programming language could be precise and relatively easy to read.

Q: Do you expect evolving algebras to be used for proving things about algorithms?

A: Yes. The possibility to tailor EAs to any given abstraction level is especially useful. I have already mentioned Börger-Rosenzweig's work on the Warren Abstract Machine [BRo1]. Starting from a Prolog evolving algebra on a high abstraction level (essentially the level of SLD-resolution), a hierarchy of more and more refined evolving algebras is constructed; the final algebra is the first formal abstract specification of WAM in the literature. It is proved that, under appropriate assumptions, every $(i+1)$-st algebra correctly implements the i-th algebra. This is a solid foundation for constructing provably correct compilers from Prolog to WAM; the mentioned proof assumptions give rise to conditions that guarantee correctness. Using similar techniques, my student Jim Huggins is attempting to prove the correctness of the Kermit file transfer protocol.

The EA approach is appropriate for the use of temporal and dynamic logics though we have only started to explore this. In a sense, the EA approach provides a foundation for the use of such logics. If you are looking for models of first-order temporal and dynamic logics, think about evolving algebras of appropriate kinds. (By the way, evolving algebras are called dynamic algebras sometimes, but the latter term is used in the dynamic logic area in a very different sense; see [Pr] for example.)

Q: Now tell me what evolving algebras are.

A: The basic idea is very simple, at least in the sequential case, when time is sequential (the algorithm starts in some initial state S_0 and goes through states S_1, S_2, etc.) and only a bounded amount of work is done on each step. Each state can be represented by a first-order structure: a set with relations and functions. (The term "first-order structure" is misleading. It is logic that is first-order, not structures. Models of second-order logic, for example, can be easily and naturally represented by "first-order" structures. But the term is common and we will use it.) Thus, the run can be seen as a succession of first-order structures, but this isn't a very fruitful way to see the process. How do we get from a state S_i to the next state S_{i+1}? Following the algorithm, we perform a bounded number of changes in S_i. It turns out that the whole algorithm can be rewritten as a finite number of transition rules of very simple form.

By the way, I am thinking about states of the algorithm as something feasible. Certainly, any computer executing the algorithm is able (all the time it is executing correctly) to represent the states. On the other hand, the whole process may be huge, unwieldy and infeasible to represent.

It isn't obvious how to generalize the basic idea to the case of asynchronous distributed algorithms; see [GMs] in this connection.

Q: It seems that you have an evolving first-order structure. Why do you call it an evolving algebra?

A: For technical reasons, it is convenient to replace relations by appropriate functions. In universal algebra, a first-order structure without relations is called an algebra [Gr]; we will use the term "static algebra".

2. Static algebras

Q: Please explain to me what static algebras are exactly.

A: Gladly. Let me start at the very beginning. The term *signature* will be used to mean a finite collection of function names. It is supposed that each function name comes with an indication of its arity, i.e., the number of arguments. A *static algebra* of a signature σ is a nonempty set $\mathcal{S}$, called the *superuniverse*, together with interpretations on $\mathcal{S}$ of function names in σ. A function name of arity r is interpreted as an r-ary function from $\mathcal{S}$ to $\mathcal{S}$, i.e., a function from $\mathcal{S}^r$ to $\mathcal{S}$, and is called a *basic function* of the algebra. A basic function of arity zero is called a *distinguished element*.

Q: I thought a function is always a function of something so that the arity of a function is at least one.

A: Stretching notions is not new to science; recall the notion of zero-speed motion in physics for example. One may insist that zero-speed motion is no motion at all, but it is more convenient to view zero-speed motion as a special case of motion. By the way, in standard terminology, the term "universe", rather than "superuniverse", is used. We reserve the term "universe" for other purposes.

Here is an example of a static algebra (which I will have to modify). Suppose that a signature σ contains zero-ary function names 0, 1 and binary function names $+$ and $\times$. One static algebra of signature σ is obtained as follows: Take the superuniverse to be the set of integers and interpret the function names in the obvious way. There are other natural static algebras of signature σ, but in general a static algebra need not be natural; it may be very arbitrary.

Q: I understand that static algebras will represent snapshots of computational processes. Isn't your definition too restrictive? For example, you may want to extend that arithmetical static algebra by the division operation but the division operation is partial.

A: Instead of generalizing the notion of a static algebra by permitting partial functions, we will restrict attention to algebras with a distinguished element *undef*. In particular, that arithmetical algebra above should be modified by adding a new element *undef*. It can be further extended by adding the division function; we will have $a \div b = undef$ unless both a and b are integers and b divides a. Formally, every r-ary basic function f is defined on every r-tuple $\bar{a}$ of elements of the superuniverse, but we will say that f is undefined at $\bar{a}$ if $f(\bar{a}) = undef$; the set of tuples $\bar{a}$ with $f(\bar{a}) \neq undef$ will be called the *domain* of f.

Q: Still, your static algebras seem too restrictive to me. How do you deal with different data types?

A: One possibility would be to generalize the notion of a static algebra and to consider many-sorted algebras; such algebras are widely used. But this is not necessary. We will suppose that every static algebra contains distinguished elements *true* and *false*. A basic function U, defined on the whole superuniverse, with values in $\{true, false\}$ will be viewed as the set of elements a such that $U(a) = true$ and called a *universe*. We will say that a belongs to U if $U(a) = true$.

Further, we will suppose that every static algebra has the equality relation, a universe *Bool*, comprising two elements *true* and *false*, and the usual boolean operations; the result of any boolean operation is *undef* if at least one of the arguments is outside *Bool*. Thus the arithmetical algebra should be modified again; a universe *Integer* comprising the integers can be declared as well. A relation R on a particular universe U will be represented by (and identified with) the characteristic function of R which is undefined (i.e. equal to *undef*) if at least one of the arguments is outside of U. Now we may further augment the twice modified arithmetic algebra by the standard ordering of integers.

Q: You can dispense with one of the two boolean values as it is done in Lisp. For example, *true* may be dropped and then any element different from *false*, with the possible exception of *undef*, will represent boolean truth.

A: Sure, but I prefer to distinguish between boolean truth and, say, numbers.

Q: Why do you need the trick of coding relations as functions? Why can't a signature contain relation names?

A: The reason is to make updates, introduced below, applicable to relations as well. By the way, ever-present basic functions with predetermined domains and values are called *logical constants*. Logical constants may be omitted when a static algebra is described. For example, the thrice modified arithmetical algebra can be described as follows. It has (1) a universe *Integer* comprising the integer numbers, (2) distinguished integers 0 and 1, (3) the usual total binary operations $+$ and $\times$, the usual partial operation $\div$ and the usual ordering $<$ on *Integer*.

Q: Did we finish with logical constants?

A: Almost. Sometimes it is necessary to suppose that there is auxiliary infinite (or sufficiently large finite) universe *Reserve* disjoint from other universes and a special function that selects an element of *Reserve*. The role of *Reserve* will be clear soon.

Q: I am confused. I thought that we will be dealing with finite feasible static algebras reflecting snapshots of real computer computations. If this is the case, then there is no room for infinite universes.

A: In principle it is possible to keep everything finite and feasible. For example, the *Reserve* can reflect real computational resources. However, it is much more practical to divide concerns. You permit whatever universes and basic functions are convenient and, if necessary, you keep track of various resources.

Finally, let me recall the definition of *closed terms* of a given signature (the first clause is superfluous but it seems to make comprehension easier):

- Every zero-ary function name is a closed term.
- If f is a function name of arity r and $t_1, \ldots, t_r$ are closed terms then $f(t_1, \ldots, t_r)$ is a closed term.

We will use the usual abbreviations and conventions in order to increase readability. Here are some terms in the arithmetical signature mentioned above:
$0, 1 + 1, (1 + 1) \times (1 + 1)$.

3. Transition rules

Q: How do you program the evolution of an evolving algebra?

A: The *basic language* of transition rules is very modest and even minimal in some justifiable sense. Think about static algebras as states of evolving algebras. What is a simplest atomic change of a static algebra?

Q: To change one function at one place, I guess. There are, however, other atomic changes: Adding an element to the superuniverse, removing an element from the superuniverse, adding a new basic function, removing a new basic function.

A: Adding elements to the superuniverse can be avoided with the help of *Reserve*. To extend a universe U, place a *Reserve* element a in U; technically, set U to *true* and Reserve to *false* at a. To remove an element a from U, set $U(a) = \mathit{false}$.

Q: I see. And what about adding and removing basic functions.

A: I do not believe that this is ever necessary if your EA properly formalizes the given algorithm.

Q: Suppose the algorithm is given by a program which declares new functions from time to time. This isn't unusual, right?

A: Right. But the new functions can be treated as data. For example, you may have a whole universe F of, say, unary operations on some universe U. To deal with this situation, you may have a binary basic function *Apply*. If $f \in F$ and $u \in U$ then $\mathit{Apply}(f, u)$ will be the result of applying f to u.

Q: What if my program declares functions of *a priori* unbounded arities? Better yet, it may declare a function that takes an arbitrary number of arguments.

A: Use the usual tricks. A function with arbitrarily many arguments can be viewed as a unary function of lists, for example.

Q: Thus, your basic language allows no extension or contraction of either the signature or the superuniverse.

A: That is right. Commands of the basic language will be called *transition rules* or simply *rules*. The only primitive transition rule is a *local function update* of the form

$$f(t_1, \ldots, t_r) := t_0$$

where f is (the name of) a basic function, r is the arity of f and every t_i is a closed term. Is the meaning clear?

Q: I think so. You evaluate all terms t_i. If a_i is the value of t_i then you reset $f(a_1, \ldots, a_r)$ to a_0.

A: That is right. For brevity, let me call local function updates simply updates. A slightly more complicated rule is a *guarded update*

if b **then** u **endif**

where b (the *guard*) is any term and u is any update. If b evaluates to *true* on the given static algebra then perform u; otherwise do nothing. Let me stress that all evaluations are done in the given static algebra.

Q: I think you can get rid of guarded updates. Introduce a logical constant, say, *Cond(x,y,z)* interpreted in any static algebra as follows: If x is *true* then the result is y; otherwise the result is z. Then a guarded update above has exactly the same effect as the unguarded update

$$f(t_1,\ldots,t_r) := \mathit{Cond}(b, t_0, f(t_1,\ldots,t_r))$$

A: You are right. With *Cond*, a guarded update is equivalent to (i.e. has exactly the same effect on any static algebra as) some unguarded update. Unfortunately, the use of *Cond* makes rules more difficult to read.

Q: By the way, *Cond* corresponds naturally to a command like this:

if b **then** u_1 **else** u_2 **endif**

A: This command, which is a legal rule as well, is equivalent to a set of two regular conditional updates:

if b **then** u_1 **endif**

if $b \neq true$ **then** u_2 **endif**

A set of guarded updates, written usually as a list, is executed in parallel, so that the order is immaterial. All guarded updates on the list are performed simultaneously. For example, the rule

$a := f(a)$
$b := f(a)$

sets a and b to the same value. Imagine a demon who evaluates all relevant terms in a given state and then makes the necessary changes; the result is a new state.

Q: But different guarded updates may contradict each other. For example, you may have

$a := true$
$b := false$

where a and b evaluate to the same value. Better yet, you may have

$a := true$
$a := false$

What is the meaning then?

A: Nondeterministically one of the contradicting updates is chosen and performed. In your example, a will be set to one of the two truth values.

Q: Why did you choose to interpret a list of rules as a set rather than a sequence? The sequence approach is more usual, it is clear, it avoids nicely the problems of contradicting updates and lends itself more naturally to the use of macros.

A: I admit that the sequence interpretation may work better in many situations though the use of guards allows one to ensure the desired sequence of actions in the set approach as well and the set interpretation has its own advantages. First, it permits a certain concurrency; this is convenient and helps to achieve a better simulation of the given algorithm. Second, the nondeterminism of the set interpretation can be handy

[GMs]. Third, the set interpretation allows a natural transition to asynchronous distributed computing, but let us stick to sequential computing for the time being.

Q: Is it difficult to define basic transition rules in full generality?

A: No. Here is the definition.

- Any local function update is a rule.
- If k is a natural number, $b_0, \ldots, b_k$ are terms and $C_0, \ldots, C_{k+1}$ are sets of rules then

  if b_0 **then** C_0
  elseif b_1 **then** C_1
  $\vdots$
  elseif b_k **then** C_k
  else C_{k+1}
  endif

  as well as

  if b_0 **then** C_0
  elseif b_1 **then** C_1
  $\vdots$
  elseif b_k **then** C_k
  endif

  is a rule.

I hope that the meaning is obvious. Every rule is equivalent to a set of guarded updates; this is easy to check by induction. Therefore every set of rules is equivalent to a set of guarded updates.

Q: And what is a program?

A: A program is a set of rules. Recall that we deal with sequential algorithms; only a bounded amount of work is done at each step.

Q: Is the definition of sequential EAs complete?

A: No, not yet. Let us see an example.

4. Example 1: A stack machine

A: Consider a stack machine for computing expressions given in reverse Polish notation, or RPN. An expression $(1 + 23) \times (45 + 6)$ written in RPN would appear as

$1\ 23 + 45\ 6 + \times$

Q: Are you talking about arithmetical operations over integers?

A: Not necessarily. Suppose we have a nonempty set *Data* and a nonempty set *Oper* of (for simplicity) total binary operations on *Data*. For example, *Data* may be the set of integers and *Oper* may be $\{+, \times\}$. We suppose that the RPN expression is given to us in the form of a list where each entry denotes a datum or an operation. The stack machine reads one entry of the list at a time from the input file. If the entry denotes a datum, it is pushed onto the stack. If the entry denotes an operation, the machine pops two items from the stack, applies the operation and pushes (the notation for) the result onto the stack. At the beginning, the stack is empty. In the case of the RPN expression above (with the usual operations), the stack goes through the following states: (), (1), (23 1), (24), (45 24), (6 45 24), (51 24), (1224).

The desired evolving algebra has *Data* and *Oper* as universes. *Arg1* and *Arg2* are distinguished elements of *Data*. To handle operations in *Oper*, the EA has a ternary function *Apply* such that *Apply(f,x,y) = f(x,y)* for all f in *Oper* and all x, y in *Data*. To handle the input, the EA has a universe *List* of all lists composed of data and operations. The basic functions *Head* and *Tail* have the usual meaning. If L is a list then *Head(L)* is the first element of L and *Tail(L)* is the remaining list. *EmptyList* has the obvious meaning. F is a distinguished list. Initially, F is the input. Finally, the evolving algebra has a universe *Stack* of all stacks of data with the usual operations *Push* (from *Data* $\times$ *Stack* to *Stack*), *Pop* (from *Stack* to *Stack*) and *Top* (from *Stack* to *Data*). S is a distinguished stack. Initially, S is empty.

Q: Since the stack machine deals with only one list and has only one stack, I do not understand why you need the universes *List* and *Stack*.

A: The EA should be on the abstraction level of the given algorithm. Operations like *Push* and *Pop* should be provided with their natural environment. We can manage without the universes *List* and *Stack* by implementing the input and the stack as arrays, but this would be a lower level of abstraction. Here are the transition rules.

if *Head(F) is a datum* **then**
    S := Push(Head(F), S)
    F := Tail(F)
endif

if *Head(F) is an operation* **then**
    if $Arg1 = undef$ **then**
        $Arg1 := Top(S)$
        $S := Pop(S)$
    elseif $Arg2 = undef$ **then**
        $Arg2 := Top(S)$
        $S := Pop(S)$
    else $S := Push(Apply(Head(F), Arg1, Arg2), S)$
        $F := Tail(F)$
        $Arg1 := undef$
        $Arg2 := undef$
    endif
endif

The phrase "*Head(F)* is a datum" means of course that *Head(F)* belongs to *Data*, i.e., $Data(Head(F)) = true$. The meaning of the phrase "*Head(F)* is an operation" is similar.

Q: You suppose that the input file contains a legal RPN expression.

A: That is true. When a given algorithm has an error handling routine, it can be formalized as well.

Q: Assume that there is a bound *Max* on the depth of the stack. What changes should be made?

A: Let me suppose for simplicity that the machine simply freezes if it attempts to push a datum onto a stack of the maximal depth. Then you may want to introduce a universe of natural numbers with successor operation $n+1$ and predecessor operation $n-1$ and distinguished natural numbers *Depth* and *Max*. Initially, *Depth* is zero. Augment the guard of the first of the two rules above by the conjunct $Depth < Max$ and add updates $Depth := Depth \pm 1$ in appropriate places.

5. Static, dynamic and external functions

Q: You speak about basic functions like a programmer; a function has a name and may change. In mathematics, a function is a particular mapping. If you set sin(1) to 1/2 then the result isn't the sin function anymore.

A: You are right. Basic functions are the current interpretations of the corresponding function names. In every state of an EA, basic functions are particular mappings, but the same function name may be interpreted differently in different states. Actually, it is convenient to distinguish between *static* and *dynamic* basic functions. The distinction is syntactical: Dynamic functions appear in function updates as subjects for updating. Static functions do not change during the evolution of the algebra whereas dynamic functions may change. In the stack machine example, for instance, the only dynamic functions are S, F, $Arg1$ and $Arg2$.

Q: I see a problem with evolving algebras. They are completely isolated. How do you deal with input, output, interrupts and other interactions of the given algorithm with the outside world? Do you want to pretend that all reactions of the outside world are written in files ahead of time?

A: Let us examine the problem. In an attempt to formalize the given algorithm as an evolving algebra, we may discover that some basic functions depend on the outside world; let f be one of them. In the case that the algorithm is given by a program, f may reflect, for example, whatever the user types at the keyboard or some activity of the operating system. I presume that f is not a subject of updating by our transition rules; in other words, f is syntactically static. Nevertheless f may have different values in different states. One way to deal with this situation is to add another argument to f which will allow us to distinguish between different evaluations. For example, we may pretend, as you said, that f reads from a file and use the position in the file as a new argument. The pretense creates the illusion that the program of an evolving algebra is the only means to change its state. This illusion may be awkward to maintain and we use the following more radical way to deal with the problem.

The basic functions of an evolving algebra $\mathcal{A}$ are partitioned into *internal static* functions (in short, static functions), *internal dynamic functions* (in short, dynamic functions), and *external* functions. External functions cannot be changed by rules of $\mathcal{A}$, but they may be different in different states of $\mathcal{A}$. From the point of view of $\mathcal{A}$, an external function is an oracle, an unpredictable black box that is used but not controlled by $\mathcal{A}$.

Q: Let me see if I understand you. I imagine again the demon responsible for the evolution of $\mathcal{A}$, the one who executes the rules. In the case of an internal function f, the demon knows exactly how to compute f. For example, he may have a fixed algorithm for computing the default version of f and a complete table to account for the deviation from the default. In the case of an external function f, the demon has an unpredictable magic box for evaluating f. Whenever he needs to evaluate f, he enters the appropriate number of arguments and miraculously gets the result.

A: That is right. From the point of view of the demon, an external function is a nondeterministic function.

Q: I guess one can have rules like

$Output := t$

too, right?

A: Sure. Nothing dramatic happens in the given algebra when this rule is fired; syntactically your *Output* is just another distinguished element. But of course such rules are extremely important for communication. You may have several communication channels (and distinguished elements like *Output-on-channel-5*) and even a whole universe of channels (and a basic output function with an argument that is a channel). You may have a net of evolving algebras, but for the time being let us concentrate on a single evolving algebra (possibly communicating with the outside world).

Q: I have a question related to the types of basic functions. Formally, all r-ary basic functions are of the same type: In each state, they map the r-th power of the superuniverse to the superuniverse. It is clear, however, that you really think in terms of multiple universes and try to specify the type of basic functions in terms of universes.

A: This is very true. Essentially, we deal with many-sorted algebras though it is convenient, technically speaking, to have the superuniverse around.

Q: The case of a static function seems clear; here typing, i.e. prescribing a type, is a part of the initial state description. But what is the meaning of typing a dynamic function? Is it a declaration of intent?

A: It is an *integrity constraint*. Integrity constraints are statements (in indicative rather than imperative mood) that should be satisfied in any state during the evolution of a given EA. Often integrity constraints are implicit, but they can be stated explicitly. In the stack machine example, for instance, we have the following integrity constraints: *Arg1* and *Arg2* are data (i.e. belong to *Data*), F is a list, and S is a stack. It is easy to see that these four statements indeed hold in every state of the EA.

Q: Are there other kinds of integrity constraints?

A: Yes, usually we expect external functions to have values of certain types.

Q: What happens if an integrity constraint is violated?

A: The ideal machine breaks down.

Q: Do you suppose that integrity constraints constitute a part of the description of the EA?

A: This is not necessary. The effect of integrity constraints in question can be achieved by additional rules. In practice, however, an integrity constraint may be a convenient way to avoid writing boring transition rules. For example, stating a constraint that an external function f produces values of certain type allows one to avoid writing rules for what to do if f produces a value of a wrong type.

6. Example 2: A Turing machine

A: For the second example, I would like to describe a generic Turing machine as an evolving algebra.

Three universes of the EA, called *Control, Char* and *Displacement*, are nonempty finite sets. *InitialState* and *CurrentState* are distinguished elements of *Control*, *Blank* is a distinguished element of *Char*. *CurrentState* is a dynamic distinguished element; initially, *CurrentState* = *InitialState*. Another universe, *Tape*, is a countable infinite set; call its elements *cells*. *Head* is a dynamic distinguished cell. *Move* is a possibly partial function from *Tape* $\times$ *Displacement* to *Tape*.

Q: What kind of Turing machines are you talking about? Is the tape linear? Is it one-way infinite or two-way infinite?

A: The kind of Turing machine is determined by further restrictions on *Displacement* and *Move*. You can choose *Displacement* = $\{+1, -1\}$, and *Move* to be such that *Tape* with unary operations *Move(c,+1), Move(c,−1)* is isomorphic to natural numbers with the successor and (partial) predecessor operations. This will give you a model with a linear one-way infinite tape. You can choose *Displacement* and *Move* in a way that gives rise to two linear tapes or to one two-dimensional tape, and so on.

We need some additional basic functions though. A dynamic function *TapeCont* maps cells to characters, and functions *NewState, NewChar* and *Shift* map the cartesian product *Control* $\times$ *Char* to *Control, Char* and *Displacement* respectively. It is required that *TapeCont(c)* = *Blank* for all but finitely many cells c. The transition rules are as follows.

CurrentState := NewState(CurrentState, TapeCont(Head))

TapeCont(Head) := NewChar(CurrentState, TapeCont(Head))

Head := Move(Head, Shift(CurrentState, TapeCont(Head))

Recall that all rules fire every time in a simultaneous fashion.

Q: By the way, the usual definition of Turing machine is shorter.

A: What usual definition?

Q: A Turing machine is a quadruple, namely, a set of control states, a set of characters, a transition function and the initial state.

A: In isolation, this short definition does not really define Turing machines. For example, one cannot derive from that definition that a Turing machine has a tape. Recall that after giving the short definition, one proceeds to define configurations, etc. These additional definitions are absorbed by the general EA framework. The short definition provides, however, a convenient notation for Turing machines.

Q: You did not define what your Turing machines compute.

A: I have to repeat the usual definitions.

Q: As an exercise, let me list the integrity constraints: *CurrentState* belongs to *Control*, *Head* belongs to *Tape*, and *TapeCont* maps *Tape* to *Char*.

A: That is correct. Again all constraints are easy to check in advance and again they can be stated in the form t = *true* where t is a closed term.

Q: But the constraint on *TapeCont* is a universal statement: For every cell, the value of *TapeCont* is a character.

A: Taking for granted that the constraint is satisfied initially, it reduces to the constraint $Char(TapeCont(Head)) = true$.

Q: I guess, it isn't always so easy to check whether a constraint $t = true$ will be eventually violated.

A: This is an undecidable problem. To prove this, modify the Turing machine example by introducing a subuniverse *Nonhalting* of the universe *Control*. In an obvious way, the halting problem for Turing machines reduces to the problem whether the constraint *CurrentState* belongs to *Nonhalting* is violated. But of course the *Nonhalting* universe is very artificial and unnatural. Often, constraints are checkable in advance. Certainly, your demon, able to evaluate a bounded number of closed terms at a time, should have no trouble to check, in each state, the validity of a constraint $t = true$.

Q: You defined the tape to be infinite. I prefer finite tapes growing if and when necessary.

A: The predefined geometry of Turing tapes makes the distinction somewhat artificial. In this connection it makes more sense to consider Kolmogorov-Uspensky machines [KU] with (finite at every moment) tapes of changing geometry. (We discussed KU machines once [Gu3].) The tape of a KU machine gives rise to a dynamic universe in a very natural sense. To handle this situation, we may use a countable universe *Reserve* (also dynamic) and an external distinguished element *New* subject to the integrity constraint $New \in Reserve$. Whenever it is required that the tape gets another cell, *New* is deleted from *Reserve* and added to the tape. Of course, when we evaluate *New* next time, it belongs to *Reserve* again. Actually, adding a new element to the tape requires some work because, contrary to *Reserve*, the tape has a relatively rich structure. Let me skip the details.

Q: By the way, the halting problem easily reduces to the following decision problem: Given an EA, tell whether it will eventually reach a state with contradicting updates. Thus, this consistency problem for EAs is undecidable. I imagine that for certain applications only consistent EAs are appropriate. What do you do?

A: Often very simple syntactic restrictions suffice to ensure consistency. Sometimes, it may be appropriate to require that, for each basic function f, the guards of different updates of f are mutually exclusive. This guarantees that no basic function is updated twice at the same step. (Since the problem of mutual exclusivity of given boolean formulas is co-NP complete, one may want to require some kind of explicit mutual exclusivity.) In general, every evolving algebra $\mathcal{A}$ can be modified into a consistent variant $\mathcal{A}'$ making 2 steps per each step of $\mathcal{A}$ and halting (with setting an error flag to *true* if desired) when $\mathcal{A}$ encounters inconsistency. The idea is to check the consistency of $\mathcal{A}$'s transition rules in given state of $\mathcal{A}$ before executing them.

7. Example 3: The strcpy function

Q: Will you show me an example where the algorithm is given by a program?

A: Yes. Consider the piece $\mathcal{P}$ of C code in Figure 1 which is taken from Kernighan and Ritchie's book "The C Programming Language" [KR, page 105].

```
void strcpy (char *s, char *t)
{
     while (*s++ = *t++)
          ;
}
```

Figure 1: The `strcpy` function.

Q: I know too little about C. What does $\mathcal{P}$ do and what does this unpronounceable "strcpy" stand for? Is $\mathcal{P}$ well formed? The `while` loop does not seem to do anything.

A: $\mathcal{P}$ is well formed and defines an algorithm that copies a character string from one place in memory to another. I guess, "strcpy" abbreviates "string-copying". The pointer variables `t` and `s` point initially to the first memory locations in the original place and the new place respectively; `*s` is the content of the memory location `s`. The equality sign denotes assignment. The expression `*s++` evaluates to the content of the location `s` and has a side effect of resetting `s` to the next location. Kernighan and Ritchie admit that the code is "cryptic at first sight" [KR, page 106].

I should make a remark about the notion of a string in C. Usually, people have in mind strings of printable characters, but this is not important for us here. The important restriction is that a string does not contain the so-called null character which is used to indicate the end of a string representation in memory. The memory can be viewed as a linearly ordered set of memory locations holding characters. Each character fits into one location. (There isn't much difference in C between characters, bytes and very small integers. In particular, the null character corresponds to the number zero.) Because of the necessary null character at the end, it takes $l+1$ memory locations to store a string of l non-null characters.

Q: How can an assignment be the condition of a `while` loop?

A: An assignment statement returns the value assigned. Nonzero values represent truth and 0 represents falsity in C. The value 0 is returned when the null character is encountered.

We will construct an evolving algebra $\mathcal{A}$ modeling the `strcpy` algorithm. First, we will describe the universes of $\mathcal{A}$ and most of the basic functions. Then we will describe the transition rules and the remaining basic functions. Since $\mathcal{P}$ uses pointers to memory locations, $\mathcal{A}$ has universes *Char*, *Loc* and *Addr*. Elements of *Char* will be called *characters*; *null* is a distinguished character. The universe *Loc* comes equipped with a successor function. Elements of *Loc* will be called *memory locations*. (C allows more arithmetic on memory locations, but only the successor function is used in $\mathcal{P}$.) A dynamic function *LocCont* from *Loc* to *Char* assigns characters to memory locations. Elements of *Addr* will be called *addresses*. A dynamic function *AddrCont* assigns memory locations to addresses. (According to C, pointer addresses are composed of memory locations and *AddrCont* is defined by *LocCont*, but this information is not necessary for explaining $\mathcal{P}$ and will be ignored.) We assume that the two content functions are total (on their respective domains).

Q: An address probably identifies a little block of a memory where the address of a pointer can be stored. What is the difference between *Addr* and *Loc*?

A: You are right of course. The size of the block depends on the implementation. For our purposes, there is no reason to go into these details.

To reflect (a somewhat simplified version of) the parse tree for $\mathcal{P}$ corresponding to the official grammar of C and shown in Figure 2, $\mathcal{A}$ has a universe *Parsetree* that comprises nodes of the parse tree. A distinguished element *Root* and unary functions *Child1, Child2, Child3, Parent* have the obvious meaning. In addition, there is a universe of labels. A function *Label* assigns labels to nodes as shown in Figure 2. (The labels do not include expressions in parentheses which are added for greater clarity.) Each label is a distinguished element and therefore can be mentioned in rules.

In order to simulate $\mathcal{P}$, we should know at each moment where we are in the program. To this end, a dynamic distinguished element C will be the currently active node of *Parsetree*. Initially, C is the root. It will be convenient to abbreviate *Child1(C), Child2(C), Child3(C)* as *C1, C2, C3* respectively. Let me write a few transition rules governing the movement of C. I presume that C never stays two moments in a row at the same place; in other words, it will never have the same value at two consecutive states.

if C = *Root* **then**
    if *Val(C1)* = *undef* **then** C := *C1*
    elseif *Val(C2)* = *undef* **then** C := *C2*
    elseif *Val(C3)* = *undef* **then** C := *C3*
    endif
endif

Q: What is *Val*?

A: *Val* is a function on *Parsetree*. Think about each node n of the parse tree as the root of the corresponding subtree. When the code corresponding to the subtree has been executed, the resulting value is assigned to n. For example, the execution in

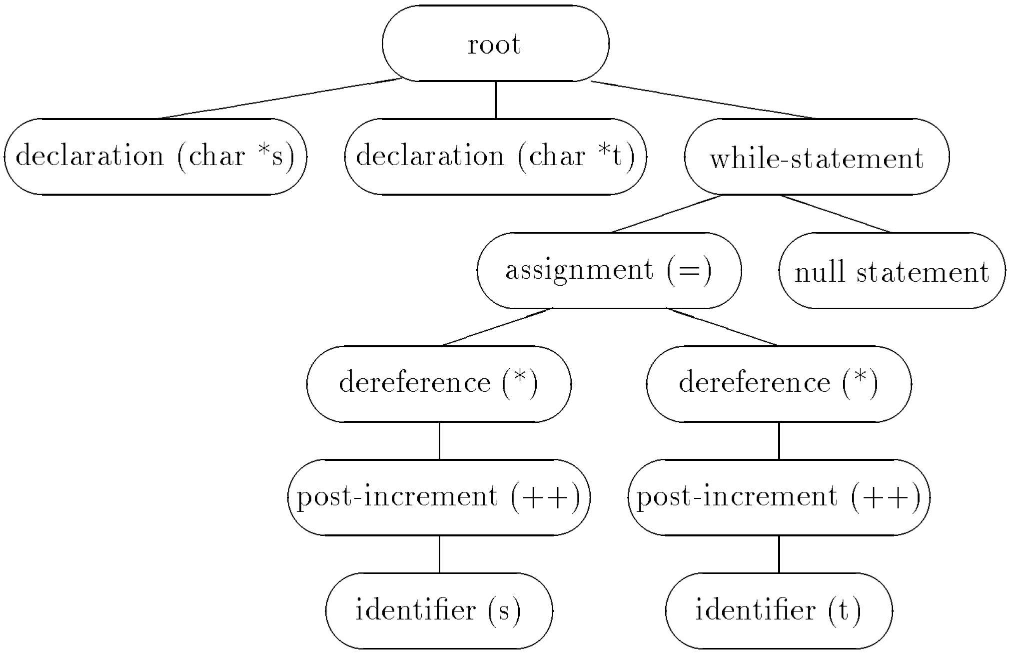


Figure 2: Our parse tree for `strcpy`.

question may be an expression evaluation; then the value of the expression is assigned to n. Initially, *Val* is nowhere defined.

Q: What if n corresponds to a statement that does not return any value?

A: When the statement has been executed, the value *done* will be assigned to n.

Q: What will be the value of a declaration node?

A: You'll see. Here are some additional rules governing the movement of C and using some obvious abbreviations.

if *C is a leaf* **then** *C := Parent(C)* **endif**

if *C has exactly one child* **then**
    if *Val(C1) = undef* **then** *C := C1*
    else *C := Parent(C)*
            Val(C1) := undef
    endif
endif

Q: I understand that the case of the while-statement node is an exception, but what about the case of the assignment node? Is it treated the same way as the case of the root node?

A: No. The problem is that the definition of C does not specify whether the left or the right expression is evaluated first. This was a deliberate decision [KR, pp. 53–54]; the order of evaluation is left to the implementer. To reflect the implementer's decision, we use an external function *ChooseChild*.

if *Label(C)* = *assignment* **then**
    if *Val(C1)* = *undef* **and** *Val(C2)* = *undef* **then** *C:=ChooseChild(C)*
    elseif *Val(C1)* = *undef* **then** *C* := *C1*
    elseif *Val(C2)* = *undef* **then** *C* := *C2*
    else *C* := *Parent(C)*
            Val(C1) := *undef*
            Val(C2) := *undef*
    endif
endif

Q: Why do you reset the values of *C1, C2* to *undef*?

A: To have a proper environment when *C* comes down to the assignment node next time.

Q: I do not see why do you need *ChooseChild*. Since you have allowed nondeterminism to resolve contradicting rules, you may as well use it. To choose a child, just say that *C* gets *C1* if *Val(C1)* is undefined and that *C* gets *C2* if *Val(C2)* is undefined. If the values of both *C1* and *C2* are undefined then one of them is nondeterministically chosen.

A: You are right. As an implementor, I would prefer the *ChooseChild* version though. It is more explicit. Now, let me give you the remaining rules. They are grouped with respect to the location of *C*.

if *Label(C)* = *declaration* **then**
    Val(C) := *ChooseAddr*
    AddrCont(ChooseAddr) := *PredefinedLoc(C)*
endif

Here *ChooseAddr* is a zero-ary external basic function that returns an address.

Q: Why should it be external?

A: $\mathcal{P}$ doesn't tell us how to allocate addresses.

Q: You suppose, I guess, that both occurrences of *ChooseAddr* evaluate to the same value.

A: Yes, *ChooseAddr* has a definite value in each state (where it is evaluated).

Q: And what is the *PredefinedLoc* function? You did not mention it earlier.

A: A call to `strcpy` should provide two parameters, namely, the initial values of `s` and `t`. *PredefinedLoc* carries this information.

Q: Is it an internal or external function?

A: In this case, the distinction does not matter. Since the two parameters should be known when $\mathcal{P}$ starts, *PredefinedLoc* can as well be an internal function, a static internal function. Further,

if *Label(C) = identifier* **then** *Val(C) := AddrCont(Val(Decl(C)))* **endif**

Q: What is *Decl*?

A: *Decl* maps each identifier node to the declaration node where the identifier in question is declared. The left identifier node is mapped to the left declaration node and the right identifier node is mapped to the right declaration node. Here is the rule for post-increment nodes.

if *Label(C) = post-increment* **and** *Val(C1)* $\neq$ *undef* **then**
  Val(C) := Val(C1)
  AddrCont(Val(Decl(C1))) := Val(C1) + 1
endif

Q: Let me understand this complicated second update. It would be easier for me to speak about a specific identifier node, say, the left one, which is related to the pointer variable `s`. Then *Decl(C1)* is the left declaration node and *Val(Decl(C1))* is the address of `s`. On the other hand, *Val(C1)* is the memory location pointed to by `s`. Thus, we are changing `s` to point to the next location (the successor of the old value of `s`).

A: That is right. Notice that *Val(C)* does not reflect the change. But the next time when our identifier node becomes active, it will have a new value.

The dereferencing operator (also known as the indirection operator) of C is formalized by our *LocCont* function. However, in the case of the left child of the assignment node, we are not really interested in dereferencing.

if *Label(C) = dereference* **and** *Val(C1)* $\neq$ *undef* **then**
  if *C = Child1(Parent(C))* **then** *Val(C) := Val(C1)* **endif**
  if *C = Child2(Parent(C))* **then** *Val(C) := LocCont(Val(C1))* **endif**
endif

The assignment-node rule should be clear now.

if *Label(C) = assignment* **and** *Val(C1)* $\neq$ *undef* **and** *Val(C2)* $\neq$ *undef* **then**
  LocCont(Val(C1)) := Val(C2)
  Val(C) := Val(C2)
endif

Q: I do not understand why the result of the assignment statement is *Val(C2)* rather than simply *done*.

A: As I mentioned before, an assignment statement returns the assigned value. This feature allows C programmers to use commands like $a = b = c = 0$. We take advantage of this feature to indicate when the string copy operation should halt. When *null* has been copied, it is passed up to the while-statement node halting execution of the `while` statement.

if *Label(C) = while-statement* **then**
    if *Val(C1) = undef* **then** *C := C1*
    elseif *Val(C1)* $\neq$ *null* **then**
        C := C2
        Val(C1) := undef
    else *C := Parent(C)*
        Val(C) := done
        Val(C1) := undef
    endif
endif

Q: Is all that a part of your EA description of C?

A: No, not really, but the description used to look like that.

8. Sequential and nonsequential evolving algebras

Q: Now I understand what the program of a sequential EA is; it is essentially a finite set of conditional function updates. But I still do not understand what a sequential EA is exactly. Do you identify a sequential EA with its program?

A: It is reasonable to identify sequential EAs and their programs and this is done sometimes [GR]. Often it is convenient, however, to avoid a formal definition. For example, many authors define precisely the language of a first-order theory, the set of theorems, etc. but leave the notion of first-order theory itself informal.

Q: What do you gain by leaving the notion of sequential EAs informal?

A: Usually, constructing an EA for a given algorithm $\mathcal{F}$ involves more than just writing a program. You indicate a class of static structures which includes presumably the presentations of all possible or all relevant states of $\mathcal{F}$. You may want also to indicate a class of initial static structures. See for example how Turing machines are formalized above. It may be convenient to view all that as a part of the definition of the EA.

Q: What non-basic rules are most common?

A: Most common is the generalization of basic rules where the terms are allowed to have free individual variables. This generalization is used even in the sequential case (as a syntactic sugar) to make the program more succinct [BRo1 and BRo2], but it is indispensable in the case of unbounded parallelism. Consider, for example, a tree algorithm that colors red – in one step – all children of the currently active node whenever the active node is green. This may be expressed by the rule

if *Color(C) = green* **and** *x is a child of C* **then**
    Color(x) := red
endif

which is not equivalent to any (finite) set of basic rules. In the paper on Occam, we used (limited) universal quantification [GMs, BRi1]. To show why one may need universal quantification, let us suppose that the tree algorithm, mentioned above, colors the active node C green if all children of C are red. This may be expressed by the rule

if *($\forall$ child x of C)(Color(x) = red)* **then**
    Color(C) := green
endif

Q: You mentioned asynchronous distributed computing a while ago. I have been wondering; how do you deal with such computations? It seems that going from one state to another is a basis of the EA approach. In the case of a distributed asynchronous computation, it may be unclear what the states are.

A: Syntactically, an evolving algebra for a distributed asynchronous algorithm may look like one that works in (discrete) sequential time. The difference is in the definition of a run. Instead of one demon, you may have a multitude of demons responsible for different rules. Whenever conditions are right, demons perform the required changes, but some demons may work with lightning speed whereas some others may be lazy. Let me again refer you to the paper [GMs] with Larry Moss on the programming language Occam. By the way, the sequential interpretation and the parallel interpretation of a list of commands coexist in Occam as well as in Parlog. This is one way to generalize the basic language of transition rules.

Q: Can one compose evolving algebras?

A: Of course [GR], but this is a topic for another conversation.

Acknowledgement. The EA project would probably die without the active participation of Egon Börger. I benefited greatly from the collaboration with Andreas Blass, Larry Moss and Dean Rosenzweig. The `strcpy` example was "lifted" from Jim Huggins' homework for the Principles of Programming Languages class. I am very greatful to Bob Blakley, Erich Grädel, Angelica Kappel, Gerti Kappel, Anatol Slissenko, Ed Wimmers and Moshe Vardi for their comments.

References

[BB] Christoph Beierle and Egon Börger, "Correctness proof for the WAM with types", Springer LNCS 626 (1992), 15–34.

The Börger-Rosenzweig's correctness proof for compiling Prolog to WAM [BRo1] is extended to the Protos-L language which extends Prolog with polymorphic (dynamic) types.

[Bl] Bob Blakley, "A Smalltalk Evolving Algebra And Its Uses", Ph. D. Thesis, University of Michigan, 1992.

An early student work on evolving algebras (the late date of 1992 is accidental). A reduced version of Smalltalk (called Mumble) is formalized and studied.

[Bo1] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control", CSL'89, 3rd Workshop on Computer Science Logic (eds. E. Börger et al.), Springer LNCS 440 (1990), 36–64.

[Bo2] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part II: Built-in Predicates for Database Manipulations", in Proc. of Mathematical Foundations of Computer Science 1990 (ed. B. Rovan), Springer LNCS 452, 1–14.

[Bo3] Egon Börger, "A Logical Operational Semantics of Full Prolog. Part III: Built-in Predicates for Files, Terms, Arithmetic and Input-Output", in: Y. Moschovakis (ed.), "Logic from Computer Science", Mathematical Sciences Research Institute Publications, vol. 21, Springer Verlag, 1992, 17–50.

These are the original 3 papers of Börger where he gives a complete evolving-algebra formalization of Prolog with all features discussed in the WG17 of ISO. An improved (tree based) version is found in [BRo2].

[BD] Egon Börger and K. Daessler, "PROLOG: DIN Papers for Discussion", in ISO/IEC JTCI SC22 WG17 No. 58, National Physical Laboratory, Middlesex, 1990, 114 pages.

A version of [Bo1–3] proposed to WG17 as a complete formal semantics of Prolog.

[BRi1] Elvinia Riccobene and Egon Börger, "A Formal Specification of Parlog", in "Semantics of Programming Languages and Model Theory" (eds. M. Droste and Y. Gurevich), Gordon and Breach, 1993, to appear. [Preliminary version in Springer LNCS 567 (1992) and 592 (1992).]

An evolving algebra formalization of Parlog, a well known parallel version of Prolog.

[BRi2] Elvinia Riccobene and Egon Börger, "A Mathematical Model of Concurrent Prolog", CSTR-92-15, Dept. of Computer Science, University of Bristol, 1992.

An evolving algebra formalization of Ehud Shapiro's Concurrent Prolog.

[BRo1] Egon Börger and Dean Rosenzweig, "The WAM – Definition and Compiler Correctness", TR-14/92, Dipartimento di Informatica, Universita di Pisa, June 1992, 57 pages. [Previous version in Springer LNCS 533 and 592.]

The first substantial example of the successive refinement method in the area. A hierarchy of evolving algebras provides a solid foundation for constructing provably correct compilers from Prolog to WAM. Various refinement steps take care of different distinctive features ("orthogonal components" in the authors' metaphor) of WAM making the specification as well as the correctness proof modular and extendible; an example of such an extension is found in [BB].

[BRo2] Egon Börger and Dean Rosenzweig, "A Simple Mathematical Model for Full Prolog", TR-33/92, Dipartimento di Informatica, Universita di Pisa, October 1992, 23 pages.

An improved version of [Bo1–3].

[BS] Egon Börger and Peter Schmitt, "A formal operational semantics for languages of type Prolog III", CSL'90, 4th Workshop on Computer Science Logic (eds. E. Börger et al.), Springer LNCS 533, 1991, 67–79.

An evolving algebra formalization of Alain Colmerauer's constraint logic programming language Prolog III.

[GR] Paola Glavan and Dean Rosenzweig, "Communicating Evolving Algebras — Part 1", Manuscript, University of Zagreb, Croatia, October 1992.

The first part of a two-part paper, presented at CSL'92, aimed to "demonstrate the power of the framework by providing simple and transparent models of the (polyadic) π-calculus of Milner, the Chemical Abstract Machine of Berry and Boudol, and TCCS of de Niccola and Henessy".

[GKS] Georg Gottlob, Gerti Kappel and Michael Schrefl, "Semantics of Object-Oriented Data Models – The Evolving Algebra Approach", in "Next Generation Information Technology" (eds. J. W. Schimdt and A. A. Stogny), Springer LNCS 504, 144–160.

"In this paper we show how evolving algebras can be used in particular to define the operational semantics of object creation, of overriding and dynamic binding, and of inheritance at the type level (type specialization) and at the instance level (object specialization)."

[Gr] George Gratzer, "Universal Algebra", Van Nostrand, 1968.

[Gu1] Yuri Gurevich, "Logic and the challenge of computer science", In "Current Trends in Theoretical Computer Science" (ed. E. Börger), Computer Science Press, 1988, 1–57.

The introduction and the first use of evolving algebras (at the end of the paper).

[Gu2] Yuri Gurevich, "Algorithms in the world of bounded resources", in "The universal Turing machine – a half-century story", (ed. R. Herken), Oxford University Press, 1988, 407–416.

[Gu3] Yuri Gurevich, "Kolmogorov machines and related issues: The column on logic in computer science", Bulletin of EATCS, No. 35, June 1988, 71–82.

[GMr] Yuri Gurevich and James Morris, "Algebraic operational semantics and Modula-2", CSL'87, 1st Workshop on Computer Science Logic, Springer LNCS 329 (1988), 81–101.

An abstract of [Mo].

[GMs] Yuri Gurevich and Larry Moss "Algebraic Operational Semantics and Occam", Springer LNCS 440, 176–192.

The first application of evolving algebras to distributed parallel computing with the challenge of true concurrency.

[GH] Yuri Gurevich and James Huggins, "The Evolving Algebra Semantics of C", CSE-TR-141-92, EECS Department, University of Michigan, 1992.

The method of successive refinements is used to give a more succinct semantics of the C programming language.

[JM] David E. Johnson and Lawrence S. Moss, "Dynamic Syntax", 1993 (to appear).

Distributed Evolving Algebras are used to model formalisms for natural language syntax.

[KS] Gerti Kappel and Michael Schrefl, "Cooperation Contracts", Proc. 10th International Conference on the Entity Relationship Approach (ed. T. J. Teorey), San Mateo, California, 1991, ER Institute, 285–307.

The authors introduce the concept of cooperative message handling and use evolving algebras to give formal semantics.

[KR] Brian W. Kernighan and Dennis M. Ritchie, "The C Programming Language", Prentice-Hall, 1988, Englewood Cliffs, NJ.

[KU] A. N. Kolmogorov and V. A. Uspensky, "To the Definition of an Algorithm", Uspekhi Mat. Nauk 13:4 (1958), 3–28 (Russian); English translation in AMS Translations, ser. 2, vol. 21 (1963), 217–245

[Mo] James Morris, "Algebraic operational semantics for Modula 2", Ph.D. thesis, University of Michigan, 1988.

The earliest formalization of a real-life language. In the meantime, the methodology has developed enabling more elegant descriptions, but one has to start somewhere.

[Pr] Vaughan R. Pratt, "Dynamic algebras and the nature of induction", 12th ACM Symposium on Theory of Computation, 1980.

[Sch] Arnold Schönhage A, "Storage Modification Machines", SIAM J. on Computing 9:3 (1980), 490–508.

[Tu] Alan M. Turing, "On computable numbers with an application to the Entscheidungsproblem," Proc. London Math. Soc. 42 (1937) 230–265; correction, *ibid*, No. 43 (1937), 544–546.

[WG17] "PROLOG. Part 1, General Core, Committee Draft 1.0", ISO/IEC JTCI SC22 WG17 No. 92, 1992.

Evolving Algebras and Linear Time Hierarchy*

Andreas Blass†and Yuri Gurevich‡

Keyword Codes: F.1.1; F.1.3; F.3.2
Keywords: Computation by Abstract Devices, Models of Computation;
Computation by Abstract Devices, Complexity Classes;
Logics and Meanings of Programs, Semantics of Programming Languages

1 Introduction

A program Q simulates a program P in *lock-step* under a given representation ρ of states of P as states of Q if there is a constant c such that if P transforms a state A to a state A^+ in time τ then Q transforms $\rho(A)$ into $\rho(A^+)$ in time $\leq c\tau$. (*Cf.* the related notions of real-time computation and simulation [R,S].) The constant c is the *lag factor* of the simulation. We speak about time rather than the number of steps because the number of steps may be too crude a measure of time and a more honest measure may be required. The representation function ρ may be multi-valued, in which case Q transforms any $\rho(A)$ into some $\rho(A^+)$. The inverse function ρ^{-1} is usually single-valued.

Neil Jones [J] calls a universal program U for a language L *efficient* if there is a constant c such that U simulates every L-program in lock-step with lag factor c. He exhibits a programming language I^{su} that admits an efficient universal program and then uses such a program and diagonalization to show that functions I^{su}-computable in linear time form an infinite hierarchy. To show the robustness of his results, Jones mentions a number of languages and machine models which admit efficient universal programs and give the same class $\mathrm{LIN}^{\mathrm{su}}$ of problems solvable in linear time. Among those machines models are the storage modification machines of Schönhage [S] and successor RAMs, that is, random access machines whose only arithmetical operation is the operation of computing the successor of a given natural number.

Jones's thought-provoking paper seems to open a possibility of proving linear time lower bounds for linear time computations. Unfortunately linear time on successor RAMs is too restrictive for many applications, e.g., computational geometry, and in general linear time is not very robust notion. Should one try to extend Jones's results to addition RAMs, or to addition and multiplication RAMs, or to addition, multiplication and division RAMs, or to something else?

The most versatile machine model we know is evolving algebras (ealgebras, or EAs) [G1,G2]. According to the EA thesis, any algorithm can be simulated by an appropriate ealgebra in lock-step. In this paper we transfer Jones's results mentioned above to evolving algebras. We restrict attention to sequential deterministic ealgebras (without variables); more general ealgebras will be considered elsewhere. The definition of ealgebras in the tutorial [G1] is sufficient for our purposes in this paper. For the reader's convenience, we include in Section 2 a brief review of ealgebras.

Our first result is a universal ealgebra U. (Another universal ealgebra U has been constructed by Anne Preller [P].) To avoid distraction from the main idea, the presentation of U in Section 3 is simplified in that we ignore external functions.

*In *Proceedings of IFIP Congress 94 – Volume 1*, eds. B. Pehrson and I. Simon, Elsevier, 1994.
†Partially supported by NSF grant DMS-9204276. Mathematics Department, University of Michigan, Ann Arbor, MI 48109-1003, USA. ablass@umich.edu
‡Partially supported by NSF grant CCR 92-04742 and ONR grant N00014-91-J-11861. EECS Department, University of Michigan, Ann Arbor, MI 48109-2122, USA. gurevich@umich.edu

Our main result is the diagonalization theorem for ealgebras formulated in Section 4. It implies the linear time hierarchy theorem for ealgebras. Our proof of the diagonalization theorem cannot be given here because of the lack of space.

Finally, we consider quickly other machine models in Section 5. Schönhage machines are essentially unary ealgebras whose only external function is the nullary function New which brings from the Reserve new elements as needed. The diagonalization and linear hierarchy theorems for Schönhage machines easily follow from those for ealgebras. One has to work a little harder to derive the diagonalization and linear time hierarchy theorems for various sufficiently powerful RAM models (e.g. RAMs with addition and multiplication) from those for ealgebras.

2 Evolving Algebras

We recall the definitions of commands. An update instruction is an expression of the form

$f(t_1, \ldots, t_r) := t_0$

where each t_i is a ground term. Transition rules are defined recursively. Any update instruction is a rule. If k is a natural number, $g_0, \ldots, g_k$ are terms and $R_0, \ldots, R_k$ are sequences of rules then the expression

$$\begin{array}{l}\texttt{if } g_0 \texttt{ then } R_0 \\ \texttt{elseif } g_1 \texttt{ then } R_1 \\ \vdots \\ \texttt{elseif } g_k \texttt{ then } R_k \\ \texttt{endif}\end{array} \qquad (*)$$

is a (compound) rule. If the last guard g_k is *true* then the last elseif clause can be abbreviated to

$\texttt{else } R_k$

A program P is a sequence of rules. A state of a program P is a static algebra, i.e., a nonempty set (the *carrier*) equipped with interpretations of the function symbols that occur in P. A special distinguished element *undef* is used to deal with partial functions. Distinguished elements *true* and *false* are used to treat predicates as functions. A unary predicate may be regarded as defining a special universe within the carrier.

To execute an update instruction $f(t_1, \ldots, t_r) := t_0$ in a state A, evaluate all terms t_i is A and set f at $(A(t_1), \ldots, A(t_r))$ to $A(t_0)$. To execute a sequence $u_1, \ldots, u_k$ of update instructions at A, evaluate all the relevant terms in A and then make all the changes. What happens if some update instructions contradict each other at A? In this paper, we use eager execution to resolve conflicts. Make all the changes from left to right; later changes may override earlier ones. For example, if $k = 2$, u_1 is $f(a) := 3$, u_2 is $f(b) := 5$ and $A(a) = A(b) = 1$ then $f(1)$ is set first to 3 and then finally to 5. At a given state, the compound rule $(*)$ reduces to a sequence of update instructions obtained recursively as follows. Find the first i (if any) for which $A(g_i) = true$ and reduce all the rules in R_i to sequences of update instructions. If there is no such i then $(*)$ yields the empty sequence. To execute P at a state A, execute the concatenation of the update instruction sequences produced by the rules in P.

A pure run of the ealgebra with program P is a sequence of states in which each except the initial state is obtained by executing P in the previous state. "Pure" refers to the run's being uninfluenced by any external environment. If there are environmental influences, they change certain functions, called external functions. A typical example of an external function is the input provided by the user. Another example is the nullary function New mentioned above.

Time A relatively obvious EA program U simulates an arbitrary EA program P in lock-step where the lag factor c depends on P. Is it possible to make c independent of P? The answer is no as long as we identify time with the number of steps. The number of updates performed by U in c steps is bounded and thus U cannot compute the next state of P in c steps if P performs more than c updates. However this answer is misleading. Ealgebras were designed to make the specification job easier. Introducing more functions and rules, one may increase forever the amount of work that an ealgebra is able to do in one step. We need a more honest measure of time. The following definition will do.

Given a state A, we define inductively the time needed to compute and execute at A. The time to compute a term $f(t_1, \ldots, t_r)$ is

$$\mathrm{Time}(f(t_1, \ldots, t_r), A) = 1 + \sum_{i=1}^{r} \mathrm{Time}(t_i, A).$$

The time to execute an update instruction $f(t_1, \ldots, t_r) := t_0$ is

$$\mathrm{Time}((f(t_1, \ldots, t_r) := t_0), A) = 1 + \sum_{i=0}^{r} \mathrm{Time}(t_i, A).$$

Let C be a clause "(else)if g then $R_1, \ldots, R_k$". The time to execute C is

$$\mathrm{Time}(C, A) = \begin{cases} 1 + \mathrm{Time}(g, A) + \sum_{i=1}^{k} \mathrm{Time}(R_i, A), & \text{if } A(g) = \mathit{true}; \\ 1 + \mathrm{Time}(g, A), & \text{otherwise.} \end{cases}$$

The time to execute a compound rule R with clauses $C_0, \ldots, C_k$ is

$$\mathrm{Time}(R, A) = 1 + \sum_{i=0}^{j} \mathrm{Time}(C_i, A),$$

where j is the first i such that the guard g_i in C_i evaluates to *true* in A, and $j = k$ if there is no such i. Let P be a program with rules $R_1, \ldots, R_k$. The time to execute one step of P at state A is:

$$\mathrm{Time}(P, A) = 1 + \sum_{i=1}^{k} \mathrm{Time}(R_i, A).$$

The time of a run $A_0, \ldots, A_k$ of program P is $\displaystyle\sum_{i=0}^{k-1} \mathrm{Time}(P, A_i)$.

3 Universal Ealgebra

In the spirit of the EA approach, we would like to avoid coding as much as possible. However, some coding is necessary in order to represent an arbitrary ealgebra P together with a state A of P as a state $P + A$ of a universal program.

Parse Trees An *oriented tree* is a single-universe static algebra whose vocabulary consists of the nullary function root and the unary functions Parent, FirstChild, LastChild and NextSib. The interpretations are obvious; in particular, Parent(root)=*undef*. A *labeled tree* is obtained by adding a new universe of labels and a labeling function from tree nodes to labels. To *compose* labeled trees $T_1, \ldots, T_k$ under label w, take disjoint copies of $T_1, \ldots, T_k$ and make their roots children of a new root labeled w; denote the result $\mathcal{C}(T_1, \ldots, T_k; w)$.

By one induction, we define parse trees $\mathrm{PT}(s) = \mathrm{ParseTree}(s)$ for various syntactical objects s related to a program P. These parse trees are labeled trees, and the universe of labels consists of the function symbols in the vocabulary of P and the special labels "update", "clause", "compound-rule", and "program".

$$\mathrm{ParseTree}(f(t_1, \ldots, t_r)) = \mathcal{C}(\mathrm{PT}(t_1), \ldots, \mathrm{PT}(t_r); f).$$

If u is an update instruction $f(t_1,\ldots,t_r) := t_0$ then

$$\text{ParseTree}(u) = \mathcal{C}(\text{PT}(f(t_1,\ldots,t_r)), \text{PT}(t_0); \text{update}).$$

If C is a clause "(else)if g then $R_1,\ldots,R_m$" then

$$\text{ParseTree}(C) = \mathcal{C}(\text{PT}(g), \text{PT}(R_1),\ldots,\text{PT}(R_m); \text{clause}).$$

If R is a compound rule with clauses $C_1,\ldots,C_k$ then

$$\text{ParseTree}(R) = \mathcal{C}(\text{PT}(C_1),\ldots,\text{PT}(C_k); \text{compound-rule}).$$

Finally, if the program P is a sequence of rules $R_1,\ldots,R_k$ then

$$\text{ParseTree}(P) = \mathcal{C}(\text{PT}(R_1),\ldots,\text{PT}(R_k); \text{program}).$$

The Apply Function A program P may have functions of higher arity than those of the universal program U. Thus the rules of U cannot deal directly with functions of P. We solve this problem by allowing the state $P+A$ to have not only elements of A but also sequences of those elements of length $\leq r$ where r is the highest arity of P's functions.

The vocabulary of our universal algebra contains the nullary function name Nil and the binary function names Append, Apply among other symbols. Nil and Append are interpreted in the obvious way; in particular, Append$(s,x) \neq$ *undef* only if s is a sequence of length $< r$ and x is a single element. If f belongs to the universe of labels and is one of the function symbols and s is a sequence whose length equals the arity of f then Apply$(f,s) = A(f(s))$ in a state $P+A$.

The Strategy for a Universal Program States $P+A$ sketched above are called *pose* states. When P makes one step and transforms A to the next state A^+, U performs a computation

$$B_0 = P+A, B_1,\ldots,B_k = P+A^+$$

going from a pose state B_0 to a pose state B_k via nonpose states $B_1,\ldots,B_{k-1}$. To make the distinction between pose and nonpose states sharp, we specify a special pose term in the vocabulary of U such that a state B of U is a pose state if and only if the pose term evaluates to *true* at B.

Theorem 3.1 *There exist an EA program U and a constant c such that U simulates any EA program P in lock-step with lag factor c.*

The computation of U splits into two phases: evaluate and update. During the evaluation phase the parse tree of P is traversed in the depth-first fashion except that certain parts are skipped because of the control structure of compound rules (specifically, elseif) and the values of guards. All visited terms are evaluated and relevant updates are marked. During the update phase, the marked updates are executed. The nullary function C (an allusion to "control") gives the current position in the parse tree of P.

Abbreviations and Aliases

F, L, N, Pa, S abbreviate
FirstChild(C), LastChild(C), NextSib(C), Parent(C), Status.
"C is the last sib" stands for NextSib(C)=*undef*.
"C is a term" stands for Vocabulary(Label(C)). The same for Pa.
"C is a clause" stands for Label(C)=clause. The same for Pa.
"C is a guard" abbreviates "C is a term and Pa is a clause"
FU, NU(C), LU abbreviate
FirstActiveUpdate, NextActiveUpdate(C), LastActiveUpdate.
Val(C) abbreviates Apply(Label(C),Arg(C))

Initial State It is supposed that the initial state of U satisfies the following conditions:

C=root and therefore Label(C)=program; halt=*false*; phase=evaluate; S=start.
Arg(x)=ValSeq(x)=Nil for all x such that Term(Label(x)) holds.
FU, LU are undefined. NU(x) is nowhere defined.

The pose term is "Label(C)=program and S=start". In particular, the initial state is a pose state, and we assume that it satisfies an earlier description of pose states.

Rules Advance Rule

```
if phase=evaluate and ¬(halt) then
   if S=start then
      if F=undef then S:=finish else C:=F endif
   elseif C is the last sib or
         C is a guard and Val(C)≠true or
         C is a clause and Val(F)=true then
      C:=Pa
   else C:=N, S:=start endif
endif
```

Advance Rule regulates the movement of C during the evaluation phase. The nullary function Status with values *start* and *finish* indicates whether whether the evaluation of the subtree rooted at C starts or finishes.

Root Rule

```
if Label(C)=program and ¬(halt) and phase=evaluate and S=finish then
   if FU=undef then halt:=true else phase:=update, C:=FU, FU:=undef endif
endif
```

Root Rule applies at the end of the evaluation phase. If FU=undef then there is and there will be nothing to update; the universal program halts. Otherwise, C jumps to FU and the update phase starts.

Term Rule

```
if C is a term and S=finish and Pa is a term then
   if C≠LastChild(Pa) then ValSeq(N):=Append(ValSeq(C),Val(C))
   else Arg(Pa):=Append(ValSeq(C),Val(C)) endif
endif
```

Term Rule applies at the end of the evaluation of a compononent t_i of a term $f(t_1, \ldots, t_r)$. At this moment ValSeq(C) is the value of $(t_1, \ldots, t_{i-1})$. The value of $(t_1, \ldots, t_i)$ is computed. If $i < r$, that value is stored as ValSeq(N) at the next sibling N of C; otherwise it is stored at the parent node Pa as Arg(Pa).

Update Rule

```
if Label(C)=update then
   if phase=evaluate and Val(L)≠Apply(Label(F),Arg(F)) then
      if FU=undef then FU:=C, LU:=C else NU(LU):=C, LU:=C endif
   else
      Apply(Label(F),Arg(F)):=Val(L)
      if Defined(NU(C)) then C:=NU(C), NU(C):=undef
      else C:=root, phase:=evaluate, S:=start, LU:=undef endif
   endif
endif
```

Update Rule applies when C is at the root of an update parse tree. During the evaluation phase, relevant updates are marked for future execution. The very first relevant update is marked with FU. The function NU is set to lead from one relevant update to the next one. LU marks the last relevant update. During the update phase, the marked updates are executed starting with FU and finishing with LU. As each update is executed, it becomes unmarked.

4 Diagonalization

Theorem 4.1 (Diagonalization) *Let σ be a vocabulary with sufficiently many nullary and unary function symbols. There exist a σ-program D and a constant c_σ such that, for any σ-program P, the run of D on input ParseTree(P) simulates in lock-step with lag factor c_σ the run of P on ParseTree(P) except that the output is changed.*

Theorem 4.2 (Linear Time Hierarchy) *For any constant a, there is a function computable by a σ-program in time $c_\sigma \cdot a \cdot n$ but not computable by any σ-program in time $a \cdot n$.*

The diagonalization is harder than constructing a universal machine U. U can use functions like Append and Apply unknown to P but D has no such luck. P may play with any functions available to D. Our D does not use sequences or the Apply function. Notice also that D cannot start by copying the whole input as the universal Turing machine does, as such copying might take much more time than P uses. If we were interested only in linear-time simulation, we could restrict attention to programs P that read their whole input. Then D would have enough time to copy the input before proceeding with the simulation. But in order to obtain lock-step simulation we must program D to copy pieces of its input when they are needed rather than all at once. Because of this, the pose states of D are not uniquely determined by the corresponding states of P. They do, however, uniquely determine the corresponding states of P. The notion of simulation of runs, used in Theorem 4.1, should be defined accordingly.

5 Other Machine Models

Schönhage Machines Schönhage machines [S] can be lock-step simulated by unary ealgebras (that is, ealgebras without functions of arity ≥ 2) whose only external function is the nullary function New which brings from the Reserve new elements as needed. To get a uniform lock-step simulation, one needs honest time counting for both Schönhage machines and ealgebras. Furthermore, every such unary ealgebra can be simulated in lock step by an appropriate Schönhage machine. Thus, the diagonalization and linear time hierarchy theorems for Schönhage machines [J] are consequences of Theorems 4.1 and 4.2.

Random Access Machines Consider RAMs that have a particular set of operations available for manipulating the contents and addresses of their registers and that have each register's content bounded in length by $O(\log n)$ where n is the input size. Such RAMs can be uniformly simulated in lock-step by an ealgebra that has those operations built into its initial state. Diagonalization and linear hierarchy theorems for sufficiently powerful RAMs can be deduced from Theorems 4.1 and 4.2 by designing RAMs to simulate the relevant ealgebras in lock-step.

References

G1 Yuri Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", Bull. EATCS, 43 (1991), 264–284, and (slightly revised) in "Current Trends in Theoretical Computer Science", Eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292.

G2 Yuri Gurevich, "Evolving Algebras 1993: Lipari Guide", in "Specification and Validation Methods", Ed. E. Börger, Oxford University Press, to appear.

J Neil D. Jones, "Constant Time Factors *Do* Matter", ACM Symp. on Theory of Computing, 1993, 602–611

P Anne Preller, Private Communication, February 1994.

R Michael Rabin, "Real Time Computations", Israel J. of Math., 1:4 (1963), 203–211.

S Arnold Schönhage, "Storage modification machines", SIAM J. on Computing, 9 (1980), 490-508.

Evolving Algebras*

Yuri Gurevich†

July 21, 1995

Keyword Codes: D.2.2; F.1.1; F.3.2
Keywords: Software Engineering, Tools and Techniques;
Computation by Abstract Devices, Models of Computations;
Logics and Meanings of Programs, Semantics of Programming Languages

1 Introduction

I am delighted to open the first workshop devoted entirely to evolving algebras. For the sake of brevity, I shorten the term "evolving algebra" to "ealgebra" (pronounce e-algebra) or "EA".

The main goal of the evolving algebra methodology is to provide succinct and executable formal specifications for algorithms. Here the term algorithm is taken in a broad sense including programming languages, architectures, distributed and real-time protocols, etc.. Some of my engineering colleagues say that formal methods had their chance and failed. This is not true. Some particular methods might have been over-advertised and failed to live up to the great promises. I believe in a great future of formal methods and in particular the EA method in designing, testing, verifying, documenting, maintaining, understanding and teaching algorithms.

There are so many things I would like to tell you. For the EA experts, I would like to survey the main ideas, to draw their attention to open problems and to congratulate them on their achievements. Keep up with the good work. We have come quite a way in so short a while.

For the skeptical engineer, I would like to demonstrate that we have a practical tool. Do not be turned off by terms like "algebra" and "algorithms". EA methodology works and it may help you. Tell us your algorithm-related problems. Give us challenges.

For the computer science theoretician, I would like to explain the EA approach from first principles and to show that this computation model bridges between complexity theory and formal methods.

But I cannot do all that in a talk. All I can do is a brief overview. I will not even

*IFIP 13th World Computer Congress 1994, Volume I: Technology/Foundations, eds. B. Pehrson and I. Simon, Elsevier, Amsterdam.
†Partially supported by ONR grant N00014-91-J-1861 and NSF grant CCR-92-04742. EECS Department, University of Michigan, Ann Arbor, MI, 48109-2122, USA. gurevich@umich.edu

define ealgebras here; see* the tutorial [Gurevich91], henceforth called the tutorial, or the Lipari guide [Gurevich94b]. I will not examine the complexity theoretic aspects of the EA model; this is the topic of the next lecture. And I will skip many issues likely to be covered in the survey lectures by Egon Börger and Dean Rosenzweig; in particular I will not say much about logic programming where the EA methodology was so successfully applied and developed.

2 The EA Thesis

When you use evolving algebras, you develop a strong feeling that ealgebras can model nicely any algorithmic process. The EA thesis can be seen as an attempt to express that feeling in terms of the EA computation model. The EA thesis was also the initial goal of the EA project.

2.1 Sequential Computations

At the beginning, there was Turing's thesis. Turing's machine model allowed Turing to formalize the notion of computable functions. But are there computing devices that allow one to formalize the notion of algorithm? The idea of the EA thesis is that every algorithm is essentially an ealgebra.

The EA thesis can be stated as follows: Every algorithm can be simulated *on its natural abstraction level* by an appropriate ealgebra in strict lock-step, so that the ealgebra needs only one step to simulates a step of the algorithm.

In the case of sequential computations, we are talking about the basic evolving algebras of the tutorial. The case of non-sequential computations is discussed below.

How can one argue for such a thesis? I see two ways: speculative philosophy and experimentation. A speculative argument for the EA thesis in the case of sequential algorithms is sketched in the tutorial. A fair amount of experimentation is mentioned in the tutorial as well. More experimentation — both sequential and non-sequential algorithms — has been done in the meantime as the annotated bibliography shows. In particular, evolving algebras have been constructed for various programming languages including C, C++, Occam and major versions of Prolog including its extensions by constraints, types and parallelism.

The confidence in the EA thesis that some of us feel comes not only from the amount of experimentation. You look at as many different kinds of algorithms as you can and you see that a strict lock-step simulation is always possible and after a while you run out of challenges. During a discussion in a 1993 Dagstuhl workshop, Andreas Blass remarked that the situation reminds him of the situation with formalizing mathematics within the standard set theory in the beginning of this century.

2.2 Non-Sequential Computations

The situation with non-sequential computations is different. A weaker form of the EA thesis holds: For every well defined class of algorithm, there is a generalization of the basic EA

*The references in this abstract refer to "Annotated Bibliography on Evolving Algebras" in "Specification and Validation Methods", ed. E. Börger, Oxford University Press, to appear. The book is related to a 1993 summer school on Lipari Island, Italy, and will be called the Lipari volume.

model that “captures” that class. The problem is that at the current stage of computer science we do not yet clearly understand, for example, what parallel computations are, what concurrent computations are, or what real-time computations are.

The evolving algebra methodology seems to be well adaptable to any of these computation paradigms. Evolving algebras with free variables and numerous daemons can capture the concurrent asynchronous computations of Occam (see [GurMos90] by Larry Moss and myself) or Parlog, a parallel version of Prolog (see [BoeRic93] by Egon Börger and Elvinia Riccobene).

By the way, Egon Börger, Igor Durdanović and Dean Rosenzweig returned recently to Occam and used advances in the EA methodology, like stepwise refinement and hiding syntax in a flowchart, to construct another ealgebra for Occam from which they were able to make a provably correct, smooth transition to the Transputer Instruction Set architecture. A more abstract study of concurrent evolving algebras has been undertaken by Paola Glavan and Dean Rosenzweig in [GlaRos93].

A group membership protocol involving real-time constraints has been formally specified and verified in [GurMan94] by my student Raghu Mani and myself. You will find a number of other instructive EA simulations of non-sequential algorithms in the annotated bibliography; some of them will be presented later in this workshop and some are found in the Lipari volume. As I said above, we are running out of obvious challenges.

In the Lipari guide, I attempt to extend the definition of ealgebras and provide a more solid foundation for the quickly growing EA field. Hopefully, ealgebras will be used not only to model, say, concurrent computations but also to understand what concurrent computations are.

3 Remarks

A survey talk is appropriate for reflection, but I philosophized on evolving algebras before in various articles, so here I will restrict myself to a few remarks.

My fellow logicians often sniff at semantics provided by computing devices because of the ad hocish character of devices and because devices often implement algorithms at a lower abstraction level. I believe that there isn’t much ad hocish about ealgebras. The two main ingredients of the concept of evolving algebra belong to the well established tradition of mathematical logic. They are the notion of structure (as in first-order logic) and the notion of transition system. The novelty is in the way the two are put together. In my eyes, ealgebras were discovered, not invented.

Of course, a benefit of semantics provided by computing devices is that, as they say, you don’t need a PhD in logic to understand it. EA specifications indeed are readable. I use them without problems in my classes whether my students are good in mathematics or not. The important thing is that they know programming.

Concerning the abstraction levels, the EA thesis mentions the natural abstraction level of the algorithm. Indeed, we are not interested in specifying an algorithm by implementing it. We are interested in specifying the algorithm on its natural abstraction level. Fortunately, the EA approach allows us to do that. Why is this so important? Well, let me raise another question. How do you know that your model faithfully reflects reality? This is a sticky question. You can’t prove that a formal model fits informal reality. A good strategy is to keep your model as close to the original as possible, so that it is obvious that the modeling is correct. The ability to specify algorithms on their natural abstraction levels allows one to use ealgebras for requirement specifications.

Let me stress the universality of the EA method. Many competing systems are good in specific domains. The EA methodology seems to work across domains. As you work with ealgebras you begin to see other computational models through the EA lens and they begin to look like special evolving algebras.

Another important issue is separation of concerns. Separating statics and dynamics (that is, static and dynamic aspects of computation) is at the heart of the EA approach. Statics plays a minor role in the definition of ealgebras and often is taken for granted; we concentrate attention on dynamics. That is why the definition of basic ealgebras is so simple. Separating statics and dynamics a little further, the syntax of the C programming language has been hidden into a flowchart in [GurHug93] by Jim Huggins and myself; this helped to simplify the ealgebra.

Of course, statics may be important and nontrivial; you cannot ignore it completely. Indeed, any EA interpreter will require not only the program of an ealgebra but also a presentation of the initial state.

By the way, what is static and what is dynamic may be in the eye of the beholder. "We suggest ... that many grammatical frameworks are static formalizations of intuitively dynamic ideas", write David Johnson and Larry Moss in their paper [JohMos93] whose name speaks for itself: Grammar Formalisms Viewed as Evolving Algebras.

Another relevant separation of concerns was suggested and discussed recently in [Gurevich94a]: First establish your claim (say, some property of an algorithm, e.g., correctness) mathematically and only then translate it into a formal system and check it on a computer if necessary. Computer verification of mathematical proofs should be a separate science or at least a separate branch of logic; I call it pedantics for the time being. I know that many disagree with me on that separation of concerns. One advantage of that separation is that you can use freely any necessary mathematics at the first stage when you are searching for a proof.

A few words on the method of stepwise refinement in the EA context. Because of the freedom to choose the appropriate abstraction level inherent in the EA approach, the method of stepwise refinement is extremely useful. The use of stepwise refinement was pioneered by Egon Börger and Dean Rosenzweig. An impressive hierarchy of abstraction levels and the simplicity of single refinement steps allowed them to establish the correctness of a general compilation schema of Prolog programs to the Warren Abstract Machine which underlies most of the current Prolog implementations and incorporates crucial optimization techniques [BoeRos94a].

Sometimes an algorithm comes with a built-in hierarchy of abstraction levels which guides the refinement process. A good example of this is the well known Kermit communication protocol specified and proved correct by my student Jim Huggins in [Huggins94]. But usually one has to analyze the algorithm in order to separate concerns and figure out the right abstraction levels. A relatively simple (comparative to [BoeRos94a]) and instructive example of stepwise refinement is found in [GurHug93] where Huggins and I constructed an ealgebra for the C programming language. Compare this ealgebra to an earlier ealgebra for Modula-2 in [Morris88] to see the benefits of stepwise refinement and hiding syntax in a flowchart.

A simple and instructive example of the use of multiple abstraction levels for verification purposes is found in [BoGuRo94].

4 Future Work

There are plenty of things to do. Here are some of them.

Explore new application areas and subareas (and in this way continue to test the broader EA thesis). Regarding hardware, I should mention the paper [BoGlMu93] by Egon Börger, Uwe Glässer and Wolfgang Müller on VHDL; Jim Huggins is looking at Verilog, another hardware description language. Speaking about software, ealgebras may for example be a good way to formalize active databases. And so on.

In connection with new application, let me mention testing of algorithms. Testing may seem an unlikely application for a specification method even if the specifications are executable. Why would you execute the specification if you can experiment with the algorithm itself? First, you may experiment with a specification before the algorithm is implemented or when no implementation of the algorithm is available. Second, various environmental forces may be represented by parameters in the specification. This may give you have more opportunities to inject faults. You may play with more operating systems than you have around. You may play with physical parameters like gravitation. Third, you may combine testing and verification.

Develop a toolkit for writing and using ealgebras. We have an interpreter for sequential ealgebras in Michigan which is being upgraded; in particular we would like to be able to interleave more elegantly multi-agent ealgebras. We have also a partial evaluator [GurHug94]. Much more is needed.

Use the EA computation model for complexity investigation. Such investigation started with a paper [BlaGur94] by Andreas Blass and myself.

Focusing on the theory of ealgebras, develop a model theory of ealgebras. Isolate the principal refinement notions. What are the main ways to combine ealgebras? The investigation of the second problem started in [GlaRos93]. Develop the proof theory to go with ealgebras. Temporal logic may be relevant. A step in that direction is made in [Poetzsch94]. By the way, it is customary in temporal logic to start with a proof system and analyze the models. It may be fruitful to explore the other direction. Start with a class of evolving algebras and see what proof system is appropriate to it.

Acknowledgment

I thank Egon Börger, Dean Rosenzweig and Jim Huggins for help with this abstract; portions of the text are direct quotations from Egon Börger.

Evolving Algebras 1993: Lipari Guide*

Yuri Gurevich†

1 Introduction

Computation models and specification methods seem to be worlds apart. The evolving algebra project started as an attempt to bridge the gap by improving on Turing's thesis [G1, G2]. We sought more versatile machines which would be able to simulate arbitrary algorithms in a direct and essentially coding-free way. Here the term algorithm is taken in a broad sense including programming languages, architectures, distributed and real-time protocols, *etc.*. The simulator is not supposed to implement the algorithm on a lower abstraction level; the simulation should be performed on the natural abstraction level of the algorithm.

The evolving algebra thesis asserts that evolving algebras are such versatile machines. The thesis suggests an approach to the notorious correctness problem that arises in mathematical modeling of non-mathematical reality: How can one establish that a model is faithful to reality? The approach is to construct an evolving algebra $\mathcal{A}$ that reflects the given computer system so closely that the correctness can be established by observation and experimentation. (There are tools for running evolving algebras.) $\mathcal{A}$ can then be refined or coarsened and used for numerous purposes. An instructive example is described in [B] by Egon Börger who championed this approach and termed $\mathcal{A}$ the *ground model* of the system. The use of the successive refinement method is facilitated by the ability of evolving algebras to reflect arbitrary abstraction levels. This has been convincingly demonstrated by Börger and Rosenzweig in [BR]; a simpler example is found in [GH].

Evolving algebras have been used to specify languages (*e.g.* C, Prolog and VHDL), to specify real and virtual architectures (*e.g.* APE, PVM and Transputer), to validate standard language implementations (*e.g.* of Prolog, Occam), to validate distributed protocols (see examples in Parts III and IV of this book), to prove complexity results [BG], *etc.*. See Börger's annotated bibliography on evolving algebras in this book and the proceedings of the first evolving algebra workshop in [PS].

Here we extend the definition of evolving algebras given in the tutorial [G2] (henceforth "the tutorial"). For the sake of brevity, the term "evolving algebra" is often shortened to "ealgebra" (pronounced e-algebra) or "EA"; the latter term is used mostly as an adjective. Static algebras are discussed in §2. Sequential ealgebras are discussed in §3; first we define basic ealgebras and then we equip them with the ability to import new elements. Nondeterministic sequential ealgebras and some other simple extensions of basic ealgebras are discussed in §4, parallel ealgebras are discussed in §5, and distributed ealgebras are discussed in §6 which can be read immediately after §3. Admittedly this guide is harder to read than the tutorial, and we intend to write a more popular version of the guide.

Now let us return to the EA thesis. In the tutorial, we defined sequential ealgebras and sketched a speculative philosophical "proof" of the sequential version of the thesis. The definition of sequential ealgebras and the sequential EA thesis have survived several years of intensive application and experimentation. As a matter of fact, we (the EA community) seem to have run out of challenges.

The situation with non-sequential computations is more complicated. It seems that, for every reasonably understood class of algorithms, there is a natural extension of the basic EA model that "captures" that class. That form of the EA thesis also has survived several years of intensive application and experimentation. The philosophy and guiding principles of the EA approach seem quite stable. However, at the current stage of computer science, there is yet no clear understanding of what parallel, distributed or real-time algorithms are

*To appear in *Specification and Validation Methods*, Ed. E. Börger, Oxford University Press, 1994.

†Partially supported by ONR grant N00014-91-J-1861 and NSF grant CCR-92-04742.

in general. Thus, the definitions of parallel and distributed ealgebras given below are necessarily tentative. They provide a foundation for existing EA applications and reflect my anticipation of things to come. (Many existing applications, including those in this volume, were done before this guide have been completed; the terminology there may reflect earlier versions of the guide.)

We try to derive our definitions from first principles. Unfortunately some arbitrariness is inescapable and one has to balance the clarity and simplicity versus programming convenience and efficient execution. When one thinks mostly about applications, as we do, there is a tendency to prefer programming convenience and efficient execution. This is a dangerous trend which leads to an idiosyncratic programming language. For future reference we formulate the following principle:

The Pragmatic Occam's Razor Logic simplicity comes first; it may be sacrificed only in those cases where a slight logic complication is demonstrated to ease programming or improve execution efficiency in a substantial way.

The EA field is quickly expanding in depth and breadth. I hope that this guide lives up to its name and guides the developments in the near future.

Acknowledgment Egon Börger and Dean Rosenzweig generously shared with me their ideas and rich application experience. Discussions with Andreas Blass were indispensable in clarifying things. Numerous working walks with Jim Huggins through the woods of Ann Arbor were very helpful. Raghu Mani raised important implementation issues. Numerous ealgebraists commented on earlier drafts of the guide. I am very thankful to all of them. To an extent, this chapter is a result of a collective effort, though I am responsible for possible blunders.

A preliminary version of the guide has been tried out during the 1993 summer school on Specification and Validation Methods for Programming Languages and Systems on the beautiful island of Lipari in Italy. I use this opportunity to thank the organizers, Egon Börger and Alfredo Ferro, and all participants.

2 Static Algebras and Updates

2.1 Static Algebras: Motivation

In first-order logic, a structure is a nonempty set with operations and relations (called the basic operations and relations of the structure). That is how Tarski defined structures. He could have defined structures differently; there were a number of reasonable options. For our purposes here, a variant of Tarski's notion is more appropriate. Respecting tradition, we do not redefine structures. Rather, we modify the notion of structure and give the new notion a new name.

Structures without relations are called *algebras* in the branch of mathematics called universal algebra. Restrict attention to algebras with distinct nullary operations *true* and *false* and define basic relations as basic operations taking only the Boolean values *true* and *false*. Further restrict attention to algebras with the equality relation and the usual Boolean operations. (We will specify later the values of the Boolean operations outside their natural domains.) The resulting notion of algebra is our variant of the notion of structure with equality. It allows us to write quantifier-free formulas as terms.

Actually, we are interested in multi-sorted structures with partial operations. The sorts can be given by unary relations (they will be called universes and the whole underlying set of a structure will be called the superuniverse). To deal with partial functions, further restrict attention to algebras with a nullary operation *undef*, different from *true* and *false*, and interpret an operation f as undefined at a tuple $\bar{a}$ if $f(\bar{a}) = undef$. These algebras will be called *static algebras* or *states*. Their operations will be called functions.

In the following subsections, we start anew and define static algebras from scratch, establishing terminology on the way.

2.2 Vocabularies

A *vocabulary* (or *signature*) is a finite collection of function names, each of a fixed arity. Some function names may be marked as *relation names* or *static names*, or both. Every vocabulary contains the following static names: the equality sign, nullary function names *true*, *false*, *undef* and the names of the usual Boolean operations. The equality sign and *true*, *false* are marked as relation names. The Greek letter Υ is reserved to denote vocabularies.

Logic Names The particular function names listed above are *basic logic names*. There are precedents of logic names in mathematical logic, though usually they are called logical constants. For example, the equality sign is a logic name in first-order logic with equality. Usually, logic names are present in every vocabulary and their interpretations satisfy some *a priori* restrictions. Accordingly, we suppose that the basic logic names appear in every vocabulary, and thus there is no need to mention them when a particular vocabulary is described.

An additional logic name is introduced in §3. It does not necessarily appear in every vocabulary and it is not marked static. The latter is one reason why we do not use the term "logical constants".

2.3 Definition of Static Algebras

A *static algebra* or (for the sake of brevity) *state* S of vocabulary Υ is a nonempty set X, the *superuniverse* of S, together with interpretations of the function names in Υ on X. An r-ary function name is interpreted as a function from X^r to X, a *basic function* of S. The interpretation of an r-ary relation name is a function from X^r to $\{true, false\}$, a *basic relation* of S. The vocabulary Υ is called the *vocabulary* of S and denoted $\mathrm{Fun}(S)$.

The interpretations of the nullary logic names *true*, *false* and *undef* are distinct elements of X. The Boolean operations behave in the usual way on the Boolean values *true* and *false* and produce *undef* if at least one of the arguments is not Boolean. The equality sign is interpreted as the characteristic function of the identity relation on X. If $f(\bar{x})$ evaluates to *true* in S, we say that $f(\bar{x})$ holds in S; and if $f(\bar{x})$ evaluates to *false* in S, we say that $f(\bar{x})$ fails in S.

Formally speaking, basic functions are total. However, we view them as being partial and define the domain $\mathrm{Dom}(f)$ of an r-ary basic function f as the set of r-tuples $\bar{x}$ such that $f(\bar{x}) \neq undef$. Let us stress though that *undef* is an ordinary element of the superuniverse. Often, a basic function produces *undef* if at least one argument equals *undef*, but this is not required and there are exceptions (*e.g.* basic relations).

Universes A basic relation f may be viewed as the set of tuples where it evaluates to *true*. We may write $\bar{x} \in f$ instead of $f(\bar{x})$. If f is unary it can be viewed as a special *universe*. For example, we may have a universe Nodes and declare a binary relation Edge over the universe of Nodes; $\mathrm{Edge}(x, y)$ will hold only if both x and y belong to Nodes. Such universes allow us to view states as many-sorted structures. Sometimes we speak about universe names. These are unary relation names intended to be used as universes.

As a rule, *undef* is not included in universes. Coming back to our example, is it natural that Edge(*undef*,*undef*) equals *false* rather than *undef*? In a sense, yes. Think about Edge as a set of pairs of nodes. It is natural that the pair (*undef*,*undef*) does not belong there.

2.4 Terms

Terms are defined recursively, as in first-order logic:

- A variable is a term.
- If f is an r-ary function name and $t_1, \ldots, t_r$ are terms, then $f(t_1, \ldots, t_r)$ is a term.

As usual, *ground terms* are terms without variables. By analogy, other syntactical objects without variables will be called ground.

Atomic Boolean terms are terms of the form $f(\bar{t})$, where f is a relation name. *Boolean terms* are built from atomic Boolean terms by means of the Boolean operations.

Appropriate States and the Fun Notation In addition to terms, we will define various other syntactic objects, *e.g.*, update instructions and transition rules. We call a state S *appropriate* for a syntactic object s if $\mathrm{Fun}(S)$ includes the collection of function names that occur in s. By default (that is, unless explicitly defined differently), that collection will be denoted $\mathrm{Fun}(s)$.

In an appropriate state S, a ground term $t = f(t_1, \ldots, t_r)$ evaluates to an element $Val_S(t) = f(Val_S(t_1), \ldots, Val_S(t_r))$. If $\bar{t}$ is a tuple $(t_1, \ldots, t_r)$ of terms, define $Val_S(\bar{t}) = (Val_S(t_1), \ldots, Val_S(t_r))$.

An expression $t_1 = t_2$ may be a Boolean term or a metalanguage statement. Often it does not matter which it is. One can use two different equality signs or just try to be careful; we choose the second alternative.

2.5 Locations and Updates

As in first-order logic, the *reduct* of an Υ-state S to a smaller vocabulary Υ' is the Υ'-state S' obtained from S by "disinterpreting" function names in $\Upsilon - \Upsilon'$; S is an *expansion* of S' to Υ.

A *carrier* is a state whose vocabulary contains only static function names. The *carrier* $|S|$ *of a state* S is the reduct of S to the static part of $\mathrm{Fun}(S)$.

A *location over a carrier* C is a pair $\ell = (f, \bar{x})$, where f is a function name outside of $\mathrm{Fun}(C)$ and $\bar{x}$ is a tuple of elements of C whose length equals the arity of f; location ℓ is *relational* if f is a relation symbol. $\mathrm{Loc}_\Upsilon(C)$ is the collection of all locations over C with function names in Υ. An Υ-state S with carrier C will sometimes be viewed as a function from $\mathrm{Loc}_\Upsilon(C)$ to (the superuniverse of) C; *locations of* S are locations in $\mathrm{Loc}_\Upsilon(C)$.

If a state S is appropriate for a ground term $t_0 = f(\bar{t})$, then the *location* of t_0 in S is the location $(f, Val_S(\bar{t}))$.

An *update* of a state S is a pair $\alpha = (\ell, y)$, where ℓ is a location of S and $y \in |S|$; if ℓ is relational then y is Boolean. (More precisely, y belongs to the superuniverse of static algebra $|S|$; the looser language is common in logic.) The location ℓ is the *location* $\mathrm{Loc}(\alpha)$ of α, and y is the *value* $\mathrm{Val}(\alpha)$ of α. To *fire* α at S, put y into the location ℓ; that is, redefine S to map ℓ to y. The result is a new state S' such that $\mathrm{Fun}(S') = \mathrm{Fun}(S)$, $|S'| = |S|$, $S'(\ell) = y$ and $S'(\ell') = S(\ell')$ for every location ℓ' of S different from ℓ.

2.6 Update Sets and Families of Update Sets

An *update set* β over a state S is a set of updates of S. $\mathrm{Loc}(\beta) = \{\mathrm{Loc}(\alpha) : \alpha \in \beta\}$. For each $\ell \in \mathrm{Loc}(\beta)$, $\mathrm{Val}_\beta(\ell) = \{\mathrm{Val}(\alpha) : \alpha \in \beta \wedge \mathrm{Loc}(\alpha) = \ell\}$.

An update set β is *consistent* at the given state S if every $\mathrm{Val}_\beta(\ell)$ is a singleton set; otherwise β is inconsistent.

To fire a consistent β at the given state S, fire all its members simultaneously. The result is a new state S' with the same vocabulary and carrier as S. If $\ell \in \mathrm{Loc}(\beta)$ then $S'(\ell)$ is the only element of $\mathrm{Val}_\beta(\ell)$; otherwise $S'(\ell) = S(\ell)$. To fire an inconsistent update set β at the given state S, do nothing; the new state S' equals S.

Remark It is reasonable to require that the detection of inconsistency manifest itself in some way; for example, a nullary function *crash* automatically gets value *true*. To keep the EA logic clean and simple, we try to minimize the number of things done automatically, and thus we leave necessary manifestations of inconsistency to the programmer. This is one application of the pragmatic Occam's razor of §1; substantial programming convenience has not been demonstrated yet.

To fire a family γ of update sets over S, nondeterministically choose some update set $\beta \in \gamma$ and fire it at S. If $\gamma = \emptyset$, do nothing. Intentionally, the empty family of update sets means inconsistency.

2.7 Conservative Determinism vs. Local Nondeterminism

The mode of dealing with inconsistent update sets described above can be called conservative determinism. The mode of dealing with inconsistent update sets in the tutorial was different: Fire all updates simultaneously; in case of conflict at any location ℓ, choose the new value for ℓ nondeterministically among all candidate values. It could be called local nondeterminism.

With the exception of this change in the treatment of inconsistent update sets, this guide is compatible with the tutorial. The change is not as big as it may seem because people are usually interested in deterministic programs. As far as we know, no existing EA application is affected. The local nondeterminism has not been exploited. The conservative determinism is simpler, and a more manageable form of nondeterminism will be introduced in §4.

3 Sequential Evolving Algebras

Basic transition rules are defined in subsection 3.1. Subsection 3.2 deals with the problem of extending universes. The reader may skip 3.2 and go directly to subsection 3.3 on programs and runs.

3.1 Basic Transition Rules

In this subsection, terms are ground.

3.1.1 Update Instructions

An *update instruction* R is an expression

$$f(\bar{t}\,) := t_0$$

where f is a non-static function name (the *subject* of the instruction), $\bar{t}$ is a tuple of terms whose length equals the arity of f, and t_0 is another term; if f is a relation name then t_0 must be a Boolean term. (Update instructions are called local function updates in the tutorial.)

Semantics To execute R at an appropriate state S, fire the update $\alpha = (\ell, y)$ at S, where $\ell = (f, Val_S(\bar{t}\,))$ and $y = Val_S(t_0)$. For future reference define $\text{Updates}(R, S) = \{\alpha\}$.

3.1.2 Two Rule Constructors

Basic rules are constructed recursively from update instructions by means of two rule constructors: the sequence constructor and the conditional constructor. Semantics is defined by means of update sets. For each rule R and every state S appropriate for R, we define an update set $\text{Updates}(R, S)$ over S. To fire R at S, fire $\text{Updates}(R, S)$.

The Sequence Constructor A sequence of rules is a rule.

Semantics If R is a sequence of rules $R_1, \ldots, R_k$ then

$$\text{Updates}(R, S) = \text{Updates}(R_1, S) \cup \cdots \cup \text{Updates}(R_k, S).$$

In other words, to fire a sequence of rules, fire all of them simultaneously. Notice that $\text{Updates}(R, S)$ is inconsistent if any R_i is so.

Remark The term "sequence" may be misleading here. We are not executing first R_1, then R_2, then R_3, *etc.*. A better term is "block". (This remark is written at the proofreading stage.)

The Conditional Constructor If k is a natural number, $g_0, \ldots, g_k$ are Boolean terms and $R_0, \ldots, R_k$ are rules, then the following expression is a rule:

```
if g0 then R0
elseif g1 then R1
   ⋮
elseif gk then Rk
endif
```

If the guard g_k is the nullary function *true*, then the last elseif clause may be replaced by "else R_k". For brevity we will say that the conditional rule R above is the conditional rule with clauses $(g_0, R_0), \ldots, (g_k, R_k)$.

Semantics Updates(R, S) = Updates(R_i, S) if g_i holds in S but every g_j with $j < i$ fails in S. Updates$(R, S) = \emptyset$ if every g_i fails in S.

3.1.3 Guarded Multi-updates

A *multi-update instruction* is a sequence of update instructions. A *guarded update instruction* (respectively, *guarded multi-update instruction*) is a rule of the form

```
if g then R endif
```

where R is an update (respectively, a multi-update) instruction.

Lemma 3.1 *For every rule R, there is a sequence R' of guarded updates such that $Fun(R') = Fun(R)$ and $Updates(R', S) = Updates(R, S)$ for all appropriate states S.*

For example, the rule

```
if FirstChild(c)≠ undef then c:=FirstChild(c)
elseif NextSib(c)≠ undef then c:=NextSib(c)
elseif Parent(c)≠ undef then c:=Parent(c)
endif
```

converts to the following sequence of guarded updates:

```
if FirstChild(c)≠ undef then c:=FirstChild(c) endif
if FirstChild(c)=undef and NextSib(c)≠ undef then
   c:=NextSib(c) endif
if FirstChild(c)=undef and NextSib(c)=undef
   and Parent(c)≠ undef then c:=Parent(c) endif
```

The Lemma suggests a simpler definition of rules. The reason for choosing the recursive definition is pragmatic. It is too tedious to write rules as sequences of guarded updates. It is feasible to write them as sequences of guarded multi-updates but it is more convenient and practical to use elseif clauses and nest conditionals. The pragmatic Occam's razor does not cut as much as the original Occam's razor would.

Remark This is another proofreading time remark. The new version of the EA interpreter permits the use of two additional rule constructors. One is the case constructor, like that in Pascal, which may make the execution substantially more efficient. Of course, the same set of updates is generated by a case command and its case-free equivalent; the difference is in how fast this set is generated. For example, consider a sequence of rules of the form "if $t = i$ then R_i endif" where i ranges from 1 to a relatively large n. This example is extreme, because the the set $\{1, \ldots, n\}$ of alternatives is so easy to deal with; but it is not unusual to have similar long sequences of rules. In addition, the case construct makes it easier to program a sequential execution of a sequence of rules, which is sometimes desirable. The other rule constructor is "let x=t in R", which prevents re-evaluations of term t in R and which has been used informally. The let constructor was advocated by Raghu Mani who is working on the new EA interpreter.

3.2 Importing New Elements

The basic rules suffice for many purposes (*e.g.*, for describing the C programming language [GH]), but they do not suffice to model all sequential algorithms. A sequential algorithm may add a new node to a graph or create a new message. We need rules that allow us to create new nodes, new messages, *etc.*, and such rules are introduced in this subsection. However, we do not create new elements; instead, we use a special universe Reserve from which the new elements come.

In this section we use individual variables, but only in a limited way. (Variables are used more extensively in §5.) Roughly speaking, only bound variables are used; free variables appear only in contexts where some values have been assigned to them.

3.2.1 Reserve

In addition to basic logic names, we introduce a new logic name: a universe name Reserve. It is not static, and we do not require that it belong to the vocabulary of every static algebra. If the vocabulary of state S contains Reserve, then the set $\{x : S \models x \in \text{Reserve}\}$ is the *reserve* of S. Intuitively the reserve is a naked set.

Reserve Proviso Every state satisfies the following conditions:

- Every basic relation, with the exception of equality and Reserve, evaluates to *false* if at least one of its arguments belongs to the reserve.
- Every other basic function evaluates to *undef* if at least one of its arguments belongs to the reserve.
- No basic function outputs an element of the reserve.

It follows that every permutation of the reserve is an automorphism of the state.

3.2.2 Transition Rules: Syntax

Generalize the definitions of terms and update instructions in 3.1 as follows:

- allow terms to have variables, and
- forbid mentioning Reserve.

Variables are often treated as auxiliary nullary function names below but *a variable cannot be the subject of an update instruction.* The reason for forbidding to mention Reserve in terms and update instructions is discussed below.

Rules are constructed from update instructions by means of three rule constructors: the sequence constructor, the conditional constructor and the import constructor.

The Import Constructor If v is a variable and R_0 is a rule, then the following expression is a rule with *main existential variable* v and *body* R_0:

```
import v
   R0
endimport
```

In the usual and obvious way define which occurrences of variables are free and which are bound. Call a rule *perspicuous* if no variable has both bound and free occurrences, and no bound variable is declared more than once. (The latter means here that different occurrences of the import command have different main existential variables.)

Let Free(R) be the set of free variables of a rule R. In other words, Free(R) is the set of variables v such that v occurs freely in rule R. Define Bound(R) similarly. If R is an import rule with main existential variable v and body R_0, we have:

$$\text{Free}(R) = \text{Free}(R_0) - \{v\}, \quad \text{and} \quad \text{Bound}(R) = \text{Bound}(R_0) \cup \{v\}.$$

3.2.3 Auxiliary Vocabularies

The names of variables are different from function names of course, but it is convenient to treat free variables of rules as auxiliary nullary functions (which cannot be subjects of update instructions). An *auxiliary vocabulary* has the form $\Upsilon \cup V$, where Υ is a genuine vocabulary and V is a finite set of variables.

If S is a state of an auxiliary vocabulary $\Upsilon' = \Upsilon \cup V$, then $\mathrm{Fun}(S) = \Upsilon'$. S is *appropriate* for a rule R if Υ contains all function names of R and V contains all free variables of R. R is *S-perspicuous* if it is perspicuous and its bound variables do not occur in V.

3.2.4 Transition Rules: Semantics

An import commands chooses an element of the reserve and removes it from the reserve. To clarify our intentions, we note that the non-perspicuous rule

```
import v
   Parent(v):=CurrentNode
endimport
import v
   Parent(v):=CurrentNode
endimport
```

creates *two* children of CurrentNode. In general, different choices from the reserve produce different elements.

For each rule R and every state S appropriate for R, we define an update set $\mathrm{Updates}(R,S)$ over S; to fire R at S, fire $\mathrm{Updates}(R,S)$.

First, we consider the case of when R is S-perspicuous. Fix an injective map ξ from $\mathrm{Bound}(R)$ to the reserve of the given S. (The injectivity means that ξ assigns different elements to different bound variables.) By induction on subrule R' of R we define sets $\mathrm{Updates}(R',S',\xi)$ where S' is an expansion of S appropriate for R' and such that R' is S'-perspicuous. (Recall that S' is an expansion of S if and only if the reduct of S' to $\mathrm{Fun}(S)$ equals S.) Let $\Upsilon' = \mathrm{Fun}(S')$.

The cases of update instructions, sequence rules and conditional rules are treated as above. (Variables in Υ' are treated as nullary functions.) Suppose that R' is an import rule with main existential variable v and body R_0. Let $a = \xi(v)$ and S'_a be the expansion of S' to the auxiliary vocabulary $\Upsilon' \cup \{v\}$ where v is interpreted as a. Recall that variables are not subjects of update instructions. Thus $\mathrm{Updates}(R_0, S'_a, \xi)$ is an update set over S'. Set

$$\mathrm{Updates}(R',S',\xi) = \{((\mathrm{Reserve}, a), \mathit{false})\} \cup \mathrm{Updates}(R_0, S'_a, \xi).$$

Finally $\mathrm{Updates}(R,S) = \mathrm{Updates}(R,S,\xi)$. Of course, $\mathrm{Updates}(R,S)$ is not defined uniquely, because it depends on ξ. It is easy to see, however, the resulting state is unique up to isomorphism.

Second, we stipulate that an arbitrary rule R is equivalent, over the given appropriate state S, to an S-perspicuous rule R' obtained from R by renaming the bound variables. (The desired R' can be obtained by iterating the following transformation: Select an innermost import subrule R_1 whose main existential variable v occurs in the rest of the rule or in $\mathrm{Fun}(S)$, and replace v with a fresh variable in R_1.) The stipulation means the following: To fire R at S, fire R' at S.

Discarding Elements from Universes Finally, we explain the reason for forbidding to mention Reserve explicitly in our rules. Terms $\mathrm{Reserve}(t)$ always evaluate to *false*, so evaluating $\mathrm{Reserve}(t)$ or setting it to *false* is useless. But why not to allow putting the value *true* into Reserve locations. Elements can be discarded from universes, of course; to discard an element (represented by a term) t from a universe U, use the instruction $U(t) := \mathit{false}$. Isn't the reserve a natural place for unwanted elements? Yes, it is. Notice, however, that moving an element into the reserve may necessitate numerous changes of basic functions in order to ensure that the Reserve proviso remains valid. Would such a move contradict the sequential character of our rules? Not necessarily. We could just mark discarded elements as reserve elements, but then it might be necessary to augment rules with numerous guards $\mathrm{Reserve}(t) = \mathit{false}$, which would be too tedious. It is preferable to leave the discarded elements alone. This pragmatic argument was put forward originally by Egon Börger.

But shouldn't the computational resources of the ealgebra simulating an algorithm A closely reflect the computational resources of A? Yes, but it is important to separate the following concerns: the logic of A and the relevant resources of A. Concentrating on the logic of A may allow one to come up with simpler rules for the simulating ealgebra. And if one needs to track the resources of A, a separate bookkeeping may be set up. This separation of concerns allows us, for example, to use infinite universes. And caring about only particular elements and universes, rather than the whole superuniverse, makes combining ealgebras easier.

3.2.5 Importing Several Elements at a Time

Let v_1, v_2 be distinct variables. Abbreviate

```
import v1
   import v2
      R0
   endimport
endimport
```

to

```
import v1, v2
   R0
endimport
```

In a similar way, define abbreviations

```
import v1, ..., vk
   R0
endimport
```

Abbreviate

```
import v1, ..., vk
   U(v1) := true
   ⋮
   U(vk) := true
   R0
endimport
```

to

```
extend U with v1, ..., vk
   R0
endextend
```

Later (in 5.4) we'll see how to import a number of elements that is not bounded *a priori* by any constant. Here is an example of the extend rule:

```
extend Nodes with v1, v2
   FistChild(CurrentNode) := v1
   SecondChild(CurrentNode) := v2
   NextSib(v1) := v2
endextend
```

3.3 Programs and Runs

3.3.1 Programs and Pure Runs

A *program* P is a rule without free variables. A *basic program* is a basic rule without free variables. In applications, a program is usually a sequence of rules referred to as rules of the program. To fire P at an appropriate state S, fire Updates(P, S) at S.

A *pure run* of P is a sequence $\langle S_n : n < \kappa \rangle$ of states of vocabulary Fun(P) such that each S_{n+1} is obtained from S_n by firing P at S_n. Here and henceforth κ is a positive integer or the first infinite ordinal. In the latter case, $\{n : n < \kappa\}$ is the set of all natural numbers.

The adjective "pure" reflects the fact that the run is not affected by the environment.

3.3.2 External Functions

In general runs may be affected by the environment. Suppose that the environment manifests itself via some basic functions $e_1, \ldots, e_k$, called *external functions*. A typical external function is the input provided by the user.

Think about an external function as a (dynamic) oracle. The ealgebra provides the arguments and the oracle gives the result. The oracle need not be consistent and may give different results for the same argument at different times. The seeming inconsistency may be quite natural. For example, the argument may specify an input channel. The next time around, another input can come via the same channel.

However, the oracle should be consistent during the execution of any one step of the program. In an implementation, this may be achieved by not reiterating the same question during a one-step execution. Ask the question once and, if necessary, save the result and reuse it.

The computation steps of a program are supposed to be atomic at an appropriate level of abstraction. A computation step is hardly atomic if during that step the ealgebra queries an oracle and then, depending on the result, submits another query to the same or a different oracle. Thus it seems reasonable to forbid nesting of external functions. Indeed, the need to nest external functions has not arisen in applications so far. But we withhold final judgement and wait for more experimentation.

Call non-external basic functions *internal*. If S is an appropriate state for a program P, let S^- be the reduct of S to the internal vocabulary.

Runs A *run* of a program P is a sequence $\langle S_n : n < \kappa \rangle$ of states where:

- every nonfinal S_n is an appropriate state for P and the final state (if any) is a state of the internal vocabulary of P, and
- every S^-_{n+1} is obtained from S_n by firing P at S_n.

Internal and External Locations It may happen that the environment controls only a part of a function e_i and the remaining part of e_i is governed internally. In such a case it is natural to speak about internal and external locations rather than internal and external functions. See an example in [BGR, 3.1]. The generalization to that case is relatively straightforward.

Irrelevant Values of External Functions In order to fire a given program at a given state, we may not need to know all about the state. Only some values of external functions may be needed for firing. We may not care about or even know the values of external functions which are not needed for the execution. Some of those values may even be ill-defined. There is also an issue of influencing the environment by requiring an extra value, *e.g.*, by requiring a user-provided datum.

It is natural to set all irrelevant values of external functions to *undef*. However, caution should be exercised in the distributed situation (see §6) where other agents may have different views of those values.

Sometimes it may be simpler to use partial states. A partial Υ-state S with carrier C can be defined as a partial function from $\mathrm{Loc}_\Upsilon(C)$ to C. See examples in [BGR, GM]. For simplicity, we will not use partial states here.

4 Nondeterministic Sequential Ealgebras and Some Other Simple Extensions of the Basic Model

Describing algorithms on higher abstraction levels, one often comes across the phenomenon of nondeterminism. Nevertheless, the built-in nondeterminism of ealgebras has been rarely used. It is often more appropriate to use external functions to reflect nondeterministic behavior. (In the distributed case, nondeterminism may be often eliminated by introducing additional agents.) Consider for instance the assignment statement of the C programming language. Should one evaluate the left side or the right side first? According to the ANSI standard (ANSI is the American National Standards Institute), the choice of the evaluation order is implementation-dependent. Moreover, an implementation does not have to be consistent; the evaluation order may change when the same assignment statement is executed next time around (say, in a loop). This is an obvious case of nondeterminism and first we, the authors of [GH], were tempted to use a nondeterministic rule to reflect the nondeterminism. But then we realized that C is perfectly deterministic. It is just that execution may depend on information provided by implementation. Thus it is more faithful to the standard (and more convenient) to use an external function that decides the evaluation order.

Still, nondeterministic commands may be desired and we provide such commands in this section. For example, it may be convenient to formalize the environment in a distributed situation, so that an external function of one agent is nondeterministically computed by another agent.

For simplicity, we ignore the import constructor in this section. It is easy to extend the language of this section with the import constructor. Moreover, the choice constructor defined below and the import constructor can be combined into one constructor.

4.1 Basic Evolving Algebras with Choice

4.1.1 Syntax

Transition rules are constructed as in 3.2, except that instead of the import constructor, we use the Choose (or Choice) Constructor:

Choose Constructor If U is a universe name different from Reserve, v is a variable and R_0 is a rule then the following expression is a rule with *main existential variable* v that ranges over U and *body* R_0:

`choose` v `in` U
  R_0
`endchoose`

This is the basic version of the choice constructor; a stronger version is defined in 4.2.2. Perspicuity is defined as 3.2.

4.1.2 Semantics

For each rule R and each state S appropriate for R, we define a family $\gamma = \text{NUpdates}(R, S)$ of update sets over S. To fire R at S, choose any $\beta \in \gamma$ and fire β at S.

We stipulate that an arbitrary rule R is equivalent, over the given S, to an S-perspicuous rule R' obtained from R by renaming the bound variables. The equivalence means here that $\text{NUpdates}(R, S) = \text{NUpdates}(R', S)$. It remains to define $\gamma = \text{NUpdates}(R, S)$ when S is S-perspicuous.

Global Choice Semantics Semantics is defined as in 3.2.4. On one hand, things are simpler this time around because there is no correlation among individual choices. On the other hand, there is a complication related to attempts to choose an element of the empty set. Such attempt cannot succeed and the execution should be aborted. To deal with this complication, we extend the collection of updates of any state by an ideal element $\perp$ that symbolizes inconsistency. If an update set β contains $\perp$ then firing β does not change the state; we call such β contradictory.

Suppose that a state S is appropriate for a rule R and R is S-perspicuous. Let V be the collection of bound variables of R such that the range of v is not empty in state S. Fix a function ξ on V such that, for each $v \in V$, $\xi(v)$ belongs to the range of v in S. By induction on subrule R' of R define $\text{Updates}(R', S', \xi)$ where S' is an expansion of S appropriate for R' and R' is S'-perspicuous.

The cases of update instructions, sequence rules and conditional rule are treated as above. Notice that if R' is a sequence of rules R_i and some $\text{Updates}(R_i, S', \xi)$ is contradictory then $\text{Updates}(R', S', \xi)$ is so.

Suppose that R' is a choose rule with main existential variable v and body R_0. If the range of v is empty then $\text{Updates}(R', S', \xi) = \perp$. Otherwise let $a = \xi(v)$ and S'_a be the expansion of S' to the auxiliary vocabulary $\Upsilon' \cup \{v\}$ where v is interpreted as a. Set $\text{Updates}(R', S', \xi) = \text{Updates}(R_0, S'_a, \xi)$.

Finally, $\text{NUpdates}(R, S)$ is the set of $\text{Updates}(R, S, \xi)$ where ξ takes all possible values.

Semantics without Global Choice The global choice semantics is straightforward. However, contrary to the situation 3.2.4, there is no correlation among individual choices this time around, and thus there is no real need for a global choice function ξ. It may be more elegant to define $\gamma = \text{NUpdates}(R, S)$ directly by induction on R. We suppose again that S is appropriate to R and R is S-conspicuous.

If R is an update instruction then $\gamma = \{\text{Updates}(R, S)\}$. If R is a sequence of rules $R_1, \ldots, R_k$, then

$$\gamma = \{\beta_1 \cup \cdots \cup \beta_k \; : \; \text{each } \beta_i \in \text{NUpdates}(R_i, S)\}.$$

Notice that γ is empty if some so is $\text{NUpdates}(R_i, S)$.

If R is a conditional rule with clauses $(g_0, R_0), \ldots, (g_k, R_k)$, we have two cases as usual; if all $k+1$ guards fail in S then $\gamma = \{\emptyset\}$, and if g_i is the first guard that holds in S then $\gamma = \text{NUpdates}(R_i, S)$. (It would be a mistake to replace $\{\emptyset\}$ with $\emptyset$ above. If $\text{NUpdates}(R_1, S) = \emptyset$ then $\text{NUpdates}((R_1, R_2), S) = \emptyset$ for every rule R_2, which is not desired.)

Finally, suppose that R is a choose rule with universe name U, main existential variable v and body R_0. For each $a \in U$, let S_a be the expansion of S of the auxiliary vocabulary $\text{Fun}(S) \cup \{v\}$ where v is interpreted as a. Then

$$\gamma = \bigcup \{\text{NUpdates}(R_0, S_a) \; : \; a \in U\}.$$

Notice that γ is empty if U is empty.

It is easy to check that if R contains no choice subrules then $\text{NUpdates}(R, S) = \{\text{Updates}(R, S)\}$.

Remark In the second approach, $\perp$ is not used. Its role is played by the empty family of update sets. This gives us an idea to eliminate the use of $\perp$ in the first approach: replace $\text{Updates}(R', S', \xi)$ with the singleton family $\{\text{Updates}(R', S', \xi)\}$ and replace $\perp$ with the empty family.

Runs The definition of runs in §3 remains in force.

4.1.3 Abbreviations

Let v_1, v_2 be distinct variables. Abbreviate

```
choose v1 in U
   choose v2 in U
      R0
   endchoose
endchoose
```

to

```
choose v1, v2 in U
   R0
endchoose
```

In a similar way define abbreviation

```
choose v1, ..., vk in U
   R0
endchoose
```

4.2 Some Other Simple Extensions of the Basic Model

We consider three extensions, which are simple in the sense that it is easy to define them. The third extension has not been used; it is just a trial balloon.

4.2.1 First-order Guards

In §3, guards were Boolean terms. Now we introduce a separate syntactic category of guards. Intuitively, guards are first-order formulas with bound variables. It is intended that bound variables range over finite domains, though exceptions are possible. Here is a recursive definition:

- If f is an r-ary relation name and $t_1, \ldots, t_r$ are terms, then $f(t_1, \ldots, t_r)$ is a guard.

- Any Boolean combination of guards is a guard.
- If g is a guard and U a universe name, then $(\exists v \in U)g$ and $(\forall v \in U)g$ are guards.

Call a guard closed if it has no free variables. Extend the definition of basic ealgebras by replacing the condition "$g_1, \ldots, g_k$ are Boolean terms" with the condition "$g_1, \ldots, g_k$ are closed guards" in the definition of the conditional rule constructor.

Semantics The definition of the value of a closed guard at an appropriate state mirrors the truth definition of formulas in first-order logic. The semantics of rules is given exactly as in 3.1.

Remark One can go further in this direction and use quantification inside other terms. To formalize this idea, the notion of terms can be redefined as follows:

- A variable v is a term.
- If f is an r-ary function name and $t_1, \ldots, t_r$ are terms, then $f(t_1, \ldots, t_r)$ is a term. The new term is Boolean if f is a relation name.
- Boolean terms are closed under the Boolean operations and quantification, and every Boolean term is a term.

4.2.2 Qualified Choose Construct

Restricting the choice by a Boolean term gives a much more powerful version of the choose constructor.

Qualified Choose Constructor If U is a universe name different from Reserve, v is a variable, $g(v)$ is a Boolean term and R_0 is a rule, then the following expression is a rule with *main existential variable* v that ranges over U and body R_0:

choose v **in** U **satisfying** $g(v)$
  R_0
endchoose

Replacing the choose constructor with the qualified choose constructor requires only a small and obvious change in the semantical definition of 4.1.2. We restrict attention to the global choice approach. Consider the case in the inductive definition of Updates(R', S', ξ) where R' is a choose rule and the range U of the main existential variable v of R in S' is not empty. If $g(\xi(v))$ fails in S, set Updates$(R', S', \xi) = \perp$.

It is easy to construct a rule to choose several elements $v_1, \ldots, v_k$ subject to a condition $g(v_1, \ldots, v_k)$.

The qualified choose constructor may be too powerful. The decision problem whether there is any tuple $(v_1, \ldots, v_k)$ in the universe U satisfying the condition g may be hard. If U is the set of natural numbers and g a polynomial, the decision problem may even be undecidable [Mt]. But the logical clarity of the constructor is attractive. It may be used in particular to reflect environmental forces that are not necessarily algorithmic.

4.2.3 Duplication

The powerful extension of basic ealgebras considered in this subsection is logically clear but untried and computationally expensive. It does not hurt to explore it though.

Call elements a and a' of a state S indistinguishable as arguments for a basic r-ary function f if $f(b_1, \ldots, b_r) = f(c_1, \ldots, c_r)$ for all r-tuples $b_1, \ldots, b_r$ and $c_1, \ldots, c_r$ such that either $b_i = c_i$ or $\{b_i, c_i\} = \{a, a'\}$. Call a, a' indistinguishable as arguments if they are indistinguishable as arguments for any basic function with the exception of equality. Now we are ready to introduce the duplicate constructor:

duplicate t **as** v
  R_0
endduplicate

Semantics To execute, calculate $a = Val_S(t)$, get some a' from the reserve and redefine basic functions on tuples involving a' in such a way that a and a' become indistinguishable as arguments. Then execute R_0 with v equal a'.

Duplication can be seen as a powerful inheritance mechanism. It is easy to see that the extend construct is not powerful enough to replace duplication.

5 Parallelism: Evolving Algebras with Variables

What does it mean that an algorithm is sequential? This usually means that the algorithm has the following two features. First, time is sequential. The algorithm proceeds from some initial state S_0 to a state S_1, then to a state S_2, *etc.*, and the steps are atomic. Second, only a bounded amount of work is done at each step. In principle, a single agent is able to move the algorithm from S_0 to S_1, then to S_2, *etc.*.

In this section, we are interested in one-agent algorithms where the agent may perform a substantial amount of work at one step. We use variables to formalize such algorithms. It is intended that non-Reserve variables range over finite (better yet, feasible) domains, though exceptions are possible.

We do not assume any particular sequential order of executing one step of the algorithm. It is possible that this work involves plenty of parallelism and is implemented by a number of auxiliary agents. But on the natural level of abstraction of the given algorithm, those auxiliary agents are invisible, and in principle a single agent may execute the algorithm.

5.1 Variables

In preceding sections, we dealt with implicit variables declarations by means import commands, bounded quantifiers, etc. In this section, we introduce explicit variable declarations.

An *explicit atomic variable declaration* is an expression "Var v ranges over U", where v is a variable and U a universe name. The universe U is the range (or type) of the variable v. A *explicit variable declaration* D is a sequence of explicit atomic variable declarations, and Var(D) is the collection of variables in D. For brevity, the adjective explicit is often omitted.

Intuitively, D is a set of explicit atomic declarations, but we do not forbid re-declarations of the same variable. The range of a variable $v \in \text{Var}(D)$ is the range in the last declaration of v in D. In other words, later declarations of a variable override the earlier ones. One may use more concise explicit variable declarations, like "Var $v_1, \ldots, v_k$ range over U".

A variable declaration D *covers* a syntactic object s if Var(D) contains all free (that is undeclared) variables of s.

As in 3.2.3, we use auxiliary vocabularies of the form $\Upsilon \cup V$, where Υ is a genuine vocabulary, V a finite set of variables and each $v \in V$ is treated as a nullary function, except it cannot be the subject of an update instruction. We say that a state S of an auxiliary vocabulary is *appropriate* for a syntactical object s if all function names and all free variables of s occur in Fun(S).

5.2 Terms and Guards

Terms and Boolean terms are defined in §3. Guards are defined in 4.2.1. The free variables of terms and guards are defined inductively, as in first-order logic. Notice that a bounded quantifier implicitly contains an atomic declaration.

As usual, every guard g is equivalent to a guard g' where no variable is both bound and free and where different quantifier occurrences bind different variables. To reduce g to g', iterate the following transformation: Select an innermost quantifier q whose variable v occurs outside the scope of q and then replace v with a fresh variable in the scope of q.

5.3 A Parallel Version of the Basic EA Model

5.3.1 Syntax

Update instructions and basic rules are defined as in 3.1, except that terms may have free variables, and guards are defined as above. In addition, we have the following third rule constructor.

The Declaration Constructor An atomic variable declaration followed by a rule is a rule.

By an obvious induction on rules, define which occurrences of variables are free (or undeclared) and which are bound. Suppose that D is a variable declaration, R is a rule, and S is a state of an auxiliary vocabulary. R is (D, S)-*perspicuous* if it satisfies the following conditions:

- no variable is declared (explicitly or implicitly) more than once in R, and
- $\text{Bound}(R)$ is disjoint from $\text{Free}(R) \cup \text{Var}(D) \cup \text{Fun}(S)$.

Programs A *program* is a rule without any undeclared variables.

5.3.2 Semantics of Rules

By induction on R, we define the update set $\beta = \text{Updates}(D, R, S)$ generated by a rule R at an appropriate state S under a declaration D that covers R. To fire R at S under D, fire β.

We stipulate that an arbitrary rule R is equivalent, for given D and S, to a (D, S)-perspicuous rule R' obtained from R by renaming the bound variables. The equivalence means that $\text{Updates}(D, R, S) = \text{Updates}(D, R', S)$.

It remains to define $\beta = \text{Updates}(D, R, S)$ in the case when R is (D, S)-perspicuous.

If D is not empty, then β is the union of $\text{Updates}(\emptyset, R, S')$, where S' ranges over expansions of S such that $\text{Fun}(S') = \text{Fun}(S) \cup \text{Var}(D)$ and S' is consistent with D (so that the values of D variables are within their ranges in S'). Notice that $\beta = \emptyset$ if the range of any D variable is empty.

Suppose $D = \emptyset$. If R is an update instruction then $\beta = \text{Updates}(R, S)$. If R is a sequence of rules $R_1, \ldots, R_k$, then β is the union of the update sets $\text{Updates}(\emptyset, R_i, S)$. Suppose that R is the conditional rule with clauses $(g_0, R_0), \ldots, (g_k, R_k)$. Since R is covered by the empty declaration, the guards g_i have no free variables. We have two cases as usual. If all guards g_i fail in S, then β is empty, and if g_i is the first guard that holds in S then $\beta = \text{NUpdates}(\emptyset, R_i, S)$. Finally, if R is a declaration rule with declaration d and body R' then $\beta = \text{Updates}(d, R', S)$.

Remark Suppose that $D = \emptyset$ and R is a sequence of a declaration-free rule R_1 and a declaration rule R_2 with atomic declaration "Var v ranges over U" followed by a declaration-free body R_2'. Further suppose that U is empty in a state S appropriate for R and thus $\text{Updates}(D, R_2, S) = \emptyset$. Then $\text{Updates}(D, R, S)$ equals $\text{Updates}(D, R_1, S)$ which may be not empty. Contrary to the situation in 4.1.2, the empty range does not give inconsistency here. One cannot choose an element from the empty set, but one can execute a $R_2'(v)$ for every v in the empty set: just do not execute anything.

5.4 Importing Elements

The recursive definition of rules in 5.3 can be extended by import commands and/or (qualified) choice commands. The adjustment of the semantic definition is straightforward. For the sake of definiteness, consider the extension by means of the import constructor. The most important novelty, in comparison to 3.2.4, is that reserve elements have to be chosen for all combinations of the values of explicitly declared variables u such that the scope of the declaration of u properly includes the given import or choose subrule. For example, the rule

```
Var u ranges over U
import v
   Parent(v):=u
endimport
```

creates a new child for every element of U, and of course all these new children are different.

To reflect the novelty we redefine the domain of the global choice function. Suppose that D is a variable declaration, R is a rule covered by D, S is a state of an auxiliary vocabulary appropriate for R, and R is (D,S)-perspicuous. For every bound variable v of R, list all explicitly declared variables u such that either u occurs in D or u occurs in R and the scope of the declaration of u properly includes the scope of the declaration of v: $u_1,\ldots,u_l$. (The adverb properly is there to exclude v from the list.) Let $U_1,\ldots,U_l$ be the ranges of $u_1,\ldots,u_l$ in S respectively, and $\bar{U}_v$ be the Cartesian product $U_1 \times \cdots \times U_l$. The desired global function ξ assigns different reserve elements to every pair $(v,\bar{a})$ where $v \in \mathrm{Bound}(R)$ and $\bar{a} \in \bar{U}_v$.

Here is a variant of the example from 3.2.5:

```
Var u ranges over U
extend Nodes with v1, v2
   if Leaf(u) then
      FirstChild(u) := v1
      SecondChild(u) := v2
      NextSib(v1) := v2
   endif endextend
```

Remark Should one provide means to say explicitly that the main existential variable of a given choose rule depends only on such and such of the free variables of the rule? Maybe. But the need for such means has not been demonstrated yet.

5.5 Runs

Runs are defined as above.

6 Distributed Evolving Algebras

In this section we consider multi-agent computations. We do not suppose that agents are deterministic or do only a bounded amount of work at each step. The program of an agent may be any program described above.

Agents may share functions, and it is convenient [GR] to assume that all states of all agents share the same carrier; see the end of 3.2.4 in this connection.

6.1 The Self Function

There is an interesting problem of self identification. It can be illustrated on the example of the following simple version of Dijkstra's dining philosophers protocol (which may deadlock). There are n philosophers, marked with numbers modulo n, each equipped with a fork. A philosopher i may think (which requires no forks) or eat using his/her fork and the fork of philosopher $i+1$. A fork cannot be used by two philosophers at the same time.

Using functions

$$\mathrm{Fork}_i = \begin{cases} \text{up} & \text{if the fork of philosopher } i \text{ is used,} \\ \text{down} & \text{otherwise,} \end{cases}$$

we can write a separate program for each philosopher i. Intuitively, however, all philosophers use the same program in the protocol.

To solve such problems, we suppose that each agent a is represented by an element of the common carrier. For simplicity, we will not distinguish between an agent and the element that represents the agent. Further, we use a special nullary function Self, interpreted differently by different agents. An agent a interprets Self as a. Thus function Self allows an agent to identify itself among other agents. Self is a logic name and cannot be the subject of an update instruction. To make rules sound a little better for humans, we use some capitalized pronouns, *e.g.* Me, as aliases for Self. Viewing agents as elements of the carrier is useful for other purposes as well. For example, it allows us to model the creation of new agents.

We return to the dining philosophers protocol. Here is a possible program (courtesy of Jim Huggins):

```
if Mode(Me)=think and Fork(Me)=Fork(Me+1)=down then
   Fork(Me):=up, Fork(Me+1):=up, Mode(Me):=eat
elseif Mode(Me)=eat then
   Fork(Me):=down, Fork(Me+1):=down, Mode(Me):=think
endif
```

It may be convenient to suppress the argument Self. For example, terms Mode(Me), Fork(Me) and Fork(Me+1) may be treated as nullary functions and abbreviated, *e.g.*, as mode, lfork and rfork, so that the rfork function of philosopher i is the lfork function of philosopher $i+1$ and mode is a private function.

6.2 Basic Definition of Distributed Ealgebras

A *distributed ealgebra* $\mathcal{A}$ consists of the following:

- A finite indexed set of single-agent programs π_ν, called *modules*. The *module names* ν are static nullary function names.
- A vocabulary $\Upsilon = \mathrm{Fun}(\mathcal{A})$ which includes each $\mathrm{Fun}(\pi_\nu) - \{\mathrm{Self}\}$ but does not contain Self. In addition, Υ contains a unary function name Mod.
- A collection of Υ-states, called *initial states* of $\mathcal{A}$, satisfying the following conditions:
  - Different module names are interpreted as different elements.
  - There are only finitely many elements a such that, for some module name ν, $\mathrm{Mod}(a) = \nu$.

A state S of vocabulary $\mathrm{Fun}(\mathcal{A})$ is a state of $\mathcal{A}$ if it satisfies the two conditions imposed on initial states. In applications it may make sense to restrict further the notion of state of the ealgebra in question.

An element a is an *agent* at S if there is a module name ν such that $S \models \mathrm{Mod}(a) = \nu$; the corresponding π_ν is the program $\mathrm{Prog}(a)$ of a, and $\mathrm{Fun}(\pi_\nu)$ is the vocabulary $\mathrm{Fun}(a)$ of a. Agent a is deterministic if $\mathrm{Prog}(a)$ is so.

$\mathrm{View}_a(S)$ is the reduct of S to vocabulary $\mathrm{Fun}(a) - \{\mathrm{Self}\}$ expanded with Self, which is interpreted as a. Think about $\mathrm{View}_a(S)$ as the local state of agent a corresponding to the global state S. (It is not necessary to define local states via global states; see [GM] for example.)

An agent a can *make a move* at S by firing $\mathrm{Prog}(a)$ at $\mathrm{View}_a(S)$ and changing S accordingly. As a part of the move, a may create new agents, *e.g.*, by importing reserve elements.

To perform a move of a deterministic agent a, fire

$$\mathrm{Updates}(a, S) = \mathrm{Updates}\Big(\mathrm{Prog}(a), \mathrm{View}_a(S)\Big).$$

Runs of a distributed ealgebra are defined below.

Cooperative Actions Consider a simple scenario with agents Sender and Receiver. If both are in mode Ready then Sender passes a value t_1 to Receiver who stores it at location $f(t_2)$. The transaction is atomic (that is, indivisible), but the Sender does not have access to $f(t_2)$ and the Receiver does not have access to t_1, and thus neither agent is able to perform the transaction. A special auxiliary agent is needed to do the job, and it may be convenient to view the auxiliary agent as a team with members Sender and Receiver.

Using functions Member_1 and Member_2 to specify the members of the team, we may write the following rule for the team, where Us is an alias for Self:

```
if Mode(Member₁(Us))=Mode(Member₂(Us))=Ready then
    f(t₂) := t₁
endif
```

In a similar way, one may have larger teams. Depending on need, teams may or may not be ordered.

6.3 Generalizations

6.3.1 Active Agents

Alter definition 6.2 as follows: Require that $\mathrm{Fun}(\mathcal{A})$ contains an additional unary relation name Active and that only agents satisfying the relation Active (*active agents*) can make moves. This is essentially a generalization; the original definition can be seen as a special case where all agents are active.

The new definition may be convenient, for example, when the initial state specifies all agents and their programs, and these agents are activated and deactivated during the evolution.

The same convenience can be achieved without altering the original definition. (This may be useful, for example, if you want to prove something about all distributed ealgebras and wish to restrict attention to the basic definition without losing generality.) Here is one way to do that. In order to indicate the program of a potential agent without making it an actual agent, use an auxiliary unary function name Mod'. $\mathrm{Active}(t)$ can be viewed as an abbreviation for $\mathrm{Mod}(t) = \mathrm{Mod}'(t)$ except if Active is the subject of an update instruction.

```
Active(t):=t₀
```

can be viewed as an abbreviation for

```
if t₀  then Mod(t):=Mod'(t)
else Mod(t):=undef
endif
```

6.3.2 Active Teams

The generalized definition of distributed ealgebras described in 6.3.1 is used in this sub-subsection. The following problem was raised by Dean Rosenzweig [R].

Consider a scenario with (agents called) players and (additional agents viewed as) teams. Players form a static universe, teams form another static universe, and each agent is assigned a program once and for all. Players are activated and deactivated during the evolution. A team is supposed to be active if and only if its members are active. It follows that activating one player may necessitate the tedious work of activating many teams. Is there a simple and elegant way to ensure that every team is active when and only when all its members are active?

One obvious solution is to make teams active all the time and augment the program of each team with a guard stating that all the members are active. A more radical solution is to make the notion of team a part of the logic of distributed ealgebras. It will be ensured automatically that a team is active if and only if all its members are active. (It may be also required that the moves made by a player or any team involving the player are linearly ordered; see the second property of runs in 6.5.1 in this connection.) If substantial programming convenience is demonstrated, use that solution.

The possibilities to pay a lesser price in logic complication for the advantages of the radical solution will be discussed elsewhere. In this connection, Rosenzweig suggested generalizing further the definition of 6.3.1 by letting a possibly compound Boolean term play the role of Active. For example, $\mathrm{Active}(v)$ may say that either v is a player satisfying an auxiliary relation Ac or v is a team with all members satisfying Ac.

6.4 Sequential Runs

We return to the basic definition of distributed ealgebras in 6.2.

A *pure sequential run* ρ of an ealgebra $\mathcal{A}$ is a sequence $\langle S_n : n < \kappa \rangle$ of states of $\mathcal{A}$, where S_0 is an initial state and every S_{n+1} is obtained from S_n by executing a move of an agent. The generalization to the case of external functions or external locations is relatively straightforward.

Stages Since S_i may be equal to S_j for some $i \neq j$, it may be convenient to speak about stages. Starting from stage 0, the run goes through stages 1, 2, *etc.*. Formally, stage i can be defined as the pair (i, S_i).

Quasi-sequential Runs An obvious generalization of a sequential run is a quasi-sequential run $\langle S_n : n < \kappa \rangle$, where each S_{n+1} is obtained from S_n by firing a collection A_n of agents. We do not mean that A_n is a team; since teams are agents, the definition of sequential runs does not exclude team moves. We mean that each $a \in A_n$ makes a move at S_n. If all agents are deterministic, then S_{n+1} is the result of firing $\bigcup\{\text{Updates}(S, a) : a \in A_n\}$.

Quasi-sequential runs may arise, for example, if you order moves in real (physical) time.

6.5 Partially Ordered Runs

Partially ordered computations are well known in the literature [L, Mz, KP, *etc.*] but we need to define our own version of that notion for our purposes here. We restrict attention to the case where moves are atomic and we use global states. Non-atomic moves have been explored in [BGR]. A simple notion of runs in [GM] does not use global states.

Let us recall some well known notions. A *poset* is a partially ordered set. An *initial segment* of a poset P is a substructure X of P such that if $x \in X$ and $y < x$ in P then $y \in X$. Since X is a substructure, $y < x$ in X if and only if $y < x$ in P whenever $x, y \in X$. A *linearization* of a poset P is a linearly ordered set P' with the same elements such that if $x < y$ in P then $x < y$ in P'.

6.5.1 Runs

For simplicity, we restrict attention to pure runs and deterministic agents. A *run* ρ of a distributed ealgebra $\mathcal{A}$ can be defined as a triple (M, A, σ) satisfying the following conditions 1–4.

1 M is a partially ordered set, where all sets $\{y : y \leq x\}$ are finite.

Elements of M represent *moves* made by various agents during the run. If $y < x$ then x starts when y is already finished; that explains why the set $\{y : y \leq x\}$ is finite.

2 A is a function on M such that every nonempty set $\{x : A(x) = a\}$ is linearly ordered.

$A(x)$ is the agent performing move x. The moves of any single agent are supposed to be linearly ordered.

3 σ assigns a state of $\mathcal{A}$ to the empty set and each initial segment of M; $\sigma(\emptyset)$ is an initial state.

$\sigma(X)$ is the result of performing all moves in X.

4 The coherence condition: If x is a maximal element in a finite initial segment X of M and $Y = X - \{x\}$, then $A(x)$ is an agent in $\sigma(Y)$ and $\sigma(X)$ is obtained from $\sigma(Y)$ by firing $A(x)$ at $\sigma(Y)$.

Intuitively, a run can be seen as the common part of histories of the same computation recorded by various observers. We hope to address this issue elsewhere.

If agents are not necessarily deterministic, we have to define moves as state transformers and make the coherence condition more precise:

4* If x is a maximal element in a finite initial segment X of M and $Y = X - \{x\}$, then $A(x)$ is an agent in $\sigma(Y)$, x is a move of $A(x)$ and $\sigma(X)$ is obtained from $\sigma(Y)$ by performing x at $\sigma(Y)$.

A run ρ' is an *initial segment* of a run ρ if (i) the move poset of ρ' is an initial segment of the move poset of ρ and (ii) the agent and state functions of ρ' are restrictions of those in ρ. A run ρ' is a *linearization* of ρ if the move poset of ρ' is a linearization of that of ρ, the agent function of ρ' is that of ρ, and the state function of ρ' is a restriction of that of ρ. Linearizations are sequential runs. A state S of is *reachable* in a run ρ if it belongs to the range of the state function of ρ.

Corollary 6.1 *All linearizations of the same finite initial segment of ρ have the same final state.*

Corollary 6.2 *A property holds in every reachable state of a run ρ if and only if it holds in every reachable state of every linearization of ρ.*

6.6 Real-time Computations

Real-time semantics appears in [BGR]. Ealgebras with clocks made their debut in [GM]. We will have to address the issue of real time elsewhere.

Bibliography

B Egon Börger, "Logic Programming: The Evolving Algebra Approach", in [PS].

BG Andreas Blass and Yuri Gurevich, "Evolving Algebras and Linear Time Hierarchy", in [PS].

BGR Egon Börger, Yuri Gurevich and Dean Rosenzweig, "The Bakery Algorithm: Yet Another Specification and Verification", this volume.

BR Egon Börger and Dean Rosenzweig, "The WAM — Definition and Compiler Correctness", to appear in "Logic Programming: Formal Methods and Practical Applications", Eds. C. Beierle and L. Pluemer, North-Holland, 1994.

G1 Yuri Gurevich, "Logic and the challenge of computer science", In "Current Trends in Theoretical Computer Science", Ed. E. Börger, Computer Science Press, 1988, 1–57.

G2 Yuri Gurevich, "Evolving Algebras: An Attempt to Discover Semantics ", Bull. EATCS 43 (1991), 264–284; a slightly revised version in "Current Trends in Theoretical Computer Science", Eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292.

GH Yuri Gurevich and James K. Huggins, "The Semantics of the C Programming Language", Springer Lecture Notes in Computer Science 702, 1993, 274–308.

GM Yuri Gurevich and Raghu Mani, "Group Membership Protocol: Formal Specification and Verification", this volume.

GR Paola Glavan and Dean Rosenzweig, "Communicating Evolving Algebras", in "Computer Science Logic", eds. E. Börger et al., Lecture Notes in Computer Science 702, Springer, 1993, 182–215.

H James K. Huggins, "Kermit: Specification and Verification", this volume.

KP Shmuel Katz and Doron Peled, "Defining Conditional Independence Using Collapses", Theoretical Computer Science 101 (1992), 337–359.

L Leslie Lamport, "On Interprocess Communication", Distributed Computing 1 (1986), 77–101.

Mt Yuri Matijasevich, "Enumerable sets are Diophantine", Soviet Math. Doklady 11:2 (1970), 354–358.

Mz Antoni Mazurkiewicz, "Trace Theory", Springer Lecture Notes in Computer Science 255 (1987), 279–324.

PS B. Pehrson and I. Simon, Editors, "IFIP 13th World Computer Congress 1994, Volume I: Technology/Foundations", Elsevier, Amsterdam, to appear.

R Dean Rosenzweig, Private Communication.

The Semantics of the C Programming Language

Yuri Gurevich*and James K. Huggins*
EECS Department, University of Michigan, Ann Arbor, MI 48109-2122, USA

February 19, 1993

This paper first appeared in [GH2], and incorporates the corrections indicated in [GH3].

0 Introduction

We present formal operational semantics for the C programming language. Our starting point is the ANSI standard for C as described in [KR]. Knowledge of C is not necessary (though it may be helpful) for comprehension, since we explain all relevant aspects of C as we proceed.

Our operational semantics is based on evolving algebras. An exposition on evolving algebras can be found in the tutorial [Gu]. In order to make this paper self-contained, we recall the notion of a (sequential) evolving algebra in Sect. 0.1.

Our primary concern here is with semantics, not syntax. Consequently, we assume that all syntactic information regarding a given program is available to us at the beginning of the computation (via static functions). We intended to cover all constructs of the C programming language, but not the C standard library functions (*e.g.* `fprintf()`, `fscanf()`). It is not difficult to extend our description of C to include any desired library function or functions.

Evolving algebra semantic specifications may be provided on several abstraction levels for the same language. Having several such algebras is useful, for one can examine the semantics of a particular feature of a programming language at the desired level of abstraction, with unnecessary details omitted. It also makes comprehension easier. We present a series of four evolving algebras, each a refinement of the previous one. The final algebra describes the C programming language in full detail.

Our four algebras focus on the following topics respectively:

1. Statements (*e.g.* `if`, `for`)
2. Expressions
3. Memory allocation and initialization
4. Function invocation and return

What about possible errors, *i.e.*, division by zero or de-referencing a pointer to an invalid address? These issues are very implementation-dependent. Even what constitutes an error is implementation-dependent. If an external function does not produce any value in a state where a value is expected, the evolving algebra will be stalled in that state forever. It is natural to suppose that if an external function does produce a value, it is of the appropriate type. One may want to augment the guards of transition rules to check for errors; in this way, the evolving Nalgebra will halt on error conditions (and may even output an error message if desired). There are more subtle ways to handle errors. We ignore the issue here.

To reflect the possibility of different implementations, our evolving algebras contain implementation-dependent parameters. For example, the set of values "storable" in a pointer variable is implementation-dependent. Thus, each of our four evolving algebras gives rise to a family of different evolving algebras.

*Partially supported by ONR and NSF.

0.1 Evolving Algebras

An evolving algebra $\mathcal{A}$ is an abstract machine. Here we restrict attention to sequential evolving algebras. The *signature* of $\mathcal{A}$ is a (finite) collection of function names, each name having a fixed arity. A state of $\mathcal{A}$ is a set, the *superuniverse*, together with interpretations of the function names in the signature. These interpretations are called *basic functions* of the state. The superuniverse does not change as $\mathcal{A}$ evolves; the basic functions may.

Formally, a basic function of arity r (*i.e.* the interpretation of a function name of arity r) is an r-ary operation on the superuniverse. (We often use basic functions with $r = 0$; such basic functions will be called *distinguished elements*.) But functions naturally arising in applications may be defined only on a part of the superuniverse. Such partial functions are represented by total functions in the following manner.

The superuniverse contains distinct elements *true, false, undef* which allow us to deal with relations (viewed as binary functions with values *true* or *false*) and partial functions (where $f(\overline{a}) = undef$ means f is undefined at the tuple $\overline{a}$). These three elements are *logical constants*. Their names do not appear in the signature; this is similar to the situation in first-order logic with equality where equality is a logical constant and the sign of equality does not appear in the signature. In fact, we use equality as a logical constant as well.

Further, a *universe* U is a special type of basic function: a unary relation usually identified with the set $\{x : U(x)\}$. The universe $Bool = \{true, false\}$ is another logical constant. When we speak about, say, a function f from a universe U to a universe V, we mean that formally f is a unary operation on the superuniverse such that $f(a) \in V$ for all $a \in U$ and $f(a) = undef$ otherwise. We use self-explanatory notations like $f : U \to V$, $f : U_1 \times U_2 \to V$, and $f : V$. The last means that the distinguished element f belongs to V.

In principle, a program of $\mathcal{A}$ is a finite collection of transition rules of the form

$$\textbf{if } t_0 \textbf{ then } f(t_1, \ldots, t_r) := t_{r+1} \textbf{ endif} \tag{1}$$

where t_0, $f(t_1, \ldots, t_r)$, and t_{r+1} are closed terms (*i.e.* terms containing no free variables) in the signature of $\mathcal{A}$. An example of such a term is $g(h_1, h_2)$ where g is binary and h_1 and h_2 are zero-ary. The meaning of the rule shown above is this: Evaluate all the terms t_i in the given state; if t_0 evaluates to *true* then change the value of the basic function f at the value of the tuple $(t_1, .., t_r)$ to the value of t_{r+1}, otherwise do nothing.

In fact, rules are defined in a slightly more liberal way; if k is a natural number, $b_0, \ldots, b_k$ are terms and $C_0, \ldots, C_{k+1}$ are sets of rules then both of the following are rules:

if b_0 **then** C_0
elseif b_1 **then** C_1
$\vdots$
elseif b_k **then** C_k
else C_{k+1}
endif

if b_0 **then** C_0
elseif b_1 **then** C_1
$\vdots$
elseif b_k **then** C_k
endif

Since the C_i are sets of rules, nested transition rules are allowed (and occur frequently). To save space, we abbreviate the series of **endif**'s at the tail of a transition rule by **ENDIF**.

A program is a set of rules. It is easy to transform a program to an equivalent program comprising only rules of the stricter form (1). We use rules of the more liberal form, as well as macros (textual abbreviations), for brevity.

How does $\mathcal{A}$ evolve from one state to another? In a given state, the demon (or interpreter) evaluates all the relevant terms and then makes all the necessary updates. If several updates contradict each other (trying to assign different values to the same basic function at the same place), then the demon chooses nondeterministically one of those updates to execute.

We call a function (name) f *dynamic* if the demon (interpreter) may change f as the algebra evolves; *i.e.* if an assignment of the form $f(t_1, \ldots, t_r) := t_0$ appears anywhere in the transition rules. Functions which are not dynamic are called *static*. To allow our algebras to interact conveniently with the outside world, we also make use of *external* functions within our algebra. External functions are syntactically static (that is, never changed by rules), but have their values determined by an oracle. Thus, an external function may have different values for the same arguments as the algebra evolves.

0.2 Acknowledgements

An earlier version of this paper appeared as a technical report [GH1]. We gratefully acknowledge comments made by on the original report by Egon Börger, Andre Burago, Martin J. Dürst, Stefano Guerrini, Raghu Mani, Arnd Poetzsch-Heffter, Dean Rosenzweig, and Marcus Vale, as well as comments in the errata made by Lars Ole Andersen, L. Douglas Baker, Arnd Poetzsch-Heffter, Thomas Tsukada, and Chuck Wallace.

1 Algebra One: Handling C Statements

Our first evolving algebra models the control structures of C.

1.1 Some Basic Functions

A universe *tasks* consists of elements representing tasks to be accomplished by the program interpreter. The notion of task is a general one: *e.g.*, a task may be the execution of a statement, initialization of a variable, or the evaluation of an expression. The elements of this universe are dependent on the particular C program being executed by the abstract machine. It is often useful to mark a given task with tags indicating its nature. This gives rise to a universe of *tags*.

A distinguished element *CurTask: tasks* indicates the current task. In order to execute tasks in the proper order, a static function *NextTask: tasks* $\rightarrow$ *tasks* indicates the next task to be performed once the current task has been completed. A static function *TaskType: tasks* $\rightarrow$ *tags* indicates the action to be performed by the task.

A universe *results* contains values which may appear as the result of a computation.

1.2 Macro: Moveto

Often, we transfer control to a particular task, modifying *CurTask* to indicate the transfer of control. In Algebra Two, the rules for modifying *CurTask* will change somewhat; in order to facilitate this change, we will use the *Moveto(Task)* macro each time that we wish to transfer control. For now, the definition of *Moveto(Task)* is shown in Fig. 1.

Moveto(Task)

CurTask := Task

Figure 1: Definition of the *Moveto(Task)* macro.

1.3 Statement Classification in C

According to [KR], there are six categories of statements in C:

1. Expression statements, which evaluate the associated expression.

2. Selection statements (`if` and `switch`).
3. Iteration statements (`for`, `while`, and `do-while`).
4. Jump statements (`goto`, `continue`, `break`, and `return`).
5. Labeled statements (`case` and `default` statements used within the scope of a `switch` statement, and targets of `goto` statements).
6. Compound statements, consisting of a (possibly empty) list of local variable declarations and a (possibly empty) list of statements.

.

1.4 Expression Statements

An expression statement has one of the following forms:

expression-statement → ;
expression-statement → *expression* ;

To execute an expression statement, evaluate the attached expression (if any), even though the resulting value will not be used. While this may seem unnecessary, note that the evaluation of an expression in C may generate side-effects (such as assigning a value to a variable). Note also that the evaluation of an expression may not halt. In this algebra, the evaluation of expressions is handled by an external function *TestValue: tasks → results*.

Since expression statements perform no additional work, the algebra simply proceeds to the next task. The transition rule for expression tasks is shown in Fig. 2.

---

if *TaskType(CurTask) = expression* **then**
    Moveto(NextTask(CurTask))
endif

Figure 2: Transition rule for expression tasks.

1.5 `if` Statements

There are two types of selection statements in C: `if` statements and `switch` statements. An `if` statement has one of the following forms:

if-statement → `if` (*expression*) *statement1*
if-statement → `if` (*expression*) *statement1* `else` *statement2*

where *statement1* and *statement2* are statements.

To execute an `if` statement, begin by evaluating the guard expression. If the resulting value is non-zero, execute *statement1*. If the resulting value is zero and an `else` clause is present, execute *statement2*. otherwise, execute the statement following the `if` statement. Static partial functions *TrueTask: tasks → tasks* and *FalseTask: tasks → tasks* indicate the task to be performed if the guard of the `if` statement evaluates to a non-zero value or zero, respectively.

The branching decision made in the `if` statement is represented by an element of the *tasks* universe for which the *TaskType* function returns *branch*. We illustrate a typical `if` statement with the graph in Fig. 3, where ovals represent tasks, labeled arcs represent the corresponding unary functions, and boxes represent subgraphs. If an `else` clause is not present in an `if` statement, the corresponding task graph omits the lower portion of Fig. 3, with the *FalseTask* function connecting the *branch* task to the task following the `if` statement. The transition rule for *branch* tasks is shown in Fig. 4.

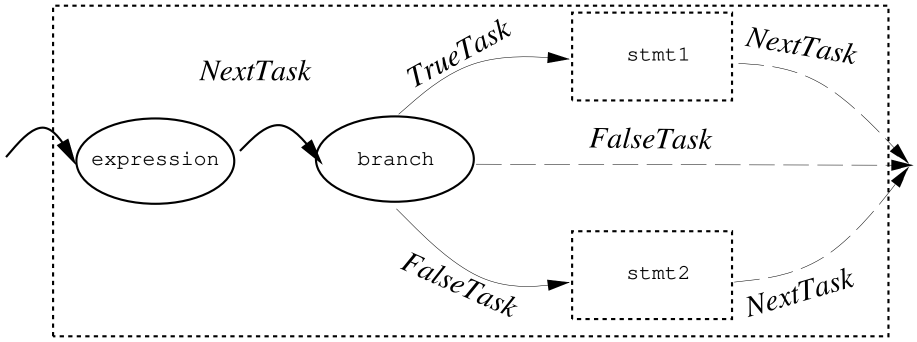


Figure 3: A typical `if` statement.

---

if *TaskType(CurTask) = branch* **then**
  if *TestValue(CurTask)* $\neq$ *0* **then**
    Moveto(TrueTask(CurTask))
  elseif *TestValue(CurTask) = 0* **then**
    Moveto(FalseTask(CurTask))
ENDIF

Figure 4: Transition rule for *branch* tasks.

Remark. Fig. 3 shows a typical `if` statement, but it is not representative of all `if` statements. The presence of a jump statement in *statement1* or *statement2* may cause *NextTask* to point to a different task than the one which immediately follows the `if` statement.

1.6 `switch` Statements

A `switch` statement has the following form:

switch-statement $\rightarrow$ `switch ( `*expression*` )` *body*

where *body* is a statement, usually compound.

Within the body of a `switch` statement there are (usually) labeled `case` and `default` statements. Each `case` or `default` is associated with the smallest enclosing `switch` statement.

To execute a `switch` statement, evaluate the guard expression, and within the body of the switch, transfer control to the `case` statement for the `switch` whose labeled value matches the value of the expression, or to the `default` statement for the `switch`, whichever comes first. If no such statement is found, transfer control to the statement following the `switch` statement.

Labels on `case` statements are required to be unique within a `switch`, and a `switch` may not have more than one `default` statement. Thus, for a given expression value, there is exactly one statement to which control should be passed. A static partial function *SwitchTask: tasks* $\times$ *results* $\rightarrow$ *tasks* indicates the next task to be executed for a given expression value. We illustrate a typical `switch` statement with the graph in Fig. 5; with regard to the possible effects of embedded jump statements, see the remark in Sect. 1.5. The rule for `switch` tasks is shown in Fig. 6.

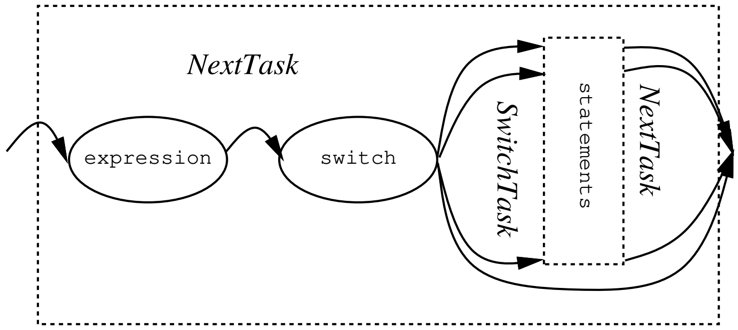


Figure 5: A typical `switch` statement.

---

if *TaskType(CurTask) = switch* **then**
Moveto(SwitchTask(CurTask, TestValue(CurTask)))
endif

Figure 6: Transition rules for *switch* tasks.

1.7 `while` Statements

A `while` statement has the following form:

while-statement $\rightarrow$ `while` `(` *expression* `)` *body*

where *body* is a statement.

To execute a `while` statement, keep evaluating the guard expression until the value of the expression becomes zero. Each time that the value of the guard expression is not zero, execute *body*.

We illustrate a typical `while` statement with the graph in Fig. 7; with regard to the possible effects of embedded jump statements, see the remark in Sect. 1.5. Since the only types of tasks used to represent `while` statements are the *expression* and *branch* tasks, our previously-presented transition rules are sufficient to model the behavior of `while` statements.

Note that it is possible to enter a `while` loop by means of a `goto` statement, thus circumventing the initial test of the expression at the beginning of the loop. The ANSI standard [KR] does not give specific semantics for such behavior. In such a situation, our abstract machine would continue as if the loop had been entered normally (*i.e.*, after completion of the statement body, control returns to the guard expression to be evaluated). We nbelieve this is a reasonable interpretation of such an event.

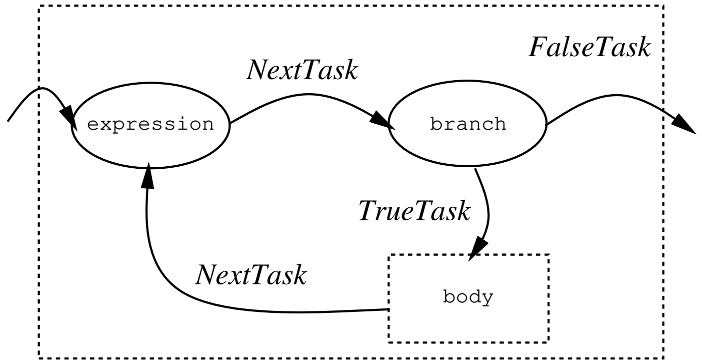


Figure 7: A typical `while` statement.

1.8 `do-while` Statements

A `do-while` statement has the following form:

do-while-statement → `do` *body* `while ( ` *expression* ` ) ;`

where *body* is a statement.

`do-while` statements are identical to `while` statements except that the guard expression and statement body are visited in the opposite order. We illustrate a typical `do-while` with the graph shown in Fig. 8; with regard to the possible effects of embedded jump statements, see the remark in Sect. 1.5. (Note the similarity between this graph and that of the `while` loop.) As with `while` loops, no new transition rules are required to model the behavior of `do-while` statements.

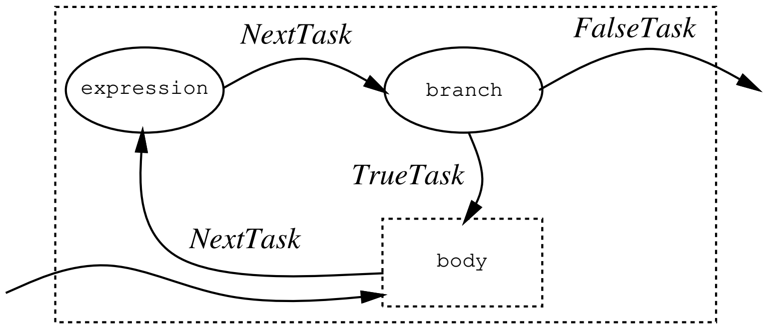


Figure 8: A typical `do-while` statement.

1.9 `for` Statements

The most complete form of the `for` statement is:

for-statement → `for ( ` *initializer* ` ; ` *test* ` ; ` *update* ` ) ` *body*

where *initializer*, *test*, and *update* are expressions, any of which may be omitted, and *body* is a statement, usually compound. We begin by describing the behavior and representation of a `for` statement when all expressions are present.

In executing a `for` statement, begin by evaluating the initializer. Evaluate the test next; if the result is non-zero, execute the body and evaluate the update (in that order) and re-evaluate the test. If the value of the test is zero, transfer control to the statement following the `for` loop.

We illustrate a typical `for` statement with the graph in Fig. 9; with regard to the possible effects of embedded jump statements, see the remark in Sect. 1.5. Again, no new transition rules are required to model the behavior of `for` statements.

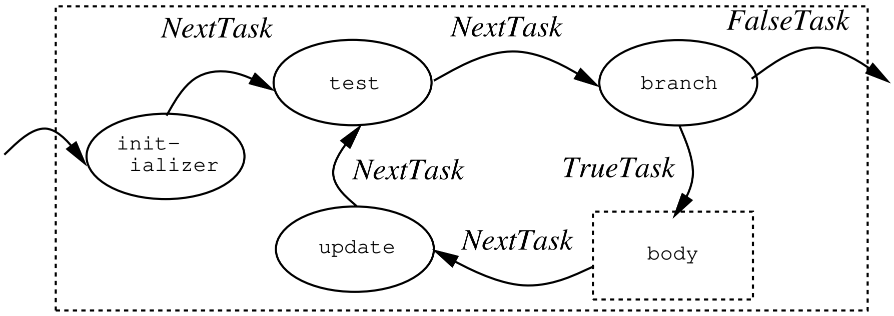


Figure 9: A typical `for` statement.

The graphs for `for` loops missing one or more of the three expressions (initializer, test, and update) omit the corresponding tasks, with *NextTask* pointing to the next task in the graph sequence. If the test is omitted, both the test and the *branch* task are omitted, which creates an infinite loop (which may still be broken through the use of jump statements).

1.10 Jump Statements

A jump statement has one of the following forms:

jump-statement → `goto` *identifier* ;
jump-statement → `continue` ;
jump-statement → `break` ;
jump-statement → `return` ;
jump-statement → `return` *expression* ;

Each of these jump statements is a command indicating that control should be unconditionally transferred to another task in the task graph:

- `goto` statements indicate directly the task to which control passes.
- `continue` statements may only occur within the body of an iteration statement. For a given `continue` statement C, let S be the smallest iteration statement which includes C. Executing C transfers control to the task within S following the statement body of S: *e.g.*, for `for` statements, control passes to the update expression, while for `while` statements, control passes to the guard expression.
- `break` statements may occur within the body of an iteration or `switch` statement. For a given `break` statement B, let S be the smallest iteration or `switch` statement which includes B. Executing B transfers control to the first task following S.
- `return` statements occur within the body of function abstractions, indicating that the current function execution should be terminated. A more complete discussion of `return` statements will be presented in Algebra Four, where function abstractions are presented. For now, we assert that executing a `return` statement should set *CurTask* to *undef*, which will bring a halt to the algebra, since we only have one function (`main`) being executed.

The *NextTask* function contains the above (static) information for jump statement tasks. Thus, the transition rule for jump statements (shown in Fig. 10) is trivial.

if *TaskType(CurTask) = jump* **then**
    Moveto(NextTask(CurTask))
endif

Figure 10: Transition rule for *jump* tasks.

1.11 Labeled Statements

A labeled statement has one of the following forms:

labeled-statement → *identifier* : *statement*
labeled-statement → `case` *constant-expression* : *statement*
labeled-statement → `default` : *statement*

Statement labels identify the targets for control transfer in `goto` and `switch` statements. *NextTask* and *SwitchTask* return the appropriate tasks in each case; no further transition rules are needed.

1.12 Compound Statements

A compound statement has the following form:

compound-statement → { *declaration-list statement-list* }

where the declaration and/or statement lists may be empty.)

Since *NextTask* indicates the order in which tasks are processed, we have no need for rules concerning compound statements. Each statement or declaration in a compound statement is linked to its successor via *NextTask*. (Declarations are not treated until Algebra Three; nonetheless, the same principle holds for declaration tasks.)

1.13 Initial and Final States

We assert that initially, *CurTask* indicates the first task of the first statement of the program.

A final state in our algebra is any state in which *CurTask = undef*. In this state, no rules will be executed, since *TaskType(undef) = undef.*

2 Algebra Two: Evaluating Expressions

Our second evolving algebra refines the first and focuses on the evaluation of expressions.

We replace each occurrence of a task of type *expression* from the first algebra with numerous tasks reflecting the structure of the expression. Also, *TestValue* is now an internal, dynamic function.

In Algebra Two, we treat the evaluation of expressions at a relatively high level of abstraction. We map variable identifiers to memory locations through external functions. We also treat function invocations as expressions whose values are provided by external functions. In Algebras Three and Four we will eliminate these abstractions.

2.1 New Basic Functions Related To Memory Management

In C, one may (re)cast types. For example, one may cast a pointer to a structure into a pointer to an array of characters. Thus, one can access the individual bytes of most values which might exist during the execution of the program.[1]

A static function *Size: typename* $\rightarrow$ *integer* indicates how many bytes are used by a particular value type in memory. A dynamic function *Memory: addresses* $\rightarrow$ *bytes* indicates the values stored in memory at a given byte. Since most values of interest are larger than a byte, we need a means for storing members of *results* as individual bytes. For example, assume that the `int` value 258 is represented in the memory of a particular system by the four (eight-bit) bytes 0, 0, 1, and 2, stored consecutively. We need a way to go from a value in *results* (*e.g.*, 258) to its component bytes (0, 0, 1, 2) and vice versa.

A static partial (n+1)-ary function *ByteToResult: typename* $\times$ *byte*^{n} $\rightarrow$ *results* converts the memory representation of a value of the specified basic type into its corresponding value in the *results* universe. Here n is the maximum number of bytes used by the memory representation of any particular basic type (and is implementation-dependent). For types whose memory representations are less than n bytes in length, we ignore any unused parameters. In our example above, *ByteToResult(int,0,0,1,2) = 258*.

A static partial function *ResultToByte: results* $\times$ *integer* $\times$ *typename* $\rightarrow$ *byte* yields the specified byte of the memory representation of the specified value from the specified universe. This function can be thought of as the inverse of *ByteToResult*. In our example above, *ResultToByte(258,3,int) = 2*. (We assume tacitly that the arguments of *ResultToByte* uniquely define the value of the function, which is the case in all the implementations that we know.)

We define an abbreviation *MemoryValue: address* $\times$ *typename* $\rightarrow$ *results*, which indicates the value of the specified type being stored in memory beginning at the indicated address. *MemoryValue (addr,type)* abbreviates *ByteToResult (type, Memory(addr), Memory(addr+1), ..., Memory(addr + Size(type) - 1))*.

2.2 Other New Basic Functions

Two sub-universes of *results*, the universe of computational results, are of particular interest in Algebra Two. A universe *bytes* contains those values which may be "stored" in a `char` variable. (This universe is usually identical to $\{0,1,\ldots,255\}$, but we prefer the more general definition.) A universe *addresses* contains positive integers corresponding to valid memory locations. This is also the universe of values which may be stored in a pointer-type variable. (Of course, these two universes are implementation-dependent.)

A universe *typename* contains elements representing the different types of storable values. A static partial function *ValueType: tasks* $\rightarrow$ *typename* indicates the type of the resulting value when an expression has been evaluated.

Static partial functions *LeftTask, RightTask: tasks* $\rightarrow$ *tasks* indicate the left and right operands of binary operators whose order of evaluation is not defined within C (*e.g.*, `+`). A static partial function *Parent: tasks* $\rightarrow$ *tasks* indicates the parent (*i.e.*, closest enclosing) expression for a given expression. For expressions which are not contained in any other expressions, *Parent* returns the corresponding *branch* task which uses the expression (if one exists) or *undef* (if none exists). A static partial function *WhichChild: tasks* $\rightarrow$ {*left, right, only, test, ...*} (where *left*, *etc.* are members of the *tags* universe) indicates the relationship between a task and its parent.

Dynamic partial functions *LeftValue, RightValue: tasks* $\rightarrow$ *results* indicate the results of evaluating the left and right operands of binary operators with ambiguous evaluation order. Similarly, a dynamic partial function *OnlyValue: tasks* $\rightarrow$ *results* indicates the result of evaluating the single operand of a unary operator. A static partial function *ConstVal: tasks* $\rightarrow$ *results* indicates the values of program constants.

[1] The distinguished value `void` is an example of a value which cannot be accessed in this manner.

2.3 Macro: DoAssign

Our rules for assignment to memory are a little complicated, since a given assignment may require an arbitrarily large number of updates to the *Memory* function. We need rules which perform a loop to make those arbitrarily large number of updates in a systematic fashion.[2]

To facilitate this loop, we use several distinguished elements. *CopyValue: results* denotes the value to be copied. *CopyType: typename* denotes the type of value to be copied. *CopyLocation: address* denotes the location to which the value is to be copied. *CopyByte: integer* denotes which byte of the representation of *CopyValue* is being copied into memory. *OldTask: tasks* denotes the task which invoked the memory copying procedure. *CopyTask: tasks* is a static distinguished element used to indicate that the copying procedure should begin.

We will invoke the copying procedure using the *DoAssign(address, value, type)* macro, defined in Fig. 11. The copying process itself is relatively straightforward. We utilize the distinguished element *CopyByte* to denote which byte of the memory representation of *CopyValue* we are copying into memory at a given moment in time. We copy bytes singly, incrementing the value of *CopyByte* after each assignment to memory, halting when all bytes have been copied. The transition rule for copying to memory is shown in Fig. 12.

DoAssign(address, value, type)

CopyValue := value
CopyType := type
CopyLocation := address
CopyByte := 0
OldTask := CurTask
CurTask := CopyTask

Figure 11: Definition of the *DoAssign* macro.

---

if *CurTask = CopyTask* **then**
    if *CopyByte < Size(CopyType)* **then**
        Memory(CopyLocation + CopyByte) :=
            ResultToByte(CopyValue, CopyByte, CopyType)
        CopyByte := CopyByte + 1
    elseif *CopyByte = Size(CopyType)* **then**
        CurTask := NextTask(OldTask)
    endif
endif

Figure 12: Transition rule for copying to memory.

[2]Most computer systems provide a means for memory assignments in units larger than a byte, but the particular sizes available are implementation-dependent. We thus present rules using the lowest-common denominator, the byte.

2.4 Macro: ReportValue

When we process tasks corresponding to expression evaluation, we assign the value of an evaluated expression to the appropriate storage function in the parent expression (*e.g. LeftValue(Parent(CurTask))*). We use the *ReportValue* macro (defined in Fig. 13) to accomplish this.

ReportValue(value)

if *WhichChild(CurTask) = left* **then**
    LeftValue(Parent(CurTask)) := value
elseif *WhichChild(CurTask) = right* **then**
    RightValue(Parent(CurTask)) := value
elseif *WhichChild(CurTask) = only* **then**
    OnlyValue(Parent(CurTask)) := value
elseif *WhichChild(CurTask) = test* **then**
    TestValue(Parent(CurTask)) := value
endif

Figure 13: Definition of the *ReportValue* macro.

2.5 Macros: EvaluateOperands and Moveto

In C, as described by [KR], many binary operators (such as the assignment operator "`=`") do not have a defined order of evaluation: either operand may be evaluated first. When one or more operands of such an operator generate side effects (as in "`a[i] = i++`"), the value or the side-effects generated by the expression may depend upon the order of evaluation. Writing such code is usually unwise, since such code may not be portable; however, an optimizing compiler may take advantage of this ambiguity to generate code which minimizes the resources required to perform a particular computation [ASU].

For expressions involving such operators, our algebra must be flexible enough to reflect any possible evaluation order of an expression's operands, even if this decision is made at run-time. While we believe most compilers make this decision at compile-time, we must still provide a mechanism for making this decision dynamically. (If this decision is always made statically in a particular system, the algebra may be explicitly structured to incorporate those static decisions into the task graph). We will use an external function *ChooseTask* to represent this decision.

We illustrate how expressions with such operators are represented in our algebra by the graph in Fig. 14. A dynamic function *Visited: tasks* $\rightarrow$ {*left, right, both, neither*} indicates which subexpressions have been evaluated at a given moment. Initially, *Visited* has the value *neither* for all tasks.

To evaluate expressions of this type, begin by evaluating the sub-expression indicated by *ChooseTask*. When that sub-expression has been evaluated, evaluate the other sub-expression. Finally, when both expressions have been evaluated, perform the desired operation.

To handle the portion of this behavior dealing with the operator task, we use the *EVALUATE OPERANDS* macro. Informally, the macro means "Evaluate both operands in the order given by *ChooseTask*; when both operands are evaluated, then do ...". To handle the portion of this behavior dealing with movement between subtasks, we redefine the *Moveto(Task)* macro to jump directly between the subtasks of an operator of this type. The definitions for *EVALUATE OPERANDS* and *Moveto* are shown in Fig 15 and Fig. 16.

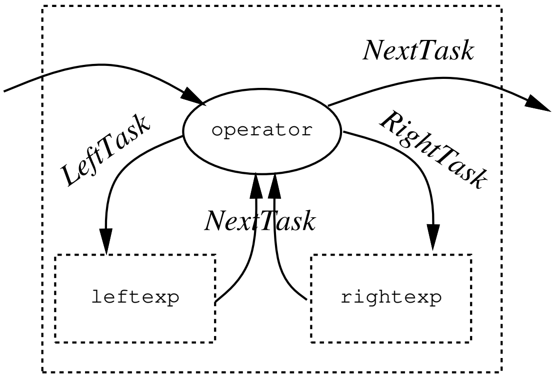


Figure 14: Binary operators.

EVALUATE OPERANDS WITH
    statements
END EVALUATE

if *Visited(CurTask) = neither* **then**
    if *ChooseTask(CurTask) = LeftTask(CurTask)* **then**
        Visited(CurTask) := left
    elseif *ChooseTask(CurTask) = RightTask(CurTask)* **then**
        Visited(CurTask) := right
    endif
    Moveto(ChooseTask(CurTask))
elseif *Visited(CurTask) = both* **then**
    Visited(CurTask) := neither
        statements
endif

Figure 15: Definition of the *EVALUATE OPERANDS* macro.

Moveto(Task)

if *Visited(Task) = neither* **then**
  CurTask := Task
elseif *Visited(Task) = both* **then**
  CurTask := Task
elseif *Visited(Task) = left* **then**
  CurTask := RightTask(Task)
  Visited(Task) := both
elseif *Visited(Task) = right* **then**
  CurTask := LeftTask(Task)
  Visited(Task) := both
endif

Figure 16: Revised definition of the *Moveto(Task)* macro.

2.6 Comma Operators

A comma expression has the following form:

comma-expression → *expr1* , *expr2*

where *expr1* and *expr2* are expressions.

To evaluate a comma expression, evaluate *expr1* and *expr2*, left to right, returning the value of *expr2* as the value of the parent expression. (Though it may seem unnecessary to evaluate the first expression since we ignore its value, recall that expressions in C may generate side-effects.) We represent comma expressions as a sequence of two expressions linked by the *NextTask* function. Thus, no additional transition rules are needed to process comma operators.

2.7 Conditional Expressions

A conditional expression has the following form:

conditional-expression → *expr1* ? *expr2* : *expr3*

where *expr1*, *expr2*, and *expr3* are expressions.

To evaluate a conditional expression, evaluate *expr1*. If the resulting value is non-zero, evaluate *expr2* and return its value as the value of the parent expression; otherwise, evaluate *expr3* and return its value as the value of the parent expression.

We will represent conditional expressions in our algebra in a manner similar to that in which we represent conditional statements, as illustrated in Fig. 17. The tasks corresponding to the center and right subexpressions will update the appropriate *Value* function for the parent expression upon completion of the evaluation of the subexpression. No new transition rules are needed to handle conditional expressions.

2.8 Logical OR Expressions

A logical OR expression has the following form:

logical-OR-expression → *expr1* || *expr2*

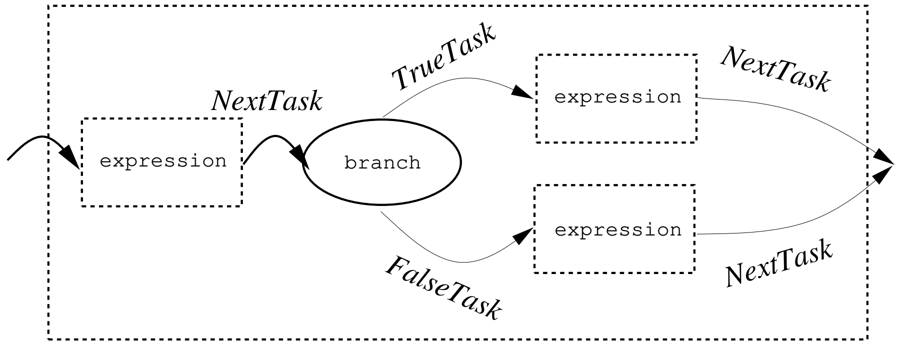


Figure 17: Conditional expression.

where *expr1* and *expr2* are expressions.

To evaluate a logical OR expression, start by evaluating *expr1*. If the result is non-zero, the value of the parent expression is 1 and *expr2* is not evaluated. Otherwise, the value of the parent expression is the value of *expr2*, with non-zero values coerced to 1.

To represent a logical OR expression, we will introduce two new task types, *OR* and *makeBool*. We illustrate how logical OR expressions are represented in our algebra by the graph in Fig. 18.

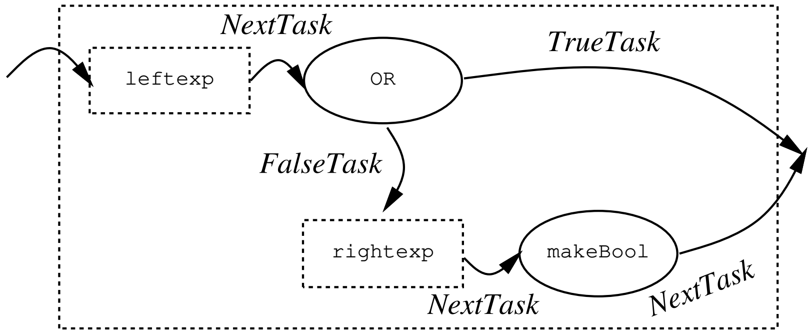


Figure 18: Logical OR expression.

In processing the *OR* task, the value of *expr1* will be examined. If the value is not zero, the rules for *OR* tasks will set the value of the expression to 1 and end processing of the parent expression. Otherwise, the rules will pass control to the tasks which evaluate *expr2*. The rules for *makeBool* tasks will examine the value of *expr2* and coerce it to 0 or 1. The transition rules for *OR* and *makeBool* tasks are shown in Fig. 19 and Fig. 20.

2.9 Logical AND expressions

A logical AND expression has the following form:

logical-AND-expression → *expr1* `&&` *expr2*

where *expr1* and *expr2* are expressions.

To evaluate an AND expression, begin by evaluating *expr1*. If the resulting value is 0, the value of the parent expression is 0. Otherwise, the value of *expr2* (coerced to 0 or 1) is the value of the parent expression.

```
if TaskType(CurTask) = OR then
    if OnlyValue(CurTask) ≠ 0 then
        ReportValue(1)
        Moveto(TrueTask(CurTask))
    elseif OnlyValue(CurTask) = 0 then
        Moveto(FalseTask(CurTask))
ENDIF
```

Figure 19: Transition rule for *OR* tasks.

```
if TaskType(CurTask) = makeBool then
    if OnlyValue(CurTask) ≠ 0 then
        ReportValue(1)
    elseif OnlyValue(CurTask) = 0 then
        ReportValue(0)
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 20: Transition rule for *makeBool* tasks.

The representation of logical AND expressions is similar to that of logical OR expressions, as illustrated in Fig. 21. The transition rule for *AND* tasks (used in such representations) is shown in Fig. 22.

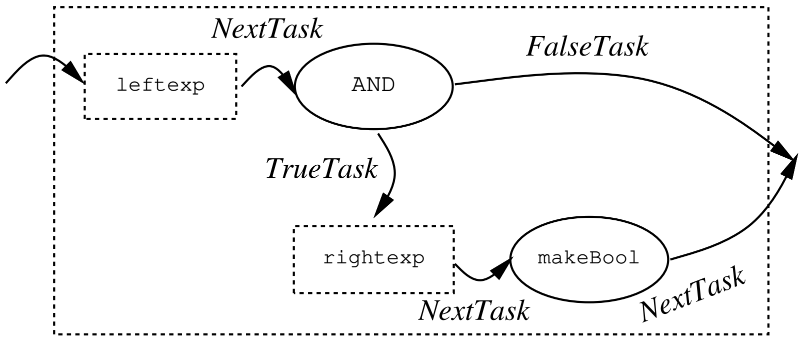


Figure 21: Logical AND expression.

2.10 Assignment Expressions

A simple assignment has the following form:

assignment-expression → *expr1* = *expr2*

where *expr1* and *expr2* are expressions.

```
if TaskType(CurTask) = AND then
    if OnlyValue(CurTask) = 0 then
        ReportValue(0)
        Moveto(FalseTask(CurTask))
    elseif OnlyValue(CurTask) ≠ 0 then
        Moveto(TrueTask(CurTask))
ENDIF
```

Figure 22: Transition rule for *AND* tasks.

To evaluate a simple assignment expression, copy the value of *expr2* into the memory location given by *expr1*, returning that value as the value of the parent expression.

This is the first occurrence of an operator in Algebra Two with an ambiguous evaluation order, as discussed in Sect. 2.5. Thus, the expression is represented in our algebra as in Fig. 14. The transition rule for assignment operators is shown in Fig. 23.

```
if TaskType(CurTask) = simple-assignment then
    EVALUATE OPERANDS WITH
        DoAssign( LeftValue(CurTask),RightValue(CurTask),
                  ValueType(CurTask))
        ReportValue(RightValue(CurTask))
    END EVALUATE
endif
```

Figure 23: Transition rule for simple assignment tasks.

Within C, there are other assignment operators ("`+=`", "`*=`", *etc.*) which perform a mathematic operation on the value of *expr2* and the value stored in the memory location given by *expr1*. The result is copied into the memory location given by *expr1*. For example, "`i *= 2`" has the same value and effect as "`i = i * 2`". However, it is not true in general that "`a` *op*`= b`" can be seen as an abbreviation for "`a = a` *op* `b`", since the expression `a` is evaluated only once in the former expression, but twice in the latter. Since evaluating expressions in C may cause side effects, the distinction is important. Thus, we must present additional transition rules for such operators.

With two exceptions, all these assignment operators have transition rules like that shown in Fig 24 for the multiplicative assignment operator ("`*=`").

The two exceptions are the additive and subtractive assignment operators, "`+=`" and "`-=`". In C, one may add an integer i to a pointer expression p, with the result being a pointer which is i units forward in memory from p. Thus, to process `''a += b''`, we must perform different actions if `a` is a pointer variable. The transition rule for additive assignment involving pointers is shown in Fig 25. The rule for additive assignment involving non-pointers is like that shown above for multiplicative assignments, with the additional condition *PointerType(LeftChild(CurTask)) = false* inserted in the guard of the rule. The rules for subtractive assignment are similar and thus omitted.

```
if TaskType(CurTask) = multiplicative-assignment then
    EVALUATE OPERANDS WITH
        DoAssign( LeftValue(CurTask),
                    MemoryValue[LeftValue(CurTask), ValueType(CurTask)]
                        * RightValue(CurTask),
                    ValueType(CurTask))
        ReportValue(MemoryValue[LeftValue(CurTask), ValueType(CurTask)]
                        * RightValue(CurTask))
    END EVALUATE
endif
```

Figure 24: Transition rule for multiplicative assignment.

```
if TaskType(CurTask) = additive-assignment
            and PointerType(LeftChild(CurTask)) = true then
    EVALUATE OPERANDS WITH
        DoAssign( LeftValue(CurTask),
                    MemoryValue(LeftValue(CurTask), ValueType(CurTask))
                     + Size(PointsToType(CurTask)) * RightValue(CurTask),
                    ValueType(CurTask)
        ReportValue(MemoryValue(LeftValue(CurTask), ValueType(CurTask))
                     + Size(PointsToType(CurTask)) * RightValue(CurTask))
    END EVALUATE
endif
```

Figure 25: Transition rule for additive assignment with pointers.

2.11 The `sizeof` Operator

Expressions involving `sizeof` have one of the following forms:

sizeof-expression $\rightarrow$ `sizeof` *expression*
sizeof-expression $\rightarrow$ `sizeof` (*type-name*)

The *ValueType* function converts either operand into a value in the *typename* universe. With that information, we may use the *Size* function to determine the size, in bytes, of an element of that particular type and return that number as the value of the unary expression. The transition rule for size operators is shown in Fig. 26.

---

if *TaskType(CurTask) = sizeof* **then**
  ReportValue(Size(ValueType(CurTask)))
  Moveto(NextTask(CurTask))
endif

Figure 26: Transition rule for *sizeof* tasks.

2.12 Constants

For constant expressions, the *ConstVal* function returns the appropriate value. (Note that we treat enumerated type values as constants.) The transition rules for constants is shown in Fig. 27.

---

if *TaskType(CurTask) = constant* **then**
  ReportValue(ConstVal(CurTask))
  Moveto(NextTask(CurTask))
endif

Figure 27: Transition rule for *constant* tasks.

2.13 General Mathematic Expressions

There are a large number of mathematic expressions in C involving binary operators ("`*`", "`+`", "`-`", *etc.*) whose behaviors are similar. (We treat the bit-wise operators (*e.g.*, `|`, `&`) as ordinary mathematic operators.) We assume that infix functions corresponding to these C functions are present within our algebra. Thus, to evaluate one of these expressions, evaluate both operand expressions and apply the appropriate function. We present the transition rule for multiplication in Fig.28 as a representative of this category of expressions, and omit the rules for other binary operators of this form for brevity.

Rules for the addition and subtraction operators are slightly more complicated, since one may add or subtract an integer to a pointer. One may add an integer i to a pointer variable p with the result being a pointer which is i units forward in memory from p. Consider, for example, a pointer p to an `int` in a system

if *TaskType(CurTask) = multiplication* **then**
    EVALUATE OPERANDS WITH
        *ReportValue(LeftValue(CurTask) * RightValue(CurTask))*
        Moveto(NextTask(CurTask))
    END EVALUATE
endif

Figure 28: Transition rule for *multiplication* tasks.

where `int`s require 4 bytes in memory. The expression "$p+1$" refers to the location in memory 4 bytes after p.

Similarly, one may subtract an integer i from a pointer variable p, with the result being a pointer i units in memory preceding p. (In our example, "$p-1$" refers to the location in memory 4 bytes before p.) Further, one may subtract two pointers of the same type, resulting in the number of units of memory lying between the two pointers. (Thus, $((p+i)-p) = i$.) In each case, the size of a "unit" of memory is determined by the size of the object type to which the pointer points.

This requires specialized rules for the addition and subtraction operators. As we process each of these operators, it now becomes necessary to know whether or not a given variable is a pointer; a static partial function *PointerType: tasks* $\rightarrow$ {*true, false*} contains this information. For tasks for which *PointerType* returns *true*, a static partial function *PointsToType : tasks* $\rightarrow$ *typename* indicates the object type to which the pointer points.

The transition rules for the addition and subtraction operators are shown in Fig. 29 and Fig. 30.

if *TaskType(CurTask) = addition* **then**
    EVALUATE OPERANDS WITH
        if *PointerType(LeftTask(CurTask)) = true* **then**
            ReportValue(LeftValue(CurTask)
                *+ (Size(PointsToType(CurTask)) * RightValue(CurTask)))*
        elseif *PointerType(RightTask(CurTask)) = true* **then**
            ReportValue(RightValue(CurTask)
                *+ (Size(PointsToType(CurTask)) * LeftValue(CurTask)))*
        else
            ReportValue(LeftValue(CurTask) + RightValue(CurTask))
        endif
        Moveto(NextTask(CurTask))
    END EVALUATE
endif

Figure 29: Transition rule for *addition* tasks.

```
if TaskType(CurTask) = subtraction then
  if PointerType(LeftTask(CurTask)) = true and
        PointerType(RightTask(CurTask)) = true then
     ReportValue((LeftValue(CurTask) - RightValue(CurTask))
            / Size(PointsToType(CurTask)))
  elseif PointerType(LeftTask(CurTask)) = true and
        PointerType(RightTask(CurTask)) = false then
     ReportValue(LeftValue(CurTask)
        - (Size(PointsToType(CurTask)) * RightValue(CurTask)))
  else
     ReportValue(LeftValue(CurTask) - RightValue(CurTask))
  endif
  Moveto(NextTask(CurTask))
endif
```

Figure 30: Transition rule for *subtraction* tasks.

2.14 Mathematical Unary Operators

Unary operator expressions have one of the following forms:

unary-expression → `+` *expression*
unary-expression → `-` *expression*
unary-expression → `~` *expression*
unary-expression → `!` *expression*

Evaluating these expressions takes a form similar to that for binary mathematical operators. We present the transition rule for the negation operator in Fig. 31 as a representative example.

```
if TaskType(CurTask) = negation then
  ReportValue( - OnlyValue(CurTask))
  Moveto(NextTask(CurTask))
endif
```

Figure 31: Transition rule for *negation* tasks.

2.15 Casting Expressions

A casting expression has the following form:

cast-expression → `(` *type-name* `)` *expression*

A static function *CastType: tasks* → *typename* indicates the old type from which the value of the expression is to be cast; *ValueType* indicates the new type into which the value will be cast. A static function *Convert: typename* × *typename* × *values* → *values* converts elements from one universe into the corresponding elements of another universe. For example, *Convert(float,int,X)* is the closest integer to X (assuming X is a floating-point value). Note that the meaning of "closest" is implementation-defined.

To perform a cast, evaluates the argument expression and use the *Convert* function to generate the proper return value. Our task sequence places the expression to be cast before the task which performs the casting; thus, the argument of the cast has been evaluated already and its value is available. The transition rule for casting expressions is shown in Fig. 32.

---

if *TaskType(CurTask) = cast* **then**
    ReportValue(Convert(CastType(CurTask),
                        ValueType(CurTask), OnlyValue(CurTask)))
    Moveto(NextTask(CurTask))
endif

Figure 32: Transition rule for *cast* tasks.

2.16 Pre-Increment and Pre-Decrement

A pre-increment or pre-decrement expression has the following form:

pre-incr-expression $\rightarrow$ ++ *expression*
pre-decr-expression $\rightarrow$ -- *expression*

To evaluate a pre-increment (resp. pre-decrement) expression, increment (decrement) the value stored at the indicated memory location by one and store the new value into that memory location; the incremented (decremented) value is the value of the parent expression. Note that the expression to be modified may be a pointer; in this case, the value in memory is incremented (decremented) by the size of the object to which the pointer points (as with normal pointer addition and subtraction). The transition rule for pre-increment expressions is shown in Fig 33. The transition rules for pre-decrement expressions are similar to those presented here and thus omitted.

2.17 Post-Increment and Post-Decrement

A post-increment or post-decrement expression has the following form:

postincr-expression $\rightarrow$ *expression* ++
postdecr-expression $\rightarrow$ *expression* --

Post-increment (resp. post-decrement) operators are handled in the same manner as pre-increment (pre-decrement) operators except that the sequence of operations is reversed: *i.e.*, the value of the parent expression is established before the incrementing (decrementing) takes place. The transition rule for the post-increment operator is shown in Fig. 34. (As before, the transition rules for the post-decrement operator are similar and thus omitted.)

```
if TaskType(CurTask) = pre-increment then
    if PointerType(CurTask) = true then
        DoAssign(OnlyValue(CurTask),
            MemoryValue(OnlyValue(CurTask), ValueType(CurTask))
                + Size(PointsToType(CurTask)),
            ValueType(CurTask))
        ReportValue(   MemoryValue( OnlyValue(CurTask),
                                    ValueType(CurTask))
                       + Size(PointsToType(CurTask)))
    else
        DoAssign(OnlyValue(CurTask),
            MemoryValue(OnlyValue(CurTask), ValueType(CurTask)) + 1,
            ValueType(CurTask))
        ReportValue(MemoryValue(   OnlyValue(CurTask),
                                   ValueType(CurTask))+1)
ENDIF
```

Figure 33: Transition rule for *pre-increment* tasks.

```
if TaskType(CurTask) = post-increment then
    if PointerType(CurTask) = true then
        DoAssign(OnlyValue(CurTask),
            MemoryValue(OnlyValue(CurTask), ValueType(CurTask))
                + Size(PointsToType(CurTask)),
            ValueType(CurTask))
    else
        DoAssign(OnlyValue(CurTask),
            MemoryValue(OnlyValue(CurTask), ValueType(CurTask))+1,
            ValueType(CurTask))
    endif
    ReportValue(MemoryValue(   OnlyValue(CurTask),
                               ValueType(CurTask)))
endif
```

Figure 34: Transition rule for *post-increment* tasks.

2.18 Addresses

An addressing expression has the following form:

addressing-expression $\rightarrow$ `&` *expr1*

where *expr1* is an expression.

The `&` operator in C passes back as its result the address of the memory location indicated by the argument expression.

As we evaluate expressions which refer to objects in memory, we need to know whether we need to use the address of an object or the object itself in our calculations. (For example, in the assignment statement "`a = b;`", the address or *lvalue* of variable `a` is needed, but the object being referenced or *rvalue* of variable `b` is needed.) A static partial function *ValueMode: tasks* $\rightarrow$ {*lvalue, rvalue*} indicates which of the two pieces of information should be computed for a given task.

We assert that *ValueMode(e) = lvalue* for the sub-expressions of *expr1*; thus, the value returned through evaluation of the argument expression is the address (and not the value) of the argument expression in memory. We simply pass this address up the task graph. The resulting simple transition rule for the addressing operator is shown in Fig. 35.

---

if *TaskType(CurTask) = address* **then**
    ReportValue(OnlyValue(CurTask))
    Moveto(NextTask(CurTask))
endif

Figure 35: Transition rule for *address* tasks.

2.19 De-Referencing

A de-referencing expression has the following form:

de-reference-expression $\rightarrow$ `*` *expression*

If the parent expression is an rvalue, evaluate the argument and use the *Memory* function to return the value stored in memory at the indicated location. Otherwise, return the address indicated by the argument (since the expression is an lvalue and requires that a pointer be returned to the parent expression). The transition rule for de-referencing is shown in Fig. 36.

2.20 Array References

An array reference has the following form:

array-ref-expression $\rightarrow$ *expr1* `[` *expr2* `]`

where *expr1* and *expr2* are expressions.

According to [KR], an array reference of the form `a[b]` is identical, by definition, to the expression `*((a)+(b))`.[3] This definition is valid because the name of an array in C may be used as a pointer to the first element of the array.

[3] Note that this means that `a[b]` and `b[a]` evaluate to the same value.

```
if TaskType(CurTask) = de-referencing then
    if ValueMode(CurTask) = rvalue then
        ReportValue(MemoryValue(  OnlyValue(CurTask),
                                  ValueType(CurTask)))
    elseif ValueMode(CurTask) = lvalue then
        ReportValue(OnlyValue(CurTask))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 36: Transition rule for *de-referencing* tasks.

We assert that any array references present in the program being modeled in our algebra are represented as an expression of equivalent form involving addition and de-referencing. Thus, we do not need to present any additional rules to handle array references.

One may prefer to think of arrays as objects in their own right, and present an algebra intermediate to Algebras One and Two where expressions like `a[b]` could be treated at a higher level of abstraction. While such a presentation is possible, we choose not to do so here.

2.21 Function Invocations

A function invocation has the following form:

func-invocation-expression $\rightarrow$ *expression* (*expression-list*)

Since we have disallowed function invocations for the moment, we will obtain the value of a function invocation expression from an external function *FunctionValue: tasks* $\rightarrow$ *results*. The transition rule for function invocations is shown in Fig. 37.

```
if TaskType(CurTask) = function-invocation then
    ReportValue(FunctionValue(CurTask))
    Moveto(NextTask(CurTask))
endif
```

Figure 37: Transition rule for *function-invocation* tasks.

2.22 Identifiers

An external function *FindID: tasks* $\rightarrow$ *addresses* maps identifier expression tasks to the corresponding memory location used by the associated variable. (In Algebra Three we shall eliminate the use of this function.) Thus, to handle an identifier expression, one returns the appropriate address or value from memory, as specified by the *ValueMode* function. The transition rule for identifiers is shown in Fig. 38.

```
if TaskType(CurTask) = identifier then
    if ValueMode(CurTask) = lvalue then
        ReportValue(FindID(CurTask))
    elseif ValueMode(CurTask) = rvalue then
        ReportValue(MemoryValue(   FindID(CurTask),
                                   ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 38: Transition rule for *identifier* tasks.

2.23 `struct` or `union` References

A `struct` or `union` reference has the following form:

struct-expression → *expr1* . *identifier*

where *expr1* is an expression evaluating to a `struct` or `union`. There is also another form:

struct-expression → *expr2* `->` *identifier*

where *expr2* is an expression evaluating to a pointer to a `struct` or `union`. Expressions of the form "`a->b`" are equivalent to those of the form "`(*a).b`". Thus, we will only consider references of the form "`a.b`", asserting that references of the other form are represented using their equivalent expansions.

The *ConstVal* function applied to the `struct` reference task returns the offset in memory to be used in obtaining the address or value of the specified field of the structure. The transition rule for struct references is shown in Fig. 39.

```
if TaskType(CurTask) = struct-reference then
    if ValueMode(CurTask) = lvalue then
        ReportValue(OnlyValue(CurTask) + ConstVal(CurTask))
    elseif ValueMode(CurTask) = rvalue then
        ReportValue(MemoryValue(OnlyValue(CurTask)
                                + ConstVal(CurTask),
                         ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 39: Transition rule for *struct-reference* tasks.

2.24 Bit Fields

Bit fields are members of `struct`s which use a user-specified number of bits for their representations. Bit fields are used to minimize the space used by a `struct` or to represent accurately input or output values with bit-level significance. Much about bit fields behave is implementation-dependent: *e.g.* how bit fields are packed into adjacent bytes, whether or not unnamed “holes” will appear in `structs` between bit fields, or whether bit fields are read left-to-right or right-to-left. As a rule, operaptions are done with bytes (even bit operations); we show how to handle bit-fields in a byte-based model of memory.

An example of a `struct` using bit fields is shown in Fig. 40. Here, the field `b` holds a 4-bit unsigned integer and `c` holds a 15-bit unsigned integer. Since bytes usually comprise 8 bits, `c` will probably lie in two or three consecutive bytes in memory, possibly sharing a byte with `b`.

```
struct {
   unsigned int a;
   unsigned int b:4;
   unsigned int c:15;
   } bitty;
```

Figure 40: Example of a `struct` using bit fields.

Suppose that we want to execute `bitty.c = 12`. We need to obtain the bytes which hold `c`'s bits and modify those bits accordingly while leaving all other bits unchanged. This gives rise to a static function *BitAssign: results* $\times$ *typename* $\times$ *results* $\rightarrow$ *results* which indicates the change occurring in the value of the appropriate collection of contiguous bytes when a bit field is modified. Given the content *oldval* of an appropriate piece of memory, *BitAssign(oldval, bittype, 12)* returns the new value; here *bittype* is the type of `c` in `bitty`.

Assigning to bit-fields is thus slightly different than usual. We present the transition rule for simple assignments to bit fields in Fig. 41.

---

if *TaskType(CurTask) = bit-assignment* **then**
    EVALUATE OPERANDS WITH
        DoAssign(LeftValue(CurTask),
            BitAssign(MemoryValue(LeftValue(CurTask),
                            ValueType(CurTask)),
                    ValueType(CurTask),
                    RightValue(CurTask)),
            ValueType(CurTask))
        ReportValue(RightValue(CurTask))
    END EVALUATE
endif

Figure 41: Transition rule for *bit-assignment* tasks.

Evaluating bit-field references is also slightly different, since we need to extract the value of the bit field from the (usually) larger enclosing value. A static function *BitExtract: results* $\times$ *typename* $\rightarrow$ *results* performs this extraction. A static partial function *BitType: tasks* $\rightarrow$ *typename* indicates the type of bit field being references in such situations.

The transition rule for structure bit-field references is shown in Fig. 42.

```
if TaskType(CurTask) = struct-bit-reference then
    if ValueMode(CurTask) = lvalue then
        ReportValue(OnlyValue(CurTask) + ConstVal(CurTask))
    elseif ValueMode(CurTask) = rvalue then
        ReportValue(BitExtract(
                MemoryValue(OnlyValue(CurTask) + ConstVal(CurTask),
                BitType(CurTask)),
            ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 42: Transition rule for *struct-bit-reference* tasks.

3 Algebra Three: Allocating and Initializing Memory

Algebra Three refines Algebra Two and focuses on memory allocation and initialization of variables.

3.1 Declarations

We represent declarations in C as elements of the *tasks* universe, linked in the proper order with statement tasks by *NextTask*.

C distinguishes between so-called *static variables* and other variables. The difference between static and non-static variables arises when control is passed to the declaration task for a variable. If the variable is not static, new memory is always allocated to the variable and its initializing expression (if it exists) is evaluated with the value of the expression being assigned to the new memory location. If the variable is static, the above allocation and initialization is performed only the first time that the declaration is executed; should the declaration become the focus of control once again, the same memory segment is allotted to the variable.

A static partial function *DecType: tasks* $\rightarrow$ {*static, non-static*} indicates what type of variable is being declared. (Note that there are also `extern` variable declarations in C which do not reserve memory but serve as syntactic linkage between variables. We omit consideration of such declarations since their function is wholly syntactic in nature.) A static partial function *Initializer: tasks* $\rightarrow$ *tasks* indicates the appropriate initializing expression (if any). We will store the value of the initializing expression using *RightValue*.

A partial function *StaticAddr: tasks* $\rightarrow$ *addresses* stores the current address (if any) that has been assigned to a static variable. In Algebra Three, *StaticAddr* is not really needed, since the *OnlyValue* function would provide the proper storage for the address of the static variable. However, it will simplify other rules to be presented. An external function *NewMemory: tasks* $\rightarrow$ *addresses* returns an address in memory to be used for the given declaration task. The transition rules for declarations are shown in Fig. 43 and Fig. 44.

3.2 Automatic Variables and Non-Local Jumps

Automatic (or local) variables are allocated memory not only when a block is entered normally, but also if a non-local `goto` statement transfers control into a block. As an example, consider the following code fragment:

```
if TaskType(CurTask) = declaration and DecType(CurTask) = static then
    if StaticAddr(CurTask) ≠ undef then
        OnlyValue(CurTask) := StaticAddr(CurTask)
        Moveto(NextTask(CurTask))
    elseif StaticAddr(CurTask) = undef then
        if Initializer(CurTask) ≠ undef and
            RightValue(CurTask) = undef then
            Moveto(Initializer(CurTask))
        else
            OnlyValue(CurTask) := NewMemory(CurTask)
            StaticAddr(CurTask) := NewMemory(CurTask)
            if Initializer(CurTask) ≠ undef then
                DoAssign(NewMemory(CurTask),
                    RightValue(CurTask), ValueType(CurTask))
            else
                Moveto(NextTask(CurTask))
ENDIF
```

Figure 43: Transition rule for static declarations.

```
if TaskType(CurTask) = declaration and DecType(CurTask) ≠ static then
    if Initializer(CurTask) ≠ undef and
        RightValue(CurTask) = undef then
        Moveto(Initializer(CurTask))
    else
        OnlyValue(CurTask) := NewMemory(CurTask)
        if Initializer(CurTask) ≠ undef then
            DoAssign(NewMemory(CurTask),
                RightValue(CurTask), ValueType(CurTask))
        else
            Moveto(NextTask(CurTask))
ENDIF
```

Figure 44: Transition rule for non-static declarations.

```
int flag;                    for (;;) {
  ⋮                             int local = 1;
if (flag) goto target;          printf("%i" ,local);
  ⋮                             target:  local = 0;
                                printf("%i" ,local);
                             }
```

In this fragment, if the `for` statement is entered normally, variable `local` will be allocated memory and initialized to 1. If the `goto` statement is executed, variable `local` should be allocated memory before the statement labeled `target` is executed, although no initialization is performed.

Our rules for constructing task graphs only provide for a unique moment when a given variable may be allocated. Consequently, we introduce a new task type *indirect-declaration*. Previously, *NextTask* mapped each *jump* task directly to its target. Now, we put an *indirect-declaration* task for each local variable between the *jump* task and its target. The *Decl* function will map each new task to the original declaration of that variable. The rule for handling indirect declarations (shown in Fig. 45) is pretty obvious.

---

if *TaskType(CurTask) = indirect-declaration* **then**
    OnlyValue(Decl(CurTask)) := NewMemory(Decl(CurTask))
    Moveto(NextTask(CurTask))
endif

Figure 45: Transition rule for indirect declarations.

3.3 Revision: Identifiers

A static partial function *Decl: tasks* → *tasks* maps tasks corresponding to occurrences of an identifier to the task corresponding to the declaration task for that variable. The revised rule for identifiers (generated by replacing each previous occurrence of *FindID()* with *OnlyValue(Decl())*) is shown in Fig. 46.

---

if *TaskType(CurTask) = identifier* **then**
    if *ValueMode(CurTask) = lvalue* **then**
        ReportValue(OnlyValue(Decl(CurTask)))
    elseif *ValueMode(CurTask) = rvalue* **then**
        ReportValue(MemoryValue(OnlyValue(Decl(CurTask)),
                        ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif

Figure 46: Revised transition rule for *identifier* tasks.

3.4 Initializers

Initializers in C come in two forms: expressions (for variables of the basic types) and lists of expressions (for variables representing arrays and structures).

Our previous rules for evaluating expressions will handle initializers for simple expressions. To assist in handling aggregate expressions, a static function *AddTo: typename* $\times$ *results* $\times$ *results* $\rightarrow$ *results* appends a value onto the end of an aggregate structure of the specified type. (For example, if $[42, 8]$ is an integer array, then $AddTo(array, [42, 8], 4) = [42, 8, 4]$.)

We illustrate how expressions with aggregate initializers (*i.e.* expressions whose initializer is an expression list) are represented in our algebra by the graph in Fig. 47. Our previous transition rules will insure that each expression in the initializer list will be evaluated; we need to provide rules that combine the results of these evaluations into the proper aggregate value.

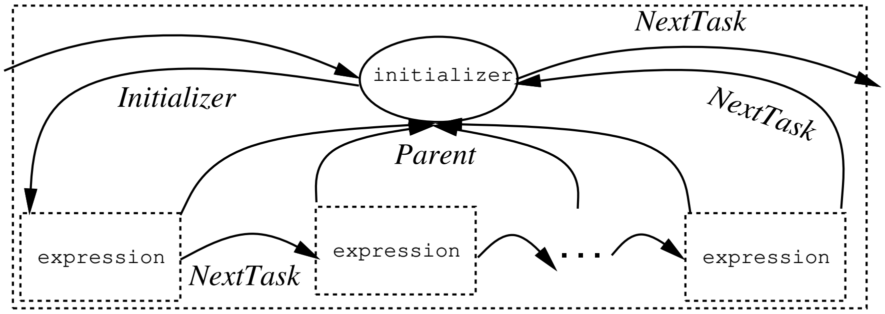


Figure 47: Aggregate initializer.

The *WhichChild* function returns the value *aggregate* when the expression being evaluated is a component of an aggregate initializer. We extend our *ReportValue* macro as shown in Fig. 48 to correctly combine aggregate expressions.

---

if *WhichChild(CurTask) = aggregate* **then**
    OnlyValue(Parent(CurTask)) :=
        AddTo(ValueType(Parent(CurTask)),
            Value(Parent(CurTask)), value)
endif

Figure 48: Extension of the *ReportValue* macro.

3.5 Initial State

We assert that initially, *CurTask* indicates the first declaration task for the program, or the first task of the first statement of the program if no declaration tasks exist.

4 Algebra Four: Handling Function Definitions

Algebra Four revises Algebra Three and focuses on C function definitions. In this way, we also (implicitly) present rules for starting a C program, since the starting function `main` is an ordinary C function (with externally provided parameters).

(There are also function declarations in C, which are used to specify syntactic information. Since their purpose is wholly syntactic in nature, we ignore them.)

4.1 Modeling The Stack

C functions may have several active incarnations at a given moment. Thus, we must have some means for storing multiple values of a function for a given task.

The universe *stack* comprises the positive integers, with a distinguished element *StackRoot = 1*. Static functions *StackPrev: stack* $\rightarrow$ *stack* and *StackNext: stack* $\rightarrow$ *stack* are the predecessor and successor functions on the positive integers. A dynamic distinguished element *StackTop* indicates the current top of the stack.

To store state-associated information on the stack, we modify the various *Value* functions *LeftValue, RightValue, OnlyValue,* and *TestValue* to be binary functions from *tasks* $\times$ *stack* to *results*. This requires us to rewrite almost every rule that has appeared previously; we simplify matters by stating that every previous reference to *V(X)* should be replaced by *V(X, StackTop)*, where *V* is one of the *Value* functions listed above. Similarly, we modify *Visited* to be a binary fruntion from *tasks* $\times$ *stack* to *tags*, replacing all previous occurrences of *Visited(X)* with *Visited(X, StackTop)*.

4.2 Function Invocations: Caller's Story

A function invocation has the following form:

func-invocation $\rightarrow$ *func-name* (*expression-list*)

In Algebra Two we used an external function *FunctionValue* to obtain the value of a function invocation. Here, we eliminate the use of this function.

The name of a function is often an identifier, but in general it is an expression referring to the address of the function. (What resides in memory at that address is implementation-dependent.) A static partial function *AddrToFunc: addresses* $\rightarrow$ *tasks* maps function addresses to the first task of the function definition.

While processing a function invocation, we wish to copy the value[4] of each parameter to an appropriate place for the callee to process. The *Parent* function (utilized by our *ReportValue* macro) maps each argument-expression task to its corresponding function parameter task. A partial function *ParamValue: tasks* $\times$ *stack* $\rightarrow$ *results* indicates the values of parameters being passed. We append the rule shown in Fig. 49 to the *ReportValue* macro.

if *WhichChild(CurTask) = param* **then**
    ParamValue(Parent(CurTask),StackNext(StackTop)) := value
endif

Figure 49: Extension of the *ReportValue* macro.

We need to store the current task in order to resume execution at this point after the callee has finished. A dynamic function *ReturnTask: stack* $\rightarrow$ *tasks* indicates the new value of *CurTask* when the execution of the current function terminates. We assert that when *CurTask = 1*, *ReturnTask(CurTask) = undef*, which will cause the algebra to terminate when the top-level function terminates.

To process a function invocation, evaluate the name of the function along with all of the arguments in the expression list, and then transfer control to the specified function. At the same time, "push another

[4] In C, all function parameters are call-by-value.

frame onto the stack"; *i.e.*, increment *StackTop* by 1. When control returns from the function, the stack value will be "popped" (*i.e.*, decremented by 1) and the function's return value will be passed to the parent expression.

As with the operands to most arithmetic operators, the ANSI standard [KR] does not specify the order in which arguments to a function are evaluated. We thus must present specialized rules for evaluating the expressions associated with a function invocation. The external function *ChooseTask* will indicate at each moment which expression associated with a function invocation should be evaluated next. Thus, our transition rules will simply make repeated calls to *ChooseTask* until all expressions have been evaluated, which will occur when *ChooseTask* returns *undef*. The transition rule for function invocation is shown in Fig. 50.

---

if *TaskType(CurTask) = function-invocation* **then**
    if *ChooseTask(CurTask)* $\neq$ *undef* **then**
        Moveto(ChooseTask(CurTask))
    elseif *ChooseTask(CurTask) = undef* **then**
        if *OnlyValue(CurTask,StackTop) = undef* **then**
            StackTop := StackNext(StackTop)
            ReturnTask(StackNext(StackTop)) := CurTask
            Moveto(AddrToFunc(LeftValue(CurTask,StackTop)))
        elseif *OnlyValue(CurTask,StackTop)* $\neq$ *undef* **then**
            ReportValue(OnlyValue(CurTask,StackTop))
            Moveto(NextTask(CurTask))
ENDIF

Figure 50: Revised transition rule for *function-invocation* tasks.

4.3 Function Invocations: Callee's Story

A function definition in C consists of a list of parameter declarations and a compound statement. To process a parameter declaration, we allocate new memory for each parameter and assign the appropriate value (stored here by the function invocation transition rules) to that new memory location. The transition rule for parameter declarations is shown in Fig. 51.

---

if *TaskType(CurTask) = parameter-declaration* **then**
    DoAssign(NewMemory(CurTask),
        ParamValue(CurTask,StackTop), ValueType(CurTask))
    OnlyValue(CurTask,StackTop) := NewMemory(CurTask)
endif

Figure 51: Transition rule for *parameter-declaration* tasks.

A `return` statement has one of the following forms:

return-statement → **return** ;
return-statement → **return** *expression* ;

If an expression is present, copy the value of the expression to the task which invoked the current function (as indicated by *ReturnTask* and *StackPrev*). Whether or not an expression is present, return control to the invoking task. The transition rule for `return` statements is shown in Fig. 52.

---

if *TaskType(CurTask) = return* **then**
  OnlyValue(ReturnTask(StackTop),StackPrev(StackTop)) :=
    OnlyValue(CurTask,StackTop)
  StackTop := StackPrev(StackTop)
  Moveto(ReturnTask(StackTop))
endif

Figure 52: Transition rule for *return* tasks.

If a `return` statement is not explicitly present at the end of a function, our algebra will still contain a *return* task as the last task of the function, as if the statement "`return ;`" was present as the last statement of the original C function.

4.4 Global Variables

Since function definitions may not contain other function definitions, a given variable identifier refers either to a variable local to the current function or to a variable declared outside any function. A static partial function *GlobalVar: tasks* → *Bool* indicates whether or not a given identifier refers to a global variable. We present the modified transition rule for identifiers in Fig. 53.

4.5 Initial State

We assert that initially, *CurTask* indicates the first declaration task for the global variables of the program, or the first task of the first statement `main()` if no declaration tasks exist.

if *TaskType(CurTask) = identifier* **then**
    if *ValueMode(CurTask) = lvalue* **then**
        if *GlobalVar(CurTask) = true* **then**
            ReportValue(OnlyValue(Decl(CurTask),StackRoot))
        elseif *GlobalVar(CurTask) = false* **then**
            ReportValue(OnlyValue(Decl(CurTask),StackTop))
        endif
    elseif *ValueMode(CurTask) = rvalue* **then**
        if *GlobalVar(CurTask) = true* **then**
            ReportValue(MemoryValue(OnlyValue
                (Decl(CurTask),StackRoot), ValueType(CurTask)))
        elseif *GlobalVar(CurTask) = false* **then**
            ReportValue(MemoryValue(OnlyValue
                (Decl(CurTask),StackTop), ValueType(CurTask)))
        endif
    endif
    Moveto(NextTask(CurTask))
endif

Figure 53: Revised transition rule for identifiers.

References

[ASU] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools", Addison-Wesley, 1988.

[Gu] Yuri Gurevich, "Evolving Algebras: An Introductory Tutorial", Bulletin of European Assocation for Theoretical Computer Science, February 1991. (A slightly updated version will appear in the EATCS Book of Columns, World Scientific Publishers.)

[GH1] Yuri Gurevich and James K. Huggins, "The Evolving Algebra Semantics of C: Preliminary Version", CSE-TR-141-92, EECS Department, University of Michigan, 1992.

[GH2] Yuri Gurevich and James K. Huggins, "The Semantics of the C Programming Language", Selected papers from *CSL'92* (Computer Science Logic), Springer Lecture Notes in Computer Science 702, 1993, 274–308.

[GH3] Yuri Gurevich and James K. Huggins, "Errata to 'The Semantics of the C Programming Language' ", Selected papers from *CSL'93*, Springer Lecture Notes in Computer Science, to appear.

[KR] Brian W. Kernighan and Dennis M. Ritchie, "The C Programming Language", 2nd edition, Prentice Hall, 1988.

Index

Evolving Algebras and Partial Evaluation*

Yuri Gurevich†and James K. Huggins†

July 21, 1995

Keyword Codes: D.2.2; D.2.m; F.3.2
Keywords: Software Engineering, Tools and Techniques;
Software Engineering, Miscellaneous;
Logics and Meanings of Programs, Semantics of Programming Languages

Abstract

We describe an automated partial evaluator for evolving algebras implemented at the University of Michigan.

1 Introduction to Sequential Evolving Algebras

A fuller discussion of evolving algebras (or *ealgebras*) can be found in [1]; to make this paper self-contained, we recall briefly the main concepts.

A sequential ealgebra $\mathcal{A}$ is an abstract machine. The *signature* of $\mathcal{A}$ is a (finite) collection of function names, each of a fixed arity. A state of $\mathcal{A}$ is a set, the *superuniverse*, together with interpretations of the function names in the signature. These interpretations are called *basic functions* of the state. The superuniverse does not change as $\mathcal{A}$ evolves; the basic functions may.

Formally, a basic function of arity r (*i.e.* the interpretation of a function name of arity r) is an r-ary operation on the superuniverse. (We often use basic functions with $r = 0$; such basic functions will be called *distinguished elements*.) But functions naturally arising in applications may be defined only on a part of the superuniverse. Such partial functions are represented by total functions in the following manner.

The superuniverse contains distinct elements *true, false, undef* which allow us to deal with relations (viewed as binary functions with values *true* or *false*) and partial functions (where $f(\overline{a}) = undef$ means f is undefined at the tuple $\overline{a}$). These three elements are *logical constants*. Their names do not appear in the signature; this is similar to the situation in first-order logic with equality where equality is a logical constant and the sign of equality does not appear in the signature. In fact, we use equality as a logical constant as well.

A *universe* U is a special type of basic function: a unary relation usually identified with the set $\{x : U(x)\}$. The universe $Bool = \{true, false\}$ is another logical constant. When we speak about a function f from a universe U to a universe V, we mean that formally f is a unary operation on the superuniverse such that $f(a) \in V$ for all $a \in U$ and $f(a) = undef$ otherwise. We use self-explanatory notations like $f : U \to V$, $f : U_1 \times U_2 \to V$, and $f : V$. The last means that the distinguished element f belongs to V.

In principle, a program of $\mathcal{A}$ is a finite collection of transition rules of the form

if t_0 **then** $f(t_1, \ldots, t_r) := t_{r+1}$ **endif** (1)

*In *Proceedings of IFIP Congress 94 – Volume 1*, eds. B. Pehrson and I. Simon, Elsevier, 1994.

†Partially supported by ONR grant N00014-91-J-1861 and NSF grant CCR-92-04742. EECS Department, University of Michigan, Ann Arbor, MI, 48109-2122, USA. gurevich@umich.edu, huggins@umich.edu

where t_0, $f(t_1,\ldots,t_r)$, and t_{r+1} are closed terms (*i.e.* terms containing no free variables) in the signature of $\mathcal{A}$. An example of such a term is $g(h_1,h_2)$ where g is binary and h_1 and h_2 are zero-ary. The meaning of the rule shown above is this: Evaluate all the terms t_i in the given state; if t_0 evaluates to *true* then change the value of the basic function f at the value of the tuple $(t_1,..,t_r)$ to the value of t_{r+1}, otherwise do nothing.

In fact, rules are defined in a slightly more liberal way; if k is a natural number, $b_0,\ldots,b_k$ are terms and $C_0,\ldots,C_k$ are sets of rules then the following is a rule:

if b_0 **then** C_0
elseif b_1 **then** C_1
  $\vdots$ (2)
elseif b_k **then** C_k
endif

In the case that $b_k = true$, the last line may be abbreviated by "**else** C_k".

Since the C_i are sets of rules, nested transition rules are allowed (and occur frequently).

A program is a set of rules. It is easy to transform a program to an equivalent program comprising only rules of the stricter form (1). We use rules of the more liberal form (2), as well as macros (textual abbreviations), for brevity.

How does $\mathcal{A}$ evolve from one state to another? In a given state, the demon (or interpreter) evaluates all the relevant terms and then makes all the necessary updates. If several updates contradict each other (trying to assign different values to the same basic function at the same place), then the demon chooses nondeterministically one of those updates to execute.

We call a function (name) f *dynamic* if an assignment of the form $f(t_1,\ldots,t_r) := t_0$ appears anywhere in the transition rules. Functions which are not dynamic are called *static*. To allow our algebras to interact conveniently with the outside world, we also make use of *external* functions within our algebra. External functions are syntactically static (that is, never changed by rules), but have their values determined by a dynamic oracle. Thus, an external function may have different values for the same arguments as the algebra evolves.

1.1 Why Partial Evaluation and Evolving Algebras?

One of the main application areas of ealgebras has been programming language semantics. One may view an ealgebra A for a language L as an abstract machine which acts as an interpreter for L. With an L-program p as an input, A gives semantics for p. However, A may be large; for many programs, an ealgebra tailored directly to p is clearer than A. Partial evaluation provides an automated means for tailoring an ealgebra for an L-program p by specializing an ealgebra interpreter for L with respect to p. These tailored ealgebras may not be as good as hand-tailored ones, but they may provide a useful beginning for tailored ealgebras. Of course, this is only one use of a partial evaluator for ealgebras.

2 Partial Evaluation Techniques

Suppose one has a program p and knows a portion of its input ahead of time. Can one take advantage of this information to transform p into a more efficient program? *Partial evaluation* is the process of transforming such a program p into a program p' which, when supplied with the remainder of p's input, has the same behavior as p.

Our partial evaluator follows the "mix" methodology (described in more detail in [3]) and has three phases: binding-time analysis, polyvariant mixed computation, and post-processing optimizations. We describe each of these phases below.

2.1 Binding-Time Analysis

Initially, the partial evaluator is given the names of the basic functions of the ealgebra which will be known ahead of time. During *binding-time analysis*, the partial evaluator determines which basic functions can be pre-computed in the next phase. This process is called binding-time analysis because it determines at what time the value(s) of a basic function can be determined (*i.e.* bound to known values).

The input to this phase is a division of the basic functions which supply input to the ealgebra into two sets: *positive* functions whose values will be known ahead of time, and *negative* functions whose values will not be known until later. The partial evaluator proceeds to classify all basic functions (including those not initially marked by the user) as positive or negative. ([3] use the terms "static" and "dynamic" to refer to these types of values; these terms have different meanings within the ealgebra paradigm.)

In the current implementation, the following algorithm is used to classify a function f as positive or negative:

- If f is syntactically static (that is, not updated by any transition rule), f remains as classified initially by the user.
- If an update $f(\bar{t}) := t_0$ exists in p such that $\bar{t}$ or t_0 references a negative function, f is negative. (Note that even if f was declared as positive by the user, f may still depend on other negative functions and must be classified as negative.)
- If for all updates $f(\bar{t}) := t_0$ in p, every function referenced in $\bar{t}$ and t_0 is positive, and f is not already negative, f is positive.

This classification algorithm is repeatedly applied until a fixed-point is reached. Any remaining unclassified functions are classified as negative and the algorithm is repeated to ensure consistency.

An interesting problem which this algorithm does not handle is the problem of circular dependencies. A basic function f is *self-referential* if some update $f(\bar{t}) := t_0$ within the ealgebra being specialized contains a reference to f within $\bar{t}$ or t_0. The above algorithm classifiess every self-referential function as negative. Often this is appropriate, as some self-referential functions can grow unboundedly. But at times, classifying such functions as positive is also appropriate. Consider the following program:

if *Num* $>$ *0* **then** *Num* := *Num* + *1* **endif**
if *MyList* $\neq$ *Nil* **then** *MyList* := *Tail(MyList)* **endif**

Suppose the initial values of *Num* and *MyList* are known. *Num* should not be classified as a positive function, since it would lead the specializer in the next stage into an infinite loop, as larger and larger values of *Num* would be computed as positive information. On the other hand, there is no problem with classifying *MyList* as positive, since *MyList* will eventually be reduced to *Nil* and remain at that value forever. An addition to the algorithm presented above properly classifies self-referential functions as positive if they are dependent only upon themselves in a bounded manner (as seen here).

The problem of circular dependencies is much more general that the problem of self-reference; it may be that several functions form a mutual dependency cycle. We intend to incorporate a more sophisticated algorithm for binding-time analysis based on an examination of the dependency graph formed by the basic functions of the algebra. In the future, we hope to extend this analysis to parts of basic functions; it may be that $f(\bar{t})$ could be classified as positive for certain tuples $\bar{t}$ but not for others.

2.2 Polyvariant Mixed Computation

After binding time analysis, the partial evaluator begins the process of specializing the input program, executing rules which depend only on positive information (that is, functions classified as positive by our binding-time analysis) and generating code for rules which depend on negative information. The process is called *polyvariant mixed computation*: "mixed" because the processes of executing positive rules and generating code for negative rules is interleaved, and "polyvariant" because the entire program is considered multiple times for different sets of positive information.

The signature τ of the algebra has been divided into two components during binding time analysis: a positive signature τ_+ and a negative signature τ_-. This leads to a corresponding division of states (or structures) S into structures S_+ and S_-. The partial evaluator creates an ealgebra with signature $\tau_- \cup \{K\}$, where K is a nullary function which will be used to hold the positive (or "known") information formerly stored by functions in τ_+.

From a given positive state S_+, the partial evaluator produces rules of the form

$$\textbf{if } K = S_+ \textbf{ then } \mathit{rules} \textbf{ endif}$$

where *rules* is a specialized version of the rules of the entire input program with respect to S_+, along with an assignment to K. Call a transition rule of this form a *K-rule*, whose *guard* is *(*$K = S_+$*)* and whose *body* is *(rules)*. Not that no two K-rules produced by our partial evaluator will have the same guard. We recursively describe how transition rules are speciailized with respect to a given positive state S_+ below.

An expression is specialized with respect to S_+ by substituting all known values of functions in S_+ into the given expression, simplifying when possible.

An update $f(\bar{t}) := t_0$ is specialized with respect to S_+ as follows. If $f \in \tau_+$, no rule is generated. Instead, the change to the positive function f is noted internally in order to generate the correct assignment to K. (Note that in this case, all functions named in $\bar{t}$ and t_0 are positive as a result of our binding time analysis.) Otherwise, an assignment to f is generated, with the values of t_0 and $\bar{t}$ specialized as much as possible using the information in S_+.

A set of rules is specialized with respect to S_+ by specializing each rule and combining the information needed to create a single assignment to K.

A guarded rule "**if** *guard* **then** R_1 **else** R_2 **endif**" is specialized with respect to S_+ as follows. If all functions in *guard* are positive, the result is the specialization of R_1 or R_2, depending on whether the value of *guard* is true or false in S_+. Otherwise, an **if** statement is generated, with *guard*, R_1, and R_2 specialized as above. A guarded rule containing **elseif** clauses is converted to an equivalent form without **elseif** clauses and specialized as above.

2.3 Optimization

The above transformations create a specialized version of the original program. Often, this specialized version contains many unneeded rules, such as:

$$\textbf{if } K = \mathit{foo} \textbf{ then } K := \mathit{bar} \textbf{ endif}$$

Such a K-rule can be deleted; just replace all references to *foo* in the program with references to *bar*. The partial evaluator performs several such optimizations on the specialized program:

- Eliminating terms which serve only as aliases for other terms or constants.
- Combining K-rules with identical bodies.
- Combining K-rules which are executed consecutively but whose bodies have independent updates that could be executed simultaneously without altering the meaning of the program.
- Eliminating K-rules which will never be executed.

These optimizations generate code which is equivalent, but textually shorter and usually requires fewer moves to execute. Of course, one pays a small price in time in order to generate these optimizations.

2.4 Results

It is important that a partial evaluator actually perform useful work. Kleene's s-m-n theorem shows that partial evaluators can in principle be constructed; his proof shows that such evaluators may not necessarily produce output that is more efficient than the original. One can specialize an ealgebra, for example, by creating two K-rules: one which initializes the functions in S_+ to their initial values and one which has the original unspecialized program as its body. This "specialized" ealgebra has the same behavior as the original, but is hardly more useful than the original algebra.

[3] suggest a standard for evaluating partial evaluators. Consider a self-interpreter for a language: that is, an interpreter for a language L written itself in L. (McCarthy's original description of a LISP interpreter written itself in LISP is such an interpreter.) Specializing such an interpreter with respect to an L-program p should yield a version of p, of size comparable to p, as output. That is, the overhead involved in interpreting a language (deciding which command is to be executed, incrementing a program counter, *etc.*) should be removed by a good partial evaluator. Our partial evaluator seems to approach this standard when run on small programs, though more detailed testing is needed.

3 An Example

Consider the C function `strcpy`:

```
void strcpy (char *s, char *t) { while (*s++ = *t++) ; }
```

This function copies a string from the memory location indicated by t to the memory location indicated by s. It is admittedly cryptic.

In [2], we presented an ealgebra interpreter for the C programming language. As a test, we ran our partial evaluator on our algebra for C, specializing it with respect to `strcpy()`. The result, with most of the functions renamed, appears below.

if *K* = *"init"* **then** *CopyFrom* := `t`, *CopyTo* := `s`, *K* := *"first-update"* **endif**
if *K* = *"first-update"* **then**
  TmpFrom := *CopyFrom*, *TmpTo* := *CopyTo*
  CopyFrom := *CopyFrom* + *1*, *CopyTo* := *CopyTo* + *1*, *K* := *"loop"*
endif
if *K* = *"loop"* **then**
  if *Memory(TmpFrom)* ≠ *0* **then**
    Memory(TmpTo) := *Memory(TmpFrom)*
    TmpFrom := *CopyFrom*, *TmpTo* := *CopyTo*
    CopyFrom := *CopyFrom* + *1*, *CopyTo* := *CopyTo* + *1*, *K* := *"loop"*
  else *Memory(TmpTo)* := *Memory(TmpFrom)*, *K* := *"done"*
  endif
endif

This algebra is considerably smaller than the entire ealgebra for C, and hopefully is more easily understandable than the original C code. It is not optimal: for example, *CopyFrom* could be replaced by `t`, since `t` is never used after the initial state. It does, however, exhibit the behavior of `strcpy()` more directly than the entire ealgebra for C given `strcpy` as input.

(For those familiar with [2], the input functions initially specified as positive were *CurTask*, *TaskType*, *NextTask*, *LeftTask*, *RightTask*, *TrueTask*, *FalseTask*, *Decl*, *WhichChild*, and *ChooseChild*, assuming that *ChooseChild* always moves to the left first.)

References

[1] Yuri Gurevich, "Evolving Algebras 1993: Lipari Guide", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, to appear.

[2] Yuri Gurevich and James K. Huggins, "The Semantics of the C Programming Language", Selected papers from CSL'92 (Computer Science Logic), Springer Lecture Notes in Computer Science 702, 1993, 274–308.

[3] N.D. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993.

Equivalence Is In The Eye Of The Beholder

Yuri Gurevich*and James K. Huggins*

Abstract

This is a reaction to Lamport's "Processes are in the Eye of the Beholder." To illustrate the "insubstantiality of processes," Lamport represents a 2-process algorithm and an N-process algorithm by temporal formulas and proves the equivalence of the two formulas. We analyze in what sense the two algorithms are and are not equivalent, and give a more direct equivalence proof.

1 Introduction

In "Processes are in the Eye of the Beholder" [7], Leslie Lamport writes:

> A concurrent algorithm is traditionally represented as the composition of processes. We show by an example that processes are an artifact of how an algorithm is represented. The difference between a two-process representation and a four-process representation of the same algorithm is no more fundamental than the difference between $2 + 2$ and $1 + 1 + 1 + 1$.

To demonstrate his thesis, Lamport uses two different programs for a first-in, first-out ring buffer of size N, both written in a CSP-like language [4]. The first, shown in Figure 1, operates the buffer using 2-processes; the second, shown in Figure 2, uses N-processes. We call these two programs $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$, respectively (for reasons which shall become apparent). After presenting the programs, Lamport represents them as formulas in TLA, the Temporal Logic of Actions [6], and proves the equivalence of the two formulas in TLA.

There are three issues where we disagree with Lamport.

Issue 1: The Notion of Equivalence. What does it mean that two programs are equivalent? The answer to the question depends on which abstraction you find appropriate. Here are some possible definitions of equivalence.

1. The two programs produce the same output when given the same input.
2. The two programs produce the same output when given the same input, and the two programs are of the same time complexity (with respect to your favorite definition of time complexity).
3. The two programs produce the same output when given the same input, and the two programs take *precisely* the same amount of time.
4. No external observer of the two programs in execution can detect any difference.

*CSE Technical Report CSE-TR-240-95. EECS Department, University of Michigan–Ann Arbor, Ann Arbor, MI, 48109-2122, USA. {gurevich, huggins}@umich.edu. Partially supported by ONR grant N00014-94-1-1182 and NSF grant CCR-92-04742.

$$
\begin{array}{l}
\mathit{in,\ out} : \textbf{channel of } \mathit{Value} \\
\mathit{buf} : \textbf{array } 0..N-1 \textbf{ of } \mathit{Value} \\
\mathit{p,g} : \textbf{internal } \mathit{Natural} \textbf{ initially } 0 \\
\mathit{Receiver} :: * \left[\begin{array}{lcl} p - g \neq N & \rightarrow & \mathit{in}\ ?\ \mathit{buf}[p\ \mathit{mod}\ N]; \\ & & p := p + 1 \end{array} \right] \\
\quad \| \\
\mathit{Sender} :: \ * \left[\begin{array}{lcl} p - g \neq 0 & \rightarrow & \mathit{out}\ !\ \mathit{buf}[g\ \mathit{mod}\ N]; \\ & & g := g + 1 \end{array} \right]
\end{array}
$$

Figure 1: A 2-process ring buffer L_2, in a CSP-like language.

$$
\begin{array}{l}
\mathit{in,\ out} : \textbf{channel of } \mathit{Value} \\
\mathit{buf} : \textbf{array } 0..N-1 \textbf{ of } \mathit{Value} \\
\mathit{pp,gg} : \textbf{internal array } 0..N-1 \textbf{ of } \{0,1\} \textbf{ initially } 0 \\
\mathit{Buffer}(i\text{: } 0..N-1) :: \\
\quad * \left[\begin{array}{rcl} \mathit{empty} : \ \mathit{IsNext}(pp, i) & \rightarrow & \mathit{in}\ ?\ \mathit{buf}[i]; \\ & & pp[i] := (pp[i] + 1)\ \mathit{mod}\ 2; \\ \mathit{full} : \ \mathit{IsNext}(gg, i) & \rightarrow & \mathit{out}\ !\ \mathit{buf}[i]; \\ & & gg[i] := (gg[i] + 1)\ \mathit{mod}\ 2; \end{array} \right] \\
\\
\mathit{IsNext}(r, i) \ \triangleq \ \textbf{if } i = 0 \ \textbf{ then } \ r[0] = r[N-1] \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \ r[i] \neq r[i-1]
\end{array}
$$

Figure 2: An N-process ring buffer L_N, in a CSP-like language.

The reader will be able to suggest numerous other reasonable definitions for equivalence. For example, one could substitute space for time in conditions 2 and 3 above. In particular, the nature of an "external observer" in condition 4 is admittedly vague and has several plausible interpretations.

Which notion of equivalence is the best one? Naturally, it depends upon the application. We do not promote any particular notion of equivalence, but note that there are many reasonable definitions. There is no one definition of equivalence that is best for all situations [1].

$\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ are indeed "strongly equivalent"; in particular, they are equivalent in the sense of definition 3 above. However, they are not equivalent in the sense of definition 4 for certain external observers, or in the sense of some space-complexity versions of definitions 2 and 3.

Issue 2: Representing Programs as Formulas. Again, we quote Lamport [7]:

> We will not attempt to give a rigorous meaning to the program text. Programming languages evolved as a method of describing algorithms to compilers, not as a method for reasoning about them. We do not know how to write a completely formal proof that two programming language representations of the ring buffer are equivalent. In Section 2, we represent the program formally in TLA, the Temporal Logic of Actions [6].

We believe that it is not only possible but also beneficial to prove the desired equivalence of programs directly. We cannot, however, prove the desired equivalence of $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ because the CSP-like language used lacks a formal semantics. (Actually, the CSP-like language seems to us to be a method for describing algorithms for humans, not compilers.) Instead, we formalize Lamport's interpretations of $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ as evolving algebra programs $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$. We then define an appropriate notion of equivalence and show that $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ are equivalent in this sense.

Issue 3: The Formality of Proofs. Continuing, Lamport writes [7]:

> We now give a hierarchically structured proof that Π_2 and Π_N [the TLA translations of $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ – GH] are equivalent [5]. The proof is completely formal, meaning that each step is a mathematical formula. English is used only to explain the low-level reasoning. The entire proof could be carried down to a level at which each step follows from the simple application of formal rules, but such a detailed proof is more suitable for machine checking than human reading. Our complete proof, with "Q.E.D." steps and low-level reasoning omitted, appears in Appendix A.

We prefer to separate the process of explaining a proof to people from the process of computer-aided verification of the same proof [2]. An important benefit of this separation of concerns is that a human-oriented exposition is much easier for humans to read and understand than expositions attempting to satisfy both concerns at once. Writing a good human-oriented proof is the art of creating the correct images in the mind of the reader. Such a proof is amenable to the traditional social process of debugging mathematical proofs. Granted, mathematicians make mistakes and computer-aided verification may be desirable, especially in safety-critical applications. But the two concerns can and should be treated separately.

These disagreements do not mean that our position on "the insubstantiality of processes" is the direct opposite of Lamport's. We simply point out that "the insubstantiality of processes" may itself be in the eye of the beholder. The same two programs can be equivalent with respect to some reasonable definitions of equivalence and inequivalent with respect to others.

This paper is self-contained. In Section 2, we describe Lamport's first-in, first-out ring buffer example. Section 3 contains a brief introduction to evolving algebras, culminating with the evolving algebra programs $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ to be compared. In Section 4, we define a strong version of lock-step equivalence and prove that $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ are equivalent in that sense. Finally, we discuss the inequivalence of $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ in section 5.

2 The Ring Buffer Example

The problem we consider is a first-in, first-out ring buffer of size N. The buffer is implemented as an array of N elements. Input number i (starting with $i = 0$) is stored in position $i \ mod \ N$ until it is sent out as output number i. Input number i and output number j may occur concurrently only if $i \neq j \ mod \ N$. Items may be placed in the buffer if the buffer is not full; of course, items may be sent from the buffer if the buffer is not empty.

Each input or output action is dependent upon other actions within the system. Input number i cannot occur until all previous inputs have occurred and either $i < N$ or output number $i - N$ has occurred. Output number i cannot occur until all previous outputs and input number i have occurred. These dependencies are illustrated pictorially in Figure 3, where circles represent the actions to be taken and arrows represent dependency relationships between actions.

$\mathcal{R}_{csp}$ decomposes this graph into two rows, each row representing one of the processes of the algorithm. Similarly, $\mathcal{C}_{csp}$ decomposes this graph into columns. Figures 4 and 5 show this decomposition.

3 Evolving Algebras

We recall only as much of evolving algebra theory [3] as needed in this paper. The term evolving algebra is often abbreviated *ealgebra* or *EA*. Those already familiar with ealgebras may wish to skip ahead to section 3.6, in which the ealgebras for $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ are given.

3.1 States

A *vocabulary* is a finite collection of function names, each of fixed arity. Every vocabulary contains the nullary function names *true* and *false* and (the names of) the usual Boolean operations.

A *state* s of vocabulary Υ is a nonempty set X, the *superuniverse* of S, together with interpretations of the function names in Υ on X. An r-ary function name is interpreted as a function from X^r to X. The interpretations of function names *true* and *false* are always distinct and are operated upon in the usual way by the Boolean operations. We denote the value of a term t in state s by t_s.

A Boolean-valued function f may be viewed as the set of tuples where it evaluates to *true*. If f is unary it can be viewed as a special *universe*. For example, we may have a universe *nodes* and declare a binary relation *Edge* over the universe of *nodes*; *Edge(x,y)* will hold only if both x and y belong to *nodes*. Such universes allow us to view states as many-sorted structures.

Certain function names in Υ are called *external*; the idea is that the values of external functions will be determined outside of the ealgebra. External functions can be used, for example, to model input provided by the external world. Non-external function names are called *internal*.

If Υ is a vocabulary, Υ^- is the set of internal function names of Υ. If s is a state of vocabulary Υ, s^- is the state of vocabulary Υ^- which agrees with s over all functions in Υ^-.

3.2 Updates

A *location* of state S is a pair $\ell = (f, \bar{x})$, where f is a function name and $\bar{x}$ is a tuple of elements of S whose length equals the arity of f. An *update* of a state S is a pair $\alpha = (\ell, y)$, where ℓ is a location of S and y is an element of S. To *fire* α at S, put y into the location ℓ; that is, if $\ell = (f, \bar{x})$, redefine S to interpret $f(\bar{x})$ as y; nothing else is changed.

An *update set* over a state S is a set of updates of S. A set of updates is *consistent* at S if no two updates in the set have the same location but different values. To fire a consistent set at S, fire all its members simultaneously; to fire an inconsistent set at S, do nothing.

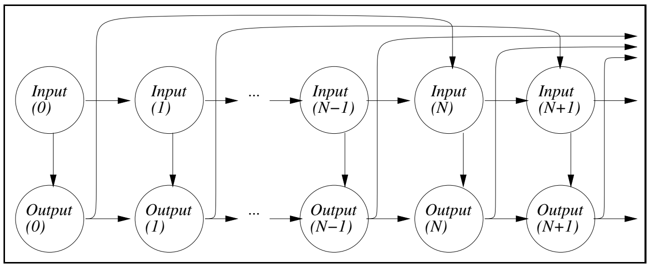


Figure 3: Moves of the ring-buffer algorithm.

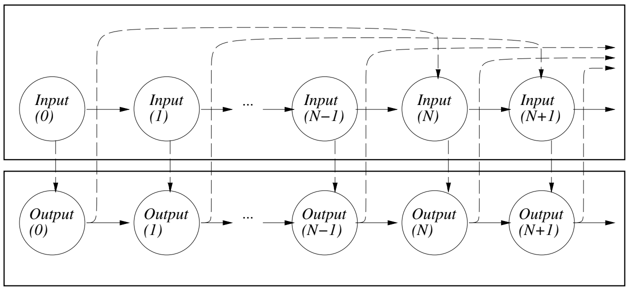


Figure 4: Moves of $\mathcal{R}_{csp}$ and $\mathcal{R}_{ea}$.

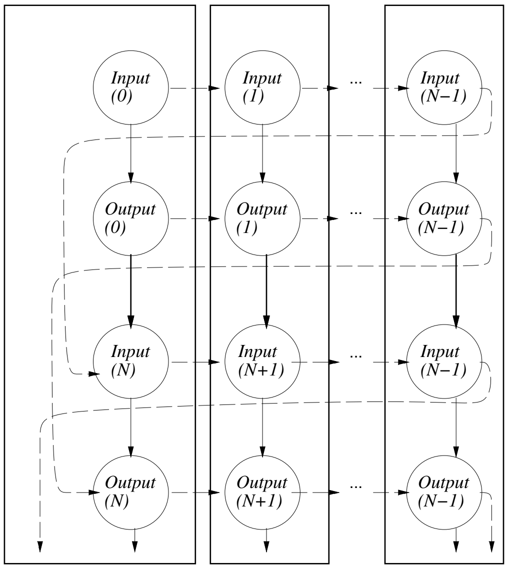


Figure 5: Moves of $\mathcal{C}_{csp}$ and $\mathcal{C}_{ea}$.

3.3 Transition Rules

Executing a rule R at state S is equivalent to executing an appropriate update set. There are three types of basic transition rules.

An *update instruction* R is an expression

$$f(t_1, \ldots, t_r) := t_0$$

where f is a non-static function name of arity r and each t_i is a term. The update set of R at state S has a single element (ℓ, y), where $y = (t_0)_S$ and $\ell = (f, (x_1, \ldots, x_r))$ with $x_i = (t_i)_S$.

A *block rule* R is a sequence of transition rules. The update set of R at S is the union of the update sets of the rules in the sequence at S.

A *conditional rule* R is an expression

if g_0 **then** R_0
elseif g_1 **then** R_1
⋮
elseif g_k **then** R_k
endif

where the g_i are terms and the R_i are rules. (If $g_k \equiv true$, we usually write "**else** R_k".) The update set of R at S is the update set of R_i at S, where g_i is true in S but every g_j with $j < i$ is false in S. If no such i exists, the update set is empty.

3.4 Distributed Evolving Algebras

A *distributed evolving algebra* $\mathcal{A}$ consists of the following:

- A finite indexed set of *modules*, each of which is a transition rule. Intuitively, a module is the program to be executed by one or more agents. Each module is assigned a unique nullary function name.
- A vocabulary Υ containing all functions used in the modules of $\mathcal{A}$, except for the nullary function *Me*. (*Me* will be interpreted differently by each agent and thus must be treated separately.) Υ also contains an additional unary function name *Mod*.
- A collection of Υ-states, called *initial states*.

Structure S of signature Υ is a *state* of $\mathcal{A}$ if module names are interpreted as different elements of S and only finitely many elements a (called *agents*) exist such that *Mod(a)* equals some module name.

$View_a(S)$ is the reduct of S to the functions mentioned in the module *Mod(a)*, expanded by interpreting *Me* as a. Think about $View_a(S)$ as the local state of agent a corresponding to the global state S. To *fire* a at S, fire *Mod(a)* at $View_a(S)$.

3.5 Runs

A *run* ρ of a distributed ealgebra $\mathcal{A}$ of vocabulary Υ from initial state S_0 can be defined as a triple (M, A, σ) satisfying the following conditions.

1. M, the set of *moves* of ρ, is a partially ordered set where every $\{y : y \leq x\}$ is finite.

   Intuitively, $x < y$ means move x completes before move y begins.

2. A assigns elements of S_0 to moves in such a way that every non-empty set $\{x : A(x) = a\}$ is linearly ordered.

   Intuitively, $A(x)$ is the agent performing move x; every agent acts sequentially.

3. σ maps finite initial segments of M (including $\emptyset$) to states of $\mathcal{A}$.

   Intuitively, $\sigma(X)$ is the result of performing all moves of X; $\sigma(\emptyset)$ is an initial state.

4. *Coherence.* If μ is a maximal element of a finite initial segment Y of M, and $X = Y - \{\mu\}$, then $A(\mu)$ is an agent in $\sigma(X)$ and $\sigma(Y)^-$ is obtained by firing $A(\mu)$ at $\sigma(X)$.

Given a finite initial segment Y with a maximal element μ and $X = Y - \{\mu\}$, one may want to associate the state $\sigma(X)$ with μ. One particular choice of X is $\{\nu : \nu < \mu\}$. We define $\Lambda(\mu) = \Lambda_{\mathcal{A}}(\mu) = \sigma(\{\nu : \nu < \mu\})$.

3.6 Evolving Algebras for the Ring Buffer Problem

Here we present evolving algebras $\mathcal{R} = \mathcal{R}_{ea}$ and $\mathcal{C} = \mathcal{C}_{ea}$, corresponding to $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$, respectively.

Each ealgebra has universes of *data, integers*, int_N (integers modulo N), and int_2. As usual, we identify elements of int_N with their representatives $0, 1, \ldots N-1$ (and similarly for int_2). The buffer is represented by a unary function *Buffer:* $int_N \rightarrow$ *data.*

Following Lamport (but using different names), we model the input and output channels with signaling bits *InBit, OutBit:* int_2 and functions *InData, OutData: data. InBit* and *InData* are external functions; when the external world places a new datum into *InDatum*, *InBit* is also flipped. *MyBit:* int_2 reflects the most recent value of *InBit* that has been seen by the algorithm.

Algorithms perform arithmetic over *integers*, int_N and int_2; we use the standard arithmetic functions with their conventional notations.

3.7 $\mathcal{R}$: The Rows Evolving Algebra

Ealgebra $\mathcal{R}$ describes the 2-process algorithm for the ring buffer. One agent (called I) reads data into the buffer; the other (called O) sends data from the buffer. (I and O are different elements of $\mathcal{R}$, *e.g.* 0 and 1.)

$\mathcal{R}$ uses counters *First, Last: integers* to indicate the current buffer slot to be used for input or output; initially, *First = Last = 0.* The two modules are given in Figure 6.

```
Module: Input
if MyBit ≠ InBit and First − Last ≠ N then
      Buffer(First mod N) := InData, MyBit := 1 − MyBit, First := First + 1
endif

Module: Output
if Last ≠ First then
      OutData := Buffer(Last mod N), OutBit := 1 − OutBit, Last := Last + 1
endif
```

Figure 6: $\mathcal{R}$: A 2-process ring buffer algorithm.

For any run (M, A, σ) of $\mathcal{R}$, we call a move μ an *input move* (respectively, an *output move*) if $A(\mu) = I$ (resp. O). We call a move μ a *k-slot move* if either μ is an input move and *First* $= k$ in state $\Lambda(\mu)$ or μ is an output move and *Last* $= k$ in state $\Lambda(\mu)$.

Lemma 3.1 *Let Y be a finite initial segment of M, μ a maximal element of Y, and $X = Y - \{\mu\}$. If μ is a k-slot input (respectively, output) move, then we have modulo N:*

1. $First_{\sigma(X)} = k$ *(resp.,* $Last_{\sigma(X)} = k$*)*

2. $First_{\sigma(Y)}$ = k+1 (resp., $Last_{\sigma(Y)}$ = k+1)

Proof: We prove the property for input moves; the proof for output moves is similar. Condition 2 follows from condition 1 by definition of μ.

Obviously $\Lambda(\mu) \subseteq X$. Since μ is a k-slot input move, $First_{\sigma(\Lambda(\mu))} = k \ mod \ N$. Any $\nu \in X - \{\mu' : \mu' < \mu\}$ is incomparable with μ and therefore is an output move. Let $s_0 = \Lambda(\mu), s_1, \ldots, s_n = \sigma(X)$ be a sequence of states of $\mathcal{R}$ such that every s_{m+1}^- is obtained from s_m by executing a move of $X - \{\mu' : \mu' < \mu\}$. Since output moves do not alter *First*, $First_{\sigma(X)} = First_{\Lambda(\mu)}$. □

3.8 $\mathcal{C}$: The Columns Evolving Algebra

Ealgebra $\mathcal{C}$ describes the N-process algorithm for our ring buffer. The universe of agents is simply int_N. It uses two nullary functions *get* and *put* with different values (*e.g.* 0 and 1).

Functions *InCount, OutCount:* $int_N \rightarrow int_2$ are used to count (modulo 2) the number of input and output operations performed; initially, both have the value 0 for each agent. Function *Mode:* $int_N \rightarrow$ *modes* notes whether each agent is receiving input (when *Mode(x) = get*) or producing output (when *Mode(x) = put*). Initially, *Mode(x) = get* for all agents *x*.

We present in Figure 7 the module used by all agents, which contains a block of two rules.

Rule: Get
if *MyBit* $\neq$ *InBit* **and** *Mode(Me) = get* **and** INPUTTURN*(Me)* **then**
  Buffer(Me) := InData, MyBit := 1 – MyBit
  InCount(Me) := 1 – InCount(Me), Mode(Me) := put
endif

Rule: Put
if *Mode(Me) = put* **and** OUTPUTTURN*(Me)* **then**
  OutData := Buffer(Me), OutBit := 1 – OutBit
  OutCount(Me) := 1 – OutCount(Me), Mode(Me) := get
endif

Abbreviations:
INPUTTURN*(Me)* $\equiv$ *(Me = 0* **and** *InCount(0) = InCount(N-1))*
  or *(Me* $\neq$ *0* **and** *InCount(Me)* $\neq$ *InCount(Me-1))*
OUTPUTTURN*(Me)* $\equiv$ *(Me = 0* **and** *OutCount(0) = OutCount(N-1))*
  or *(Me* $\neq$ *0* **and** *OutCount(Me)* $\neq$ *OutCount(Me-1))*

Figure 7: $\mathcal{C}$: An N-process ring buffer algorithm.

3.9 Remarks

One may argue that *InBit* and *OutBit* are not used in $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ and should not be used in $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$. It is easy to avoid using *InBit* and *OutBit* and make the input and output channels implicit, as in $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$. Lamport uses a trick with bits to model his input and output channels, and we followed him (though we altered the trick insignificantly for elegance). Note that both algorithms use identical input and output mechanisms; thus, different presentations of this mechanism do not affect the equivalence of the two algorithms.

One the other hand, one may wish to make the actions of the external world explicit in $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$. It is easy to do this, but is not needed for our purposes here.

4 Equivalence

Here we define a strong version of lock-step equivalence for ealgebras which, for brevity, we call simply *lock-step equivalence*. We also define an even stronger version of lock-step equivalence which we call *strict lock-step equivalence.*

4.1 Strict Lock-Step Equivalence

Let $\mathcal{A}$ and $\mathcal{B}$ be two ealgebras with the same superuniverse.

Suppose h is a one-to-one mapping from states of $\mathcal{A}$ onto states of $\mathcal{B}$, such that if $h(s) = t$, then s and t coincide on function names common to $\mathcal{A}$ and $\mathcal{B}$. A sequential run $\langle s_i : i < \alpha \rangle$ of $\mathcal{A}$ is *strictly h-similar* to a sequential run $\langle s'_j : j < \beta \rangle$ if $\alpha = \beta$ and $s'_i = h(s_i)$ for every $i < \alpha$. A partially ordered run (M, A, σ) of $\mathcal{A}$ is *strictly h-similar* to a partially ordered run (M', A', σ') of $\mathcal{B}$ if there is an isomorphism $\eta : M \longrightarrow M'$ such that for every finite initial segment X of M, $h(\sigma(X)) = \sigma'(Y)$, where $Y = \{\eta(\mu) : \mu \in X\}$. (Note that we permit $X = \emptyset$, in which case $Y = \emptyset$ as well.)

$\mathcal{A}$ and $\mathcal{B}$ are *strictly h-similar* if every run of $\mathcal{A}$ is strictly h-similar to a run of $\mathcal{B}$, and every run of $\mathcal{B}$ is h^{-1}-similar to a run of $\mathcal{A}$. We say $\mathcal{A}$ and $\mathcal{B}$ are *strictly lock-step equivalent* if $\mathcal{A}$ and $\mathcal{B}$ are h-similar for some h.

Ideally, this is the type of equivalence we would like to use. However, this is not possible. $\mathcal{R}$ uses functions (*First, Last*) which have infinitely many possible values while their counterparts in $\mathcal{C}$ (*InCount, OutCount*) have only finitely many values. Imagine that the universe of *data* is finite. Then $\mathcal{R}$ has infinitely many states while $\mathcal{C}$ has only finitely many states. A one-to-one mapping h will not be possible in this case.

We can easily re-write either $\mathcal{R}$ or $\mathcal{C}$ to make them strictly lock-step equivalent. (For example, it suffices to modify $\mathcal{C}$ to perform math on *InCount* and *OutCount* over *integers* instead of int_2.) However, a slightly weaker version of this equivalence will do.

4.2 Lock-Step Equivalence

Let $\cong$ be an equivalence relation over states of an ealgebra $\mathcal{A}$. Let *Result(a,s)* denote the state resulting from agent a firing at state s. We call $\cong$ a *congruence* if for any states s_1 and s_2 and any agent a, $s_1 \cong s_2 \rightarrow Result(a, s_1) \cong Result(a, s_2)$. We call the equivalence classes of $\cong$ *configurations* (with respect to $\cong$), and denote by $[s]$ the configuration to which state s belongs.

Let $\mathcal{A}$ and $\mathcal{B}$ be ealgebras with congruences $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$, respectively. (We will drop the subscripts on $\cong$ when no confusion arises.) We assume that $\mathcal{A}$ and $\mathcal{B}$ have the same superuniverse and $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$ preserve the interpretation of function names common to $\mathcal{A}$ and $\mathcal{B}$. In other words, if f is a common function name, p and q are both states of $\mathcal{A}$ (or both states of $\mathcal{B}$), and $p \cong q$, then $f_p = f_q$; that is, $f(\bar{x})_p = f(\bar{x})_q$ for all $\bar{x}$.

Let h be a one-to-one mapping from the set of configurations of $\mathcal{A}$ onto the set of configurations of $\mathcal{B}$ such that for every function name f common to $\mathcal{A}$ and $\mathcal{B}$, if $h([p]) = [s]$, then $f_p = f_s$. A partially ordered run (M, A, σ) of $\mathcal{A}$ is *h-similar* to a partially ordered run (M', A', σ') of $\mathcal{B}$ if there is an isomorphism $\eta : M \longrightarrow M'$ such that for every finite initial segment X of M, $h([\sigma(X)]) = [\sigma'(Y)]$, where $Y = \{\eta(\mu) : \mu \in X\}$.

$\mathcal{A}$ and $\mathcal{B}$ are *h-similar* if every run of $\mathcal{A}$ is h-similar to a run of $\mathcal{B}$, and every run of $\mathcal{B}$ is h^{-1}-similar to a run of $\mathcal{A}$. $\mathcal{A}$ and $\mathcal{B}$ are *lock-step equivalent* (with respect to $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$) if $\mathcal{A}$ and $\mathcal{B}$ are h-similar for some h.

Note that strict lock-step equivalence is a special case of lock-step equivalence, where $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$ are both the identity relation.

We will show that $\mathcal{R}$ is lock-step equivalent to $\mathcal{C}$ with respect to the congruences defined below. We assume that $\mathcal{R}$ and $\mathcal{C}$ have the same superuniverse.

Definition 1 *For states s, t of $\mathcal{C}$, $s \cong_{\mathcal{C}} t$ if $s = t$.*

Since each configuration of $\mathcal{C}$ has only one element, we identify a state of $\mathcal{C}$ with its configuration.

Definition 2 *For states p, q of $\mathcal{R}$, $p \cong_{\mathcal{R}} q$ if:*

- $Last_p = Last_q \ mod \ 2N$
- $(First - Last)_p = (First - Last)_q$
- $f_p = f_q$ *for all other function names* f.

Let *div* represent integer division: $i \ div \ j = \lfloor i/j \rfloor$.

Lemma 4.1 *If $p \cong_{\mathcal{R}} q$ then we have the following modulo 2:*

- $First_p \ div \ N = First_q \ div \ N$
- $Last_p \ div \ N = Last_q \ div \ N$

Proof: We prove the desired property for *First*; the proof for *Last* is similar.

By the definition of $\cong_{\mathcal{R}}$, we have modulo $2N$ that $First_p = Last_p + (First - Last)_p = Last_q + (First - Last)_q = First_q$. Thus, there are non-negative integers x_1, x_2, x_3 such that $First_p = 2Nx_1 + Nx_2 + x_3$ and $x_2 \leq 1$ and $x_3 < N$. Since $First_p = First_q \ mod \ 2N$, there exists a non-negative integer y_1 such that $First_q = 2Ny_1 + Nx_2 + x_3$. Hence $First_p \ div \ N = 2x_1 + x_2$ and $First_q \ div \ N = 2y_1 + x_2$, which are equal modulo 2. □

We define a mapping h from configurations of $\mathcal{R}$ onto configurations of $\mathcal{C}$.

Definition 3 *If $h([p]) = s$, then*

$$InCount(i)_s = \begin{cases} (First_p \ div \ N) \ mod \ 2 & \text{if } i \geq First_p \ mod \ N \\ 1 - (First_p \ div \ N) \ mod \ 2 & \text{otherwise} \end{cases}$$
$$OutCount(i)_s = \begin{cases} (Last_p \ div \ N) \ mod \ 2 & \text{if } i \geq Last_p \ mod \ N \\ 1 - (Last_p \ div \ N) \ mod \ 2 & \text{otherwise} \end{cases}$$

and for all other common function names f, $f_p = f_s$.

Thus, h relates the counters *First* and *Last* used in $\mathcal{R}$ and the counters *InCount* and *OutCount* used in $\mathcal{C}$. We have not said anything about *Mode* because *Mode* is uniquely defined by the rest of the state (see Lemma 4.7 in section 4.3) and is redundant. We only include *Mode* in our specification for $\mathcal{C}$ because we wish the translation of $\mathcal{C}_{csp}$ to $\mathcal{C}$ to be more faithful.

We now prove that $\mathcal{R}$ and $\mathcal{C}$ are h-similar.

4.3 Properties of $\mathcal{R}$

Lemma 4.2 *For any state q of any run of $\mathcal{R}$, $0 \leq (First_q - Last_q) \leq N$.*

Proof: By induction. Initially, $First = Last = 0$.

Let X be a finite initial segment of a run with maximal element x, such that $0 \leq First - Last \leq N$ holds in $p = \sigma(X - \{x\})$. Let $q = \sigma(X)$.

- If x is an execution of rule Input, $(First - Last)_p < N$. Input increments *First* and does not alter *Last*; thus, $0 < (First - Last)_q \leq N$.
- If x is an execution of rule Output, $Last_p < First_p$ (*i.e.* $0 < (First - Last)_p$). Output increments *Last* and does not alter *First*; thus, $0 \leq (First - Last)_q < N$. □

Lemma 4.3 *Fix a non-negative integer $k < N$. For any run (M, A, σ) of $\mathcal{R}$, the k-slot moves of M are linearly ordered.*

Proof: By contradiction, suppose that two k-slot moves μ, ν are incomparable. Since all input moves are comparable and all output moves are comparable, one of these moves (say μ) is an input move while the other (that is, ν) is an output move.

Let $X = \{\xi : \xi < \mu \text{ or } \xi < \nu\}$, $p = \sigma(X)$, $Y = X \cup \{\mu\}$, and $Z = X \cup \{\nu\}$. By the coherence condition applied to Y and μ, $A(\mu) = I$ is enabled in X; consequently, $First_p = k \mod N$ and $First_p - Last_p \neq N$. By the coherence condition applied to Z and ν, $A(\nu) = O$ is enabled in X; consequently, $Last_p = k \mod N$ and $First_p - Last_p \neq 0$. Combining with Lemma 4.2, we have $0 < First_p - Last_p < N$ and $First_p = Last_p \mod N$, which is impossible. □

4.4 Properties of $\mathcal{C}$

Lemma 4.4 *For any state t of any run of $\mathcal{C}$, there is an agent $k = In(t) < N$ such that:*

- InputTurn*(Me) is true for agent k and for no other agent.*
- *For all $i < k$, $InCount(i) = 1 - InCount(k)$.*
- *For all $k \leq i < N$, $InCount(i) = InCount(k)$.*

Proof: By induction. Initially, agent 0 (and no other) satisfies InputTurn*(Me)* and $InCount(Me) = 0$ for all agents. Thus, if t is an initial state, $In(t) = 0$.

Let Y be a finite initial segment of a run of $\mathcal{C}$ with maximal element y, such that the requirements hold in $s = \sigma(Y - \{y\})$. Let $t = \sigma(Y)$. Since rule Put does not modify *InCount*, it cannot affect the three requirements.

The desired $In(t) = In(s) + 1 \mod N$. This is obvious in the case that $In(s) < N{-}1$. If $In(s) = N{-}1$, then all values of *InCount* are equal in t, so $In(t) = 0$ satisfies the three requirements. □

Lemma 4.5 *For any state t of any run of $\mathcal{C}$, there is exactly one agent $k = Out(t) < N$ such that:*

- OutputTurn*(Me) is true for agent k and no other agent.*
- *For all $i < k$, $OutCount(i) = 1 - OutCount(k)$.*
- *For all $k \leq i < N$, $OutCount(i) = OutCount(k)$.*

Proof: Parallel to that of the last lemma. □

Lemma 4.6 *For any run (M, A, σ) of $\mathcal{C}$, the set $\{\mu : A(\mu)$ executes rule Get in $\Lambda(\mu)\}$ is linearly ordered, as is the set $\{\mu : A(\mu)$ executes rule Put in $\Lambda(\mu)\}$.*

Proof: We prove the claim for Get; the proof for Put is similar. By contradiction, suppose that are two incomparable moves μ, ν such that Get is enabled in both $\Lambda(\mu)$ and $\Lambda(\nu)$.

Let $X = \{\xi : \xi < \mu \text{ or } \xi < \nu\}$, $p = \sigma(X)$, $Y = X \cup \{\mu\}$, and $Z = X \cup \{\nu\}$. By the coherence condition applied to Y and μ, $A(\mu)$ fires at p; by Lemma 4.4, $In(p) = A(\mu)$. Similarly, $A(\nu)$ fires at p and $In(p) = A(\nu)$. Thus, μ and ν must be moves of the same agent k. But all moves of the same agent are ordered, so μ and ν cannot be incomparable. □

Lemma 4.7 *In any state t of any run of $\mathcal{C}$, for any agent k,*

$$Mode(Me) = \begin{cases} get & \text{if } InCount(Me) = OutCount(Me) \\ put & \text{if } InCount(Me) = 1 - OutCount(Me) \end{cases}$$

Proof: We fix a k and perform an induction over runs. Initially, *Mode(Me) = get* and *InCount(Me) = OutCount(Me) = 0* for every agent.

Let Y be a finite initial segment of a run with maximal element y. Thus, the required condition holds in $s = \sigma(Y - \{y\})$.

If move Y is executed by any agent other than k, no functions named in the required condition are affected. If agent k does execute move y, there are two cases.

- If y is an execution of rule Get, the guard ensures that *Mode(Me) = get*. Get sets *Mode(Me)* to *put* and executes *InCount(Me) := 1 - InCount(Me)*, maintaining the requirement.
- If y is an execution of rule Put, the guard ensures that *Mode(Me) = put*. Put sets *Mode(Me)* to *get* and executes *OutCount(Me) := 1 - OutCount(Me)*, maintaining the requirement. □

Remark. This lemma shows that function *Mode* is indeed redundant. The only substantial difference between the signatures of $\mathcal{R}$ and $\mathcal{C}$ is the presence of *First* and *Last* in $\mathcal{R}$ and the presence of *InCount* and *OutCount* in $\mathcal{C}$.

4.5 Proof of Equivalence

Lemma 4.8 *If $h([p]) = s$, then In(s) = $First_p$ mod N and Out(s) = $Last_p$ mod N.*

Proof: Recall that *In(s)* is the agent k for which INPUTTURN*(k)*$_s$ holds. Lemma 4.4 asserts that $InCount_s$ has one value everywhere below k and another value everywhere else. By definition of h, this "switch-point" in *InCount* occurs at $First_p$ *mod N*. The proof for *Out(s)* is similar. □

Lemma 4.9 *Rule Input is enabled in state p of $\mathcal{R}$ iff rule Get is enabled in state $s = h([p])$ of $\mathcal{C}$ for agent In(s).*

Proof: Let $k = In(s)$, so that INPUTTURN*(k)*$_s$ holds. Both Input and Get have $MyBit \neq InBit$ in their guards. It thus suffices to show that $(First - Last)_p \neq N$ iff $Mode(k)_s = get$. By Lemma 4.7, it suffices to show that $(First - Last)_p \neq N$ iff $InCount(k)_s = OutCount(k)_s$.

Suppose $(First - Last)_p \neq N$. There exist non-negative integers a_1, a_2, a_3, a_4 such that $First_p = a_1 N + a_3$, $Last_p = a_2 N + a_4$, and $a_3, a_4 < N$. (Note that by 4.8, $k = First_p \ mod \ N = a_3$.)

By Lemma 4.2, $0 \leq (First - Last)_p < N$. There are two cases.

- $a_1 = a_2$ and $a_3 \geq a_4$. By definition of h, we have that, modulo 2, $InCount(a_3)_s = First_p \ div \ N = a_1$ and $OutCount(i)_s = Last_p \ div \ N = a_2$ for all $i \geq Last_p \ mod \ N = a_4$. Since $a_3 \geq a_4$, $(OutCount(a_3) = a_2 = a_1 = InCount(a_3))_s$.
- $a_1 = (a_2 + 1)$ and $a_3 < a_4$. By definition of h, we have that, modulo 2, $InCount(a_3)_s = First_p \ div \ N = a_1$ and $OutCount(i)_s = 1 - (Last_p \ div \ N) = a_2 + 1$ for all $i < Last_p \ mod \ N = a_4$. Since $a_3 < a_4$, $(OutCount(a_3) = a_2 + 1 = a_1 = InCount(a_3))_s$.

On the other hand, suppose $(First - Last)_p = N$. Then $First_p$ *div N* and $Last_p$ *div N* differ by 1. By definition of h, $InCount(i)_s = 1 - OutCount(i)_s$ for all i, including k. □

Lemma 4.10 *Rule Output is enabled in state p iff rule Put is enabled in state $s = h([p])$ for agent Out(s).*

Proof: Similar to that of the last theorem. □

Lemma 4.11 *Suppose that rule Input is enabled in a state p of $\mathcal{R}$ and rule Get is enabled in a state $s = h([p])$ of $\mathcal{C}$ for agent In(s). Let q = Result(I,p) and t = Result(In(s), s). Then $t = h([q])$.*

Proof: We check that $h([q]) = t$.

- Both Input and Get execute *MyBit := InBit*

- Input executes *Buffer(First mod N) := InData*. Get executes *Buffer(In(s)) := InData*. By Lemma 4.8, *In(s)* = $First_p$ *mod N*, so these updates are identical.

- Input executes *First := First + 1*. Get executes *InCount(In(s)) := 1 - InCount(In(s))*. The definition of h and the fact that $InCount(i)_s = InCount(i)_{h([p])}$ imply that $InCount(i)_t = InCount(i)_{h([q])}$.

- Get executes *Mode(In(s)) := put*. By Lemma 4.7, this update is redundant and need not have a corresponding update in Input. □

Lemma 4.12 *Suppose that rule* Output *is enabled in a state* p *of* $\mathcal{R}$ *and rule* Put *is enabled in a state* $s = h([p])$ *of* $\mathcal{C}$ *for agent Out(s). Let* q = *Result(O,p) and* t = *Result(Out(s), s). Then* $t = h([q])$.

Proof: Parallel to that of the last theorem. □

Theorem 1 $\mathcal{R}$ *is lock-step equivalent to* $\mathcal{C}$.

Proof: Let $\Lambda(\mu) = \Lambda_{\mathcal{R}}(\mu)$ and $\Lambda'(\mu) = \Lambda_{\mathcal{C}}(\mu)$.

We begin by showing that any run (M, A, σ) of $\mathcal{R}$ is h-similar to a run of $\mathcal{C}$, using the definition of h given earlier. Construct a run (M, A', σ') of $\mathcal{C}$, where $\sigma'(X) = h([\sigma(X)])$ and A' is defined as follows. Let μ be a move of M and let $s = h([\Lambda(\mu)])$. Then $A'(\mu) = In(s)$ if $A(\mu) = I$, and $A'(\mu) = Out(s)$ if $A(\mu) = O$.

We check that (M, A', σ') satisfies the four requirements for a run of $\mathcal{C}$ stated in Section 3.5.

1. Trivial, since (M, A, σ) is a run.

2. By Lemma 4.3, it suffices to show that for any μ, if $A'(\mu) = k$, then $A(\mu)$ is a k-slot move. By the construction above and Lemma 4.8, we have modulo N that $k = In(s) = First_{\Lambda(\mu)}$ if $A(\mu) = I$ and $k = Out(s) = Last_{\Lambda(\mu)}$ if $A(\mu) = O$. In either case, μ is a k-slot move.

3. Since $\sigma' = h \circ \sigma$, σ' maps finite initial segments of M to states of $\mathcal{C}$.

4. *Coherence.* Let Y be a finite initial segment of M with a maximal element μ, and $X = Y - \{\mu\}$. Thus *Result*$(A(\mu),\sigma(X)) = \sigma(Y)$. By Lemma 4.9 or 4.10, $A'(\mu)$ is enabled in $\sigma'(X)$. By Lemma 4.11 or 4.12, *Result*$(A'(\mu), \sigma'(X)) = \sigma'(Y)$.

Continuing, we must also show that for any run (M, A', σ') of $\mathcal{C}$, there is a run (M, A, σ) of $\mathcal{R}$ which is h-similar to it.

We define A as follows. Consider the action of agent $A'(\mu)$ at state $\Lambda'(\mu)$. If $A'(\mu)$ executes rule Get, set $A(\mu) = I$. If $A'(\mu)$ executes rule Put, set $A(\mu) = O$.

We check that $\{\mu : A(\mu) = I\}$ is linearly ordered. By Lemma 4.6, it suffices to show that if $A(\mu) = I$, then $A'(\mu)$ executes Get in state $\Lambda'(\mu)$ — which is true by construction of A. $\{\mu : A(\mu) = O\}$ is linearly ordered by a similar argument.

We define σ inductively over finite initial segments of M. $\sigma(\emptyset)$ is the unique initial state in $h^{-1}(\sigma'(\emptyset))$.

Let Y be a finite initial segment with a maximal element μ such that σ is defined at $X = Y - \{\mu\}$. Choose $\sigma(Y)$ from $h^{-1}(\sigma'(Y))$ such that $\sigma(Y)^- = Result(A(\mu), \sigma(X))$. Is it possible to select such a $\sigma(Y)$? Yes. By Lemma 4.9 or 4.10, $A(\mu)$ is enabled in $\sigma(X)$ iff $A'(\mu)$ is enabled in $\sigma'(X)$. By Lemma 4.11 or 4.12, *Result*$(A(\mu), \sigma(X)) \in h^{-1}($*Result*$(A'(\mu), \sigma'(\mu)))$.

It is easy to check that (M, A, σ) is a run of $\mathcal{R}$ which is h-similar to (M, A', σ'). □

5 Inequivalence

Even though we have proven that $\mathcal{R}$ and $\mathcal{C}$ are lock-step equivalent, there are meaningful differences between $\mathcal{R}$ and $\mathcal{C}$. We consider several differences which could be detected by an external observer.

Magnitude of Values. $\mathcal{R}$ uses unrestricted integers as its counters; in constrast, $\mathcal{C}$ uses only single bits for the same purpose. Imagine that the universe of *data* is finite and small, and that an observer uses a computer with little memory to execute $\mathcal{R}$ and $\mathcal{C}$. $\mathcal{R}$'s counters may eventually exceed the memory capacity of the computer. $\mathcal{C}$ would have no such difficulties.

Types of Sharing. $\mathcal{R}$ shares access to the buffer between both agents; each agent in $\mathcal{C}$ has exclusive access to its portion of the buffer. Conversely, agents in $\mathcal{C}$ share access to both the input and output channels; each agent in $\mathcal{R}$ has exclusive access to one channel. Imagine an architecture in which input or output channels may not be exclusively held by an agent. It may not be possible to implement $\mathcal{R}$ directly on such a system.

Degree of Sharing. $\mathcal{R}$ has $N+2$ shared locations: the N locations of the buffer and 2 counter variables. $\mathcal{C}$ shares access to $2(N+2)$ locations: the input and output channel functions and the $2N$ counter variables. Sharing locations can be an expensive matter; if a location is being shared, some provision must be made for handling read/write conflicts to a given location. Imagine that a user must pay (either in time or in money) for each shared location (but not for private variables), regardless of size. In such a scenario, $\mathcal{C}$ would be more expensive than $\mathcal{R}$ to run.

These constrasts can be made more a little more dramatic. For example, one could construct an algorithm which uses $2N$ agents, each of which is responsible for an input or output action (but not both) to a particular buffer position. Virtually all of the locations it uses will be shared; yet, it is lock-step equivalent to $\mathcal{R}$ and $\mathcal{C}$. Yet few people would use this algorithm instead of $\mathcal{R}$ or $\mathcal{C}$ because it combines their disadvantages. Alternatively, one could write a single processor (sequential) algorithm which is equivalent in a different sense to $\mathcal{R}$ and $\mathcal{C}$; it would produce the same output as $\mathcal{R}$ and $\mathcal{C}$ when given the same input, but would not allow all orderings of actions possible for $\mathcal{R}$ and $\mathcal{C}$.

References

[1] Y. Gurevich. "Logic and the challenge of computer science." In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pp. 1–57, Computer Science Press, 1988.

[2] Y. Gurevich, "Logic Activities in Europe", ACM SIGACT News 25, No. 2, June 1994, 11–24.

[3] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995.

[4] C.A.R. Hoare, "Communicating sequential processes." *Communications of the ACM*, 21(8):666-667, August 1978.

[5] L. Lamport, "How to write a proof." Research Report 94, Digital Equipment Corporation, Systems Research Center, February 1993. To appear in American Mathematical Monthly.

[6] L. Lamport, "The temporal logic of actions." *ACM Transactions on Programming Languages and Systems*, 16(3):872-923, May 1994.

[7] L. Lamport, "Processes are in the Eye of the Beholder." Research Report 94, Digital Equipment Corporation, Systems Research Center, December 1994.

The Bakery Algorithm: Yet Another Specification and Verification

Egon Börger* Yuri Gurevich† Dean Rosenzweig‡

Abstract

In a meeting at Schloss Dagstuhl in June 1993, Uri Abraham and Menachem Magidor have challenged the thesis that an evolving algebra can be tailored to any algorithm at its own abstraction level. As example they gave an instructive proof which uses lower and higher views to show correctness of Lamport's bakery algorithm. We construct two evolving algebras capturing lower and higher view respectively, enabling a simple and concise proof of correctness for the bakery algorithm.

Introduction

Uri Abraham [Abraham93] has devised an instructive correctness proof for various variants of Lamport's bakery algorithm relying on a distinction between a lower view and a higher view of the algorithms. Actions at the higher level represents complex lower level computations. He formulates abstract conditions on higher level actions which are then shown to suffice for correctness and fairness (in form of a 'first-come-first-served' property and deadlock–freedom) and to be satisfied by the corresponding lower level computations.

At a seminar in Schloss Dagstuhl in June 1993 Uri Abraham and Menachem Magidor have expressed doubts that such a proof could be naturally carried out in the evolving algebra framework of [Gurevich91], since the latter uses a notion of atomic instantaneous action.

We construct, in Section 1, two evolving algebras, reflecting the lower and higher views of Lamport's improved version of the bakery algorithm (see [Lamport79]).

In Section 2 we display abstract conditions on higher level actions, in terms of atomic–action semantics, enabling a simple and concise proof of the first–come–first–served property (FCFS) and deadlock–freedom. The conditions are easily seen to be satisfied by corresponding lower level computations. Since actions of an evolving algebra are assumed there to be atomic, that proof treats the case of atomic reads and writes to shared registers.

In Section 3 we explain the semantics of evolving algebras assuming *durative actions*, actions taking time, and allowing overlapping of reads and writes to shared registers. Refining the abstract conditions for the case of *regular* reads (see [Lamport86]), we show that

[0]To appear in "Specification and Validation Methods", ed. E. Börger, Oxford University Press.

[1]Dipartimento di Informatica, Universita di Pisa, Corso Italia 40, I–56100 Pisa, boerger@di.unipi.it. Partially supported by MURST 91.

[2]EECS, University of Michigan, Ann Arbor MI 48109–2122, gurevich@umich.edu. Partially supported by NSF Grant CCR 92-04742 and ONR grant N00014-91-J-11861.

[3]FSB, University of Zagreb, Salajeva 5, HR–41000 Zagreb, dean@math.hr. Partially supported by C-NR/Gnasaga grant 2.94.

the proof of the previous section goes through with only slight modifications. For the more general case of *safe* registers correctness of the algorithm from [Lamport74] is then easily proved by a slight adaptation of the present argument—the improved algorithm from [Lamport79] is not correct for safe registers, as shown by a simple counterexample.

Thus the two interpretations of evolving algebra dynamics reflect two disciplines for accessing shared registers—by atomic and non–overlapping reads and writes, or by durative and possibly overlapping ones. What really changes is the notion of state: for atomic actions we have global states, whereas for durative actions we have instead local states of agents (see the concept of external and internal locations in section 3.1). The correctness proof however remains essentially the same.

In order to make the paper technically self contained, except for basic notions about evolving algebras of [Gurevich91], we start Section 1 with a review of Lamport's 1979 algorithm and give full details of proofs also in those places where we borrow from [Abraham93].

1 The algorithms

This section presents Lamport's algorithm (taken in a form which is adapted from [Abraham93]), the corresponding 'lower level' evolving algebra, and the more abstract evolving algebra reflecting the 'higher level view'.

1.1 Lamport's algorithm

For arbitrary but fixed N let $P_1, \ldots, P_N$ be processes that may want from time to time to access a 'critical section' CS of code. Any mutual exclusion protocol—which each P_i is supposed to execute in order to enter the critical section—has to prevent two processes from being in the critical section simultaneously. The Bakery Algorithm provides each P_i with a (shared) register R_i and a (private) array $n[1], \ldots, n[N]$ holding natural numbers. Only P_i is allowed to write to R_i but every process can read the register. We assume each register to be initialized with value 0.

The Bakery Algorithm is divided into six consecutive phases: *start*, *doorway*, *ticket* assignment, *wait* section, *critical section* and *finale*. A process P_i starts by declaring its interest in accessing the critical section through writing 1 into its register recording the value written also in its corresponding array variable. In the doorway section, P_i copies all the other registers into its array. It then writes a *ticket*, greater than each number in its array, into its register and into $n[i]$. During the subsequent wait section, process P_i keeps reading, into its array, the registers of each other process P_j, until the resulting array value $n[j] = 0$ or $n[j] > n[i]$ or $n[j] = n[i] \wedge j > i$. Then P_i enters the critical section. Upon leaving CS, as finale, P_i sets its register to 0.

Start

```
n[i] := 1
write(Rᵢ,n[i])
```

Doorway

```
for all j≠i, read(Rⱼ,n[j])
```

Ticket

```
n[i] := 1 + max_j n[j]
write(R_i,n[i])
```

Wait

```
for all j≠i, repeat
   read(R_j,n[j]) until
   n[j]=0 or n[j]>n[i] or (n[j]=n[i] and j>i)
```

Critical Section

Finale

```
write(R_i,0)
```

Note that by ordering pairs of positive integers lexicographically:

$$(i, j) < (k, l) \longleftrightarrow [i < k \text{ or } (i = k \text{ and } j < l)]$$

one can write the until condition as follows: n[j]=0 or (n[j],j)>(n[i],i). The condition assures that, in case two processes get the same 'ticket', the one with smaller identifier gets the priority.

Note also that the for-all commands in the doorway and the wait section may be executed in many ways, in various sequences, all at once, concurrently etc.

1.2 The lower level algebra $\mathcal{B}_1$

As the basis for the subsequent analysis and 'higher level' abstraction, we reformulate here the above Bakery Algorithm as an evolving algebra $\mathcal{B}_1$. It contains, for each process, a *customer–agent*. The customers execute the module with rules Start, Ticket, Entry, Exit, Finale—corresponding to the homonymous Bakery Algorithm phases. In order to preserve the freedom of choosing an ordering of reads, in Doorway and Wait, $\mathcal{B}_1$ contains also *reader–agents* $r(X, Y)$, where X, Y are customers. Each reader–agent $r(X, Y)$ reads, during the doorway and the wait section of X, the register $R(Y)$ of process Y into X's array component $A(X, Y)$, doing the work of the *Doorway* and *Wait* phases. The module of a reader agent has two rules, Read and Check.

Each customer X can execute the rules Start, Ticket, Entry, Exit, Finale only sequentially, in that order; this is assured by the function *mode* which for each X assumes cyclically the values *satisfied, doorway, wait, CS, done, satisfied.* The *mode* function also assures that Ticket and Entry can be executed by X only after all readers have executed their Read and Check rules respectively. Thus the following rules faithfully represent the corresponding phases of the Bakery Algorithm (given that initially all registers $R(X)$ have value 0 and all customers are satisfied).

Customer X

Start

```
if mode(X)=satisfied then
   A(X,X) := 1, R(X) := 1, mode(X) := doorway
```

Ticket

```
if mode(X)=doorway and (∀Y ≠ X) mode(r(X,Y))=wait then
   A(X,X) := 1 + max_Y A(X,Y), R(X) := 1 + max_Y A(X,Y)
   mode(X) := wait
```

Entry

```
if mode(X) = wait and (∀Y ≠ X) mode(r(X,Y)) = doorway then
   mode(X) := CS
```

Exit

```
if mode(X) = CS then
   mode(X) := done
```

Finale

```
if mode(X) = done then
   R(X) := 0, mode(X) := satisfied
```

Reader $r(X, Y)$

Read

```
if mode(r(X,Y))) = mode(X) then
   A(X,Y) := R(Y)
   if mode(r(X,Y))=doorway then mode(r(X,Y)) := wait
   if mode(r(X,Y))=wait then mode(r(X,Y)) := check
```

Check

```
if mode(r(X,Y)) = check then
   if A(X,Y)=0 or (A(X,Y),id(Y)) > (A(X,X),id(X)) then
      mode(r(X,Y)) := doorway
   else mode(r(X,Y)) := wait
```

The modules of rules are written as templates, i.e. there is a module for each customer X and a module for each reader $r(X, Y)$.

1.3 The higher level algebra $\mathcal{B}_2$

In this subsection we define an evolving algebra expressing a 'higher level' view of the Bakery Algorithm. The relevant datum to be described abstractly is the *ticket* assigned to a customer X (and written into its register $R(X)$) when X leaves the doorway and enters the wait section. We introduce for this purpose an external function T whose values are determined dynamically by the outside world, cf. [Gurevich91].

The relevant moment to be analyzed is the moment at which a process which has received a ticket is allowed to enter the critical section. This 'permission to go' will also be represented by an external function, *Go*.

In subsequent sections we will impose conditions upon T and Go which will be shown to guarantee the correctness of the higher level Bakery Algebra.

The higher level algebra has only one module, parametric in a customer X, which has five rules. Again, we assume that initially all registers have value 0 and all customers are satisfied.

Start

```
if mode(X) = satisfied then
   R(X) := 1, mode(X) := doorway
```

Ticket

```
if mode(X) = doorway then
   R(X) := T(X), mode(X) := wait
```

Entry

```
if mode(X) = wait and Go(X) then
   mode(X) := CS
```

Exit

```
if mode(X) = CS then
   mode(X) := done
```

Finale

```
if mode(X) = done then
   mode(X) := satisfied, R(X) := 0
```

2 Atomic actions interpretation

2.1 Semantics of $\mathcal{B}_1$

We rely on the notion of run of [Gurevich94], specialized to *real time*.

This means that we shall speak of a move (rule execution) taking place at moment a. Since we consider atomic actions here, we assume moves to take zero time. Each move is performed by an agent (a customer or a reader) and, since agents are sequential, two moves by the same agent cannot take place at the same moment. For any moment a the set of all moves taking place not later than a is finite (let us call this property 'cofiniteness'). The state (static algebra) $\mathcal{S}_b$ at time b is the one resulting from all moves taking place before b. We shall denote the value a term t takes (in the state) at time b by t_b.

If a move is executed at time b, $\mathcal{S}_b$ is the state in which the move is executed; for some sufficiently small ϵ, $\mathcal{S}_{b+\epsilon}$ is the state resulting from the move. We do not allow (in this section) to read from and write to the same location at the same time. We assume that no module stalls forever; eventually it makes a move (provided a move is enabled all the time). There is one exception: customers are allowed to remain in mode *satisfied* forever.

We now define intervals of (real) time characterized by the moments of successive executions, by a process X, of its rules Start, Ticket, Entry, Exit.

Definition 2.1 Suppose X executes Start and Ticket rules at moments a and b and does not execute anything in between. Then the open interval $x = (a, b)$ is a *doorway* of X. If b is the last execution of X then the *wait interval* $W(x) = (b, \infty)$ and the *CS interval* $CS(x)$ is undefined. Suppose that the execution of Ticket rule at b is followed by executions of Entry rule at c and Exit rule at d. Then $W(x) = (b, c)$ and $CS(x) = (c, d)$). $\square$

By the assumption that no module stalls forever, every doorway is finite. This is in accordance with the fact that in the low–level Bakery Algebra, $T(x)$ is always defined when interpreted as $1 + \max_Y A(X, Y)$.

2.2 Semantics of $\mathcal{B}_2$

The semantics of $\mathcal{B}_2$ is similar to that of $\mathcal{B}_1$. There are no readers around. The definition of doorways and related periods applies also to $\mathcal{B}_2$.

Contrary to $\mathcal{B}_1$, $\mathcal{B}_2$ has external functions, namely T and Go. We are going to impose some constraints on them. To avoid repetitive case distinctions for processes which (being satisfied) have register 0, and of processes which happen to receive the same ticket, we introduce the following notation. If f is a function from the original processes to natural numbers, let

$$f'(X) = \begin{cases} N \cdot f(X) + \mathrm{id}(X), & \text{if } f(X) > 0; \\ \infty, & \text{otherwise.} \end{cases}$$

We assume that the identifiers of the N processes are natural numbers $< N$.

For real intervals I, J we define $I < J$ to mean that $a < b$ for all $a \in I$, $b \in J$. This ordering will help us to formalize the idea that tickets increase together with doorways (see $C1$ below). This should also apply in a way to overlapping doorways; these are ordered by the following relation $\prec$, borrowed from [Abraham93].

Let $X \neq Y$, x ranges over doorways of X, y ranges over doorways of Y.

Definition 2.2 $x \lhd y$ if $x \cap y \neq \emptyset$ and $T'(x) < T'(y)$. Further, $x \prec y$ if $x \lhd y$ or $x < y$. $\square$

Lemma 2.1 *$x \prec y$ or $y \prec x$.*

Proof Note that $T'(y) \neq T'(x)$ for $X \neq Y$. $\square$

Constraints on T and Go

C0 $T(x)$ is a positive integer > 1.

C1 If $y < x$ then either $\mathrm{CS}(y) < \sup(x)$ or $T'(y) < T'(x)$.

C2 If Go(X) holds at moment $t > \sup(x)$ then, for every $Y \neq X$, there exists a moment $b \in W(x)$ such that $T'(x) < R'_b(Y)$.

C3 If $W(y)$ is finite for all $y \prec x$, then $W(x)$ is finite.

Intuitively, C1 says that tickets respect the temporal precedence of doorways with overlapping wait periods, C3 is an induction principle, and C2 expresses that permission to go is obtained by checking the ticket against competitors' registers.

2.3 $\mathcal{B}_1$ implements $\mathcal{B}_2$ correctly

We check that the constraints are satisfied in the first algebra, where $T(X) = 1 + \max_Y A(X,Y)$, and $Go_t(X)$ means that the condition of the rule Entry is satisfied at moment t.

C0 is satisfied since the maximum in the rule *Ticket* is taken over each Y, including X which at that moment has register value $R(X) = 1$.

C1. Let t be the time of the Read move by $r(X,Y)$ during x. If there exists a Finale move by Y during $(\sup(y), t)$, then $\mathrm{CS}(y) < \sup(x)$. Otherwise $R_t(Y) = T(y)$ and therefore $T(x) \geq 1 + R_t(Y) > T(y) > 0$. Hence $T'(x) > T'(y)$.

C2. $Go(X)$ becomes true in $\mathcal{B}_1$ when all readers $r(X,Y)$ finish their wait-section readings. Fix a $Y \neq X$ and consider the last Read move by $r(X,Y)$ during $W(x)$. In view of the corresponding Check move, the time of that Read move is the desired b.

C3. By contradiction, suppose that the premise is satisfied but the conclusion is false, i.e. $W(x)$ is infinite.

Claim: There is a moment $b \in W(x)$ so late that the following two properties hold for each y:

$$\text{(i) if } y \prec x \text{ then } b > sup(CS(y)), \qquad \text{(ii) if } x \prec y \text{ then } b > sup(y).$$

Given the claim, it suffices to prove that any $r(X,Y)$ finishes its reading during $W(x)$ (in contradiction to the assumption that $W(x)$ is infinite). If $r(X,Y)$ finishes its reading before b, we are done since $b \in W(x)$. Otherwise, by definition of b, no $Y \neq X$ can be in mode doorway at or after b. Thus, at or after b, $r(X,Y)$ can read either 0 or $T(y)$ for some $y \succ x$. In the first case the next Check of $r(X,Y)$ will succeed; in the second case it will also succeed, since $T'(y) > T'(x)$ (by C1 if $x < y$, and by definition of $\lhd$ if $x \lhd y$). Thus, the very first reading at or after b will be the last reading of $r(X,Y)$.

To prove the claim, note that, by the cofiniteness condition of runs, there are only finitely many doorways y coming earlier than or overlapping with x. Note that, for $y \prec x$, $sup(CS(y)) < \infty$ by the assumption that $W(y)$ is finite and that no module stalls forever. It suffices to prove that, for each Y, there is at most one $y > x$. Suppose $x < y$. Then, by C1, $T'(x) < T'(y)$ (since $W(x)$ is infinite), and Y remains waiting forever, i.e. $r(Y,X)$ keeps forever executing waiting section Reads.

2.4 Correctness and fairness of $\mathcal{B}_2$

Lemma 2.2 (FCFS) *If $y \prec x$ and $W(x)$ is finite, then $W(y)$ is finite and $CS(y) < CS(x)$.*

Proof Assume the premise is satisfied and the conclusion is false. Take b as given by C2.

Claim 1 : $T'(y) < T'(x)$.

Claim 2 : $\sup(y) < b$.

Given the claims, we have $T'(y) < T'(x) < R'_b(Y)$ and thus Y must be writing to $R(Y)$ sometime in $(\sup(y), b)$. But the first such write after $\sup(y)$ must be a Finale move, which contradicts the assumption that the conclusion of the lemma is false.

Claim 1 follows immediately from definition of $\prec$ in case of overlap, and from C1 otherwise.

To prove Claim 2, we first note that $b > \inf(y)$, in view of $y \prec x$. It is impossible that $\inf(y) < b \leq \sup(y)$, since then $R_b(Y) = 1$. □

Lemma 2.3 $\prec$ *is transitive.*

Proof by contradiction. Suppose $x \prec y \prec z \prec x$. Count the number n of $<$'s in the above sequence of $\prec$ signs. In case $n = 0$ the statement follows from the fact that the order of integers (tickets) is transitive, and in cases $n = 2, 3$ the statement follows from the fact that the order $<$ of real intervals is transitive. In case $n = 1$, without loss of generality, we have $x \lhd y \lhd z < x$ and therefore $T'(x) < T'(y) < T'(z)$. By Lemma 2.2, the assumption $x \prec y \prec z \prec x$ implies that $W(x), W(y), W(z)$ are all infinite. Thus we can apply C1 to obtain also $T'(z) < T'(x)$, which is impossible. □

Lemma 2.4 (Deadlock freedom) *Every* $W(x)$ *is finite.*

Proof By cofiniteness condition on runs, $\prec$ is well-founded. Then C3 is precisely the induction principle required to establish the claim. □

This section is summarized in the following

Theorem 2.1 *Doorways are linearly ordered by* $\prec$. *All waiting sections are finite, and* $x \prec y$ *implies* $CS(x) < CS(y)$.

3 Durative actions interpretation

3.1 Semantics of $\mathcal{B}_1$

Let S be an initial state where all customers are in mode *satisfied*, all readers are in mode *doorway*, and all registers $R(X)$ have value 0—the values of A don't matter. We consider runs from S.

A run of $\mathcal{B}_1$ consists of the following:

- A collection M of elements, called *moves*.
- A function $\mathcal{A}$ from M to the set of agents. $\mathcal{A}(\mu)$ is the agent that makes the move μ.
- A function P that associates a nonempty finite open time interval with each move. $P(\mu)$ is the execution period of μ. No move can last forever.

However, not every triple $(M, \mathcal{A}, P)$ is a run. The following conditions 1–6 should be satisfied. The first condition reflects the fact that our agents are sequential:

1 For each agent X, $\{P(\mu) : \mathcal{A}(\mu) = X\}$ is linearly ordered by $<$. Moreover, this ordered set is isomorphic to an initial segment of positive integers, and if it is infinite then $\sup_\mu P(\mu) = \infty$.

We say that an agent Z is passive at moment t (resp. in interval I) if t does not belong to (resp. I does not intersect) the period $P(\mu)$ of any move of Z. We would like to insure that X has a well defined state $\mathcal{S}_t(X)$ at every passive moment t of X.

2 If $[a, b]$ is a passive interval of an agent X then $\mathcal{S}_b(X) = \mathcal{S}_a(X)$.

To insure that condition 2 is satisfied, we stipulate the following.

A customer X. Locations of dynamic functions internal to X: mode(X), $A(X, X)$ and $R(X)$. Locations of dynamic functions external to X:
mode($r(X, Y)$) and $A(X, Y)$ where $Y \neq X$.

A reader $r(X, Y)$. Internal locations: mode($r(X, Y)$) and $A(X, Y)$. External locations: mode(X), $A(X, X)$ and $R(Y)$.

States of an agent reflect only the values of internal locations. Notice that every location of any function is internal to some agent.

Call a move μ of an agent X *atomic* with respect to an external location ℓ if ℓ is not updated during $P(\mu)$. A move μ is *atomic* if it is atomic with respect to all its external locations. An agent is *atomic* if all its moves are so.

3 If an agent X makes an atomic move μ and $P(\mu) = (a, b)$ then $\mathcal{S}_b(X)$ is the result of executing one step of X at $\mathcal{S}_a(X)$. (See [Gurevich94] for the definition of the result of a one-step execution of a sequential evolving algebra at a given state.)

4 All customers are atomic. All Check moves of readers are atomic. All Read moves of any $r(X, Y)$ are atomic with respect to mode(X).

Read moves of a reader $R(X, Y)$ may be non atomic with respect to $R(Y)$. We adopt Lamport's notion of regular reads (with a different but equivalent definition):

5 Suppose that (a, b) is the period of a Read move μ by a reader $Q = r(X, Y)$. The value of $A(X, Y)$ in state $\mathcal{S}_b(Q)$, at passive moment b of Q, is nondeterministically chosen among the values of $R(Y)$ at moments t satisfying at least one of the following conditions:

- t is Y's last passive moment $\leq a$,
- t is one of Y's passive moments in (a, b),
- t is Y's first passive moment $\geq b$.

Let $\xi(\mu)$ be the chosen moment t.

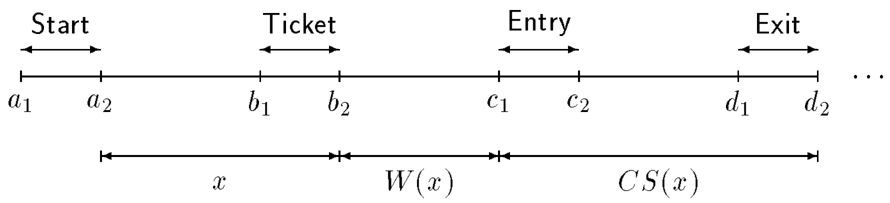
6 If an agent Z has an infinite passive interval during which it is enabled in its final state then Z is an original agent and its mode is *satisfied.*

In other words, we assume again that no agent stalls forever except if it is an original agent in mode satisfied. We will use the following refined definition of doorway, wait and CS sections.

Definition 3.1 • Suppose X executes Start during (a_1, a_2) and then executes Ticket during (b_1, b_2), so that the interval $[a_2, b_1]$ is passive for X. Then $x = (a_2, b_2)$ is a *doorway* of X.

- Suppose that X executes Ticket during (b_1, b_2). If the execution of Ticket is not followed by an execution of Entry then the *wait period* $W(x)$ is (b_2, ∞). Suppose that the execution of Ticket is followed by an execution of Entry during some period (c_1, c_2), so that the interval $[b_2, c_1]$ is passive for X. Then $W(x) = (b_2, c_1)$.
- Suppose that X executes Entry during (c_1, c_2) and then executes Exit during (d_1, d_2), so that the interval $[c_2, d_1]$ is passive for X. Then the *critical section period* $CS(x)$ is (c_1, d_2).

$\square$



3.2 Semantics of $\mathcal{B}_2$

The semantics of $\mathcal{B}_2$ is similar to that of $\mathcal{B}_1$. The constraints C0, C1 and C3 of the previous section remain the same, while C2 is refined for regular registers as follows.

C2 If Go(X) holds at moment $t > \sup(x)$ then, for every $Y \neq X$, there exists a passive moments b for Y such that $T'(x) < R'_b(Y)$ and one of the following holds:

either $b \in W(x)$;

or b is the last passive moment of Y which is $\leq \inf(W(x))$;

or b is the first passive moment of Y which is $\geq \sup(W(x))$.

3.3 $\mathcal{B}_1$ implements $\mathcal{B}_2$ correctly

The proofs that C0 and C3 hold of $\mathcal{B}_1$ remain the same; the proofs for C1 and C2 are modified as follows.

C1. Let μ be the Read move by $r(X, Y)$ during x and $t = \xi(\mu)$. If there exists a Finale move ν by Y such that $P(\nu)$ intersects $P(\mu)$, then $\mathrm{CS}(y) < \sup(x)$. Otherwise $R_t(Y) = T(y)$ and therefore $T(x) \geq 1 + R_t(Y) > T(y) > 0$. Hence $T'(x) > T'(y)$.

C2. $Go(X)$ becomes true in $\mathcal{B}_1$ when all readers $r(X, Y)$ finish their wait-section readings. Fix a $Y \neq X$ and consider the last Read move ν by $r(X, Y)$ during $W(x)$. The desired b is $\xi(\nu)$.

3.4 Correctness and fairness of $\mathcal{B}_2$

All proofs of the previous section remain, except for the proof of Lemma 2.2, which is modified as follows.

Proof Assume premise is satisfied and conclusion is false. Take b as given by C2.

Claim 1 : $T'(y) < T'(x)$.

Claim 2 : $\sup(y) < b$.

Given the claims, we have $R'_{\sup(y)}(Y) = T'(y) < T'(x) < R'_b(Y)$ and thus Y must be writing to $R(Y)$ somewhere in $(\sup(y), b)$ so that this write starts before $\sup(W(x))$. But the first such write after $\sup(y)$ must be a Finale move, which contradicts the assumption that the conclusion of the lemma is false.

Claim 1 follows immediately from definition of $\prec$ in case of overlap, and from C1 otherwise.

To prove Claim 2, we first establish that $b \geq \inf(y)$. Since $\inf(y)$ is a passive moment of Y such that $\inf(y) < \sup(x) = \inf(W(x))$ (in view of $y \prec x$), so $b < \inf(y)$ could not be the last passive moment of Y which is $\leq inf(W(x))$. Neither can we have $\inf(y) \leq b < \sup(y)$, since then $R_b(Y) = 1$. Finally $b \neq \sup(y)$, since otherwise we would have $R_b(Y) = T(y)$, contradicting Claim 1. □

3.5 Counterexample for safe registers

The following example shows that the algorithm of [Lamport79] is not correct for the more general case of *safe* registers (see [Lamport86])—where a read overlapping with a write may get any admissible value whatsoever.

There are two customers X and Y which act at the indicated times as follows:

12.00–12.05:	X and Y both write 1 into their registers and the array
12.05–12.10:	Y reads 1 from $R[X]$
12.10–12.40:	Y writes ticket 2 into $R[Y]$ and the array
12.15–12.20:	X reads from $R[Y]$ getting (by overlap) 17
12.25–12.30:	X writes ticket 18 to $R[X]$ and the array
12.30–12.35:	X reads $R[Y]$ getting 117 (by overlap)
12.45–12.50:	Y reads 18 from $R[X]$
13.00:	X and Y both go to CS

It is however easy to adapt the present proof to show correctness of the algorithm of [Lamport74] for safe registers, rephrased as an appropriate evolving algebra, using the same abstract conditions C0–C3.

Acknowledgement. We thank Uri Abraham for useful discussions.

References

[Abraham93] Uri Abraham, *Bakery Algorithms*, Manuscript. 1993, pp.35

[Gurevich91] Yuri Gurevich, *Evolving Algebras: A Tutorial Introduction*, EATCS Bulletin 43, February 1991, pp. 264–284. A slightly revised version in "Current Trends in Theoretical Computer Science", Eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292.

[Gurevich94] Yuri Gurevich, *Evolving Algebra 1993: Lipari Guide*, this volume.

[Lamport74] Lesli Lamport, *A New Solution of Dijkstra Concurrent Programming Problem*, in *Comm. ACM*, vol.17:8 (1974), 453–455.

[Lamport79] Leslie Lamport, *A New Approach to Proving the Correctness of Multiprocess Programs*, in: *ACM Transaction on Programming Languages and Systems*, vol. 1:1, July 1979, 84–97.

[Lamport86] Leslie Lamport, *On Interprocess Communication*, in *Distributed Computing*, vol. 1, 77–101.

Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras*

Egon Börger
Dipartimento di Informatica
Università di Pisa
56125 Pisa, Italy
boerger@di.unipi.it

Uwe Glässer
Heinz Nixdorf Institut
Universität-GH Paderborn
33098 Paderborn, Germany
glaesser@uni-paderborn.de

July 19, 1995

Abstract

This is a tutorial introduction into the evolving algebra approach to design and verification of complex computing systems. It is written to be used by the working computer scientist. We explain the salient features of the methodology by showing how one can develop from scratch an easily understandable and transparent evolving algebra model for PVM, the widespread virtual architecture for heterogeneous distributed computing.

Introduction

In 1988 Yuri Gurevich has discovered the notion of *evolving algebra* in an attempt to sharpen Turing's thesis by complexity theoretic considerations (see [22]). Through numerous case studies (see [4] for an annotated list which is complete up to 1994) it has become clear since then that using the notion of evolving algebras one can develop a powerful and elegant specification methodology which has a huge yet unexplored potential for industrial applications. In this report we are going to explain the basic concepts of this approach to the design and analysis of complex systems and illustrate its salient features through a challenging example from real life.

We provide an abstract formal specification of central components of PVM, the widespread virtual architecture for heterogeneous distributed computing [20, 32]. We start from scratch without presupposing any knowledge of PVM; the fact that we will nevertheless end up with a clear and easily understandable

*Preliminary versions of this paper have been presented by U. Glässer at the PVM Users' Group Meeting, Oak Ridge, TN, USA, May, 1994 and by E. Börger at the Euro–PVM Users' Group Meeting, Rome, Oct., 1994.

formal definition of the main features of PVM's *virtual machine* at the level of the user interface[1] is not the worst argument for the specification methodology.

It is for the benefit of the practitioner that we are going to introduce here the notion of evolving algebra by a non–toy example from real–life; for a precise mathematical foundation we refer the interested reader to [23]. Our exposition is centered around what we consider to be the four outstanding merits of the proposed approach to the specification and verification of complex systems:

- the *freedom of abstraction* which allows you to structure a system into a hierarchy of appropriate subsystems,
- the powerful and simple mechanism for *information hiding* and definition of *precise interfaces*,
- the principle of *locality for dynamics* (state transitions), and
- the satisfactory link to application domains by the construction of *appropriate ground models*.

1 Freedom of Abstraction

It is well known that general abstraction principles are needed to cope with the complexity of large systems. For data structures the algebraic specification approach (see [33]) shows a way to deal with abstract data types; for actions the action semantics approach (see [30]) proposes a scheme for constructing complex operations out of basic components. Evolving algebras offer the possibility to choose both, the data and the basic actions, at any level of abstraction and independently of each other. The way how this is done is very simple and corresponds to common practice in systems engineering: when specifying a software or hardware system one has to define its basic *objects* and the elementary *operations* which the system uses for its actions (dynamical behaviour). In other words one has to define the basic domains and functions of a system. This leads in a natural way to the mathematical notion of *structures* as formalization of system *states*, as we are going to explain now.

1.1 Universes

Each system S deals with certain basic *objects* which might be classified into different categories. This is reflected in an evolving algebra model of S by corresponding *sets* (also called universes or domains), one for each category of objects. These sets can be completely abstract—this is the case if no restriction is imposed on the corresponding category of objects. In case that the objects are assumed to have certain properties or to be in certain relations with other objects, we formalize these properties and relations by corresponding

[1]More precisely, we specify the C–interface of the virtual machine of PVM 3, the current version of PVM [19].

conditions (integrity constraints) which the objects in those sets are required to satisfy. The evolving algebra approach accepts any precise formalization of such conditions, in whatever language or framework they are given.

In the remaining part of this subsection we illustrate this data abstraction principle by a discussion of basic PVM domains.

Under PVM a heterogeneous collection of physically interconnected Unix–based machines of various kinds of architectures (including serial, parallel, and vector computers) appears logically as a loosely coupled distributed–memory computer. The constituting member computers are called *host* machines and are formalized as elements of a corresponding set $HOST$. The architecture of each host machine is indicated by a function

$$arch : HOST \rightarrow ARCH$$

where $ARCH$ is the set of possible architectures to be used with PVM 3 as listed in [19]. An important feature which distinguishes the universe $HOST$ from $ARCH$ is that the latter is static—i.e. it does not change—whereas the former is dynamic. Indeed, host machines can be dynamically added to or deleted from the virtual machine—except for a designated host

$$master : HOST$$

on which PVM is started and which maintains the control over the dynamically changing machine configuration. Since the main intention of the concept of evolving algebras is to reflect the dynamical system behaviour in a direct and simple way, there are two basic operations which formalize the growing and shrinking of universes:

$$\textbf{extend } A \textbf{ by } x_1, \ldots, x_n \textbf{ with } \; - \; \textbf{ endextend}$$

where '—' is used to define certain properties or functions for (some of) the new objects x_i of the universe A. For example, if a new host machine of architecture type at has to be added—this can be obtained in PVM using the routine `pvm_addhosts()`—one can write

$$\textbf{extend } HOST \textbf{ by } x \textbf{ with } arch(x) := at \;\; \textbf{endextend} \; .$$

The corresponding deletion operation has the following form

$$\textbf{discard } t \;\; \textbf{from } A$$

where t is a first–order term[2] and A a universe. One might be tempted to assume that once t is deleted from A, each function where t appears as value or in the arguments is automatically undefined there; but since this is more a type–checking or implementation concern (about error detection or garbage collection) than a

[2]By *first–order term* we mean any expression built up from constants or variables by using function symbols.

semantical concern, we assume here only that '**discard** t **from** A' has the effect of setting the characteristic function of A at t to false.

The basic computational units of PVM are constituted by concurrently running application programs (Unix processes), which can enroll into PVM as *tasks*. We formalize these tasks as elements of a dynamic domain

$$TASK \subset PROCESS$$

which is required to consist of the processes enrolled into PVM[3]. When a process enrolls into PVM (see below the PVM instruction `pvm_mytid()`) it is registered by a new task identifier, i.e. an appropriate set TID—which is left abstract here—is extended by a new element which is assigned to the given process, say $Process$, by an injective function tid as follows:

$$\textbf{extend } TID \textbf{ by } Tid \textbf{ with } tid(Process) := Tid \ \textbf{ endextend}$$

Note that also the inverse function:

$$task : TID \rightarrow TASK$$

of tid is used. By requiring $task$ to be the inverse function of tid we avoid to have to explicitly set $task(TID) := Process$ when $tid(Process)$ is set to Tid. This formalizes the global addressing of tasks in PVM.

On the basis of this homogeneous global address space TID in which all tasks are uniformly addressed through tid, when tasks communicate to each other they do not need to know whether their communication partner resides on the same host or not. Indeed, the asynchronous message–passing model of PVM does not distinguish between local and global intertask communication. However, on each of the hosts there is a local daemon process, called $pvmd$, which acts as a local supervisor in operations that require task management or intertask communication. We formalize this by introducing another dynamic set $DAEMON$ together with a nullary function $pvmd : DAEMON$ identifying the concurrently operating PVM daemons. The correspondence between daemons and their hosts is expressed by a dynamic bijective function:

$$host : DAEMON \rightarrow HOST.$$

Clearly, each time a host is added, a new daemon has to be created and the function $host$ to be updated correspondingly. Thus the above $HOST$–extension is refined as follows:

$$\begin{array}{l}
\textbf{extend } HOST \textbf{ by } x \textbf{ with}\\
\quad arch(x) := at\\
\quad \textbf{extend } DAEMON \textbf{ by } d \textbf{ with}\\
\quad\quad host(d) := x\\
\quad \textbf{endextend}\\
\textbf{endextend}
\end{array}$$

[3]Note that there are also other processes which are not running under PVM, for instance, the PVM system itself is such a process.

For the initialization we have to require that there is a distinguished daemon

$$demiurge : DAEMON$$

who resides on the master host (i.e. $host(demiurge) = master$) and will in particular be responsible for creation and deletion of hosts.

1.2 Dynamic Functions

Once it has become clear what are the basic objects of a system S, one has to think about what are the elementary operations which are performed on those objects in S. Typically, a basic operation consists of setting a certain value, given the values of certain parameters. The most general framework of such operations is the following *function update*:

$$f(t_1, \ldots, t_n) := t$$

where f is an arbitrary n–ary function and $t_1, \ldots, t_n$ represent the parameters (arguments) at which the value of the function is set to t. In writing evolving algebras it is allowed to use arbitrary function updates, for functions f and terms t_i, t of arbitrary complexity or level of abstraction. Note that such functions f are called *dynamic* in contrast to *static* functions which do not change.

Function updates provide the basic notion of *destructive assignment at any level of abstraction*. Here are two very simple and well known but characteristic examples for function updates. The updating of a program counter in an architecture is expressed by the following function update

$$pc := next(pc)$$

where pc represents a 0–ary function and $next$ the monadic function which determines for a given value of pc what is the next value to be stored in pc. The updating of a buffer by adding a new datum can be expressed by the following function update:

$$cont(buffer) := append(datum, cont(buffer))$$

where *buffer* is a 0-ary function, *cont* a monadic function providing the buffer content and *append* an abstract function which given a datum and a buffer content yields the result of appending this datum to the current content of the buffer.

Besides domain extension and deletion of elements, function updates are the mechanism by which the dynamics of arbitrary systems can be described in an explicit way. In accordance with usual practice the execution of updates in evolving algebras can be conditioned by so–called guards, giving rise to so–called *transition rules* (also called *guarded multi–update instruction* in [23]) of the following form:

$$\textbf{if } Cond \textbf{ then } Updates$$

Cond is an arbitrary boolean valued expression (first–order logic formula) and *Updates* a set of updates. The rule can be executed if *Cond* is true; its effect is to simultaneously execute each update in the set *Updates*. (The simultaneous execution of more than one update helps to avoid an explicit description of intermediate storage, see for example the two updates $a := b, b := a$.)

A simple example from PVM is provided by the routine **pvm_mytid()**. If this routine is called by a process *Process*, the responsible *pvmd* checks whether *Process* is among the tasks which are registered already—the set of those tasks is stored by a dynamic function *tids* at argument *pvmd*. If *Process* is already registered, its task identifier is returned to *Process*; if it is not registered yet, then it gets enrolled into PVM as described in the preceding subsection and its task identifier is added to *tids*(*pvmd*). Thus we have the following rule:

if $tid(Process) \neq undef$
then *return* $tid(Process)$ *to Process*
else extend *TID* **by** *Tid* **with**
    $tid(Process) := Tid$
    $tids(pvmd) := insert(Tid, tids(pvmd))$
    return Tid to Process
  endextend

Here "*return t to P*" is an abstract update which we are not going to specify furthermore at this level of abstraction.

1.3 States as Algebras

To speak about a system means to speak about its objects in terms of functions and relations defined on them. Domains, functions, and relations constitute what in mathematics is called a *structure*. Structures without relations are traditionally called *algebras*. Since relations (and in particular sets) can be represented by their characteristic functions, for simplicity, we deal in the following only with algebras.

An *evolving algebra* can (in a first approximation) be defined as a finite set of transition rules

if *Cond* **then** *Updates*

where *Updates* consists of finitely many function updates (and domain extensions or deletions of elements; these two update forms can be reduced to function updates, see [23]). The effect of a transition rule R when applied to an algebra $\mathcal{A}$ is to produce another algebra $\mathcal{A}'$ which differs from $\mathcal{A}$ by the new values for those functions at those arguments where the values are updated by the rule R.

We will construct the PVM rules in such a way that the guards imply consistency of simultaneously executable updates. Note that no rule changes the type of the functions; only the incarnation (the concrete interpretation) of a function changes by changing some of its values. We speak therefore of algebras also as *static* algebras, to distinguish them from evolving algebras. The latter are transition systems which transform the former.

Thus, the abstraction principle which is built into the notion of evolving algebra consists in proposing

- (static) algebras as the mathematical notion of "state" and
- guarded destructive assignments for abstract functions as basic dynamic operations

This is the most general notion of state and of dynamic changes of states modern mathematics offer. As a consequence evolving algebras are the most general notion of a (discrete) dynamic system. A priori no restriction is imposed on the abstraction level where one might want to place an evolving algebra description of a system. This freedom explains the success of the simple and transparent evolving algebra models for the semantics and the implementation of numerous complex programming languages like Prolog [15, 16], C [24], VHDL [11], Occam [9, 8], for protocols [12, 27], architectures [6, 5, 13], real–time algorithms [25, 26], etc.

The importance of the freedom of abstraction which is offered through the notion of evolving algebra is also confirmed by a common experience in the design of algorithms. Namely, the need to model phenomena of the real world, which are given a priori, leads the designer of programs to use 'abstract structures', as has been well expressed a long time ago by N. Wirth:

> *"... Data in the first instance represent abstractions of real phenomena and are preferably formulated as abstract structures not necessarily realized in common programming languages."* (see [34], page 10)

Furthermore, the reciprocal dependency of algorithms and data structures makes it important for the designer not to be hindered by inappropriate restrictions of the framework; to say it again with Wirth's words:

> *"It is clear that decisions about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often depend strongly on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably interwined."* (see [34], page 9)

The framework of evolving algebras offers the freedom the designer needs to 'tailor' his models to the given level of abstraction.

2 Information Hiding and Interfaces

The basic idea of *information hiding*, as introduced by D. Parnas in [31], addresses the modular structuring of systems and can be summerized as follows:

> *"A module achieves program simplification by providing an abstraction. That is, its function can be understood through its interface definition without any need to understand the internal details."* [29].

In a practical specification methodology information hiding has to go hand in hand with a good discipline to handle interfaces. The evolving algebra approach offers both in a most general way through the concept of *external functions*.

Each function f which appears in an update $f(t_1, \ldots, t_n) := t$ of some transition rule R is called internal for R; for a given evolving algebra $\mathcal{A}$ a function g of the vocabulary of $\mathcal{A}$ which is not internal for any of the rules of $\mathcal{A}$ is called external for $\mathcal{A}$.

For a function f which is internal for $\mathcal{A}$ all the information on (the dynamical behaviour of) f is available through the rules of $\mathcal{A}$; the programmer who knows the rules of $\mathcal{A}$ can use that information about f. In contrast, for an external function g for $\mathcal{A}$ the rules of $\mathcal{A}$ give no information on the behaviour of g. An external function g cannot be modified ('written') by $\mathcal{A}$, but it can be used ('read') in the rules of $\mathcal{A}$ to determine arguments at which an internal function is changed dynamically or to determine the new values in such updates. External functions can in particular be used to represent the environment in which an evolving algebra is intended to work. It is the task of the system designer to provide exactly that information on g which he wants the programmer to know and to use. In the evolving algebra methodology this interface information can range from nothing at all—this is the case of an external function for which only the number and the types of its arguments and values are known—to a full specification by some axioms or by a set of equations or by another evolving algebra (module), etc. Note that due to the abstraction principle explained in the previous section the evolving algebra approach imposes no restriction at all on the choice of external functions and the way they are described. The use of evolving algebras does not trivialize the difficult task of "designer control of the distribution of information" ([31]:pg. 344), but at least it does not hinder this task by extraneous formal overhead and offers a flexible and open framework to guarantee information hiding and the definition of precise abstract interfaces.

In the following subsections we explain some outstanding examples for the use of external functions in evolving algebra descriptions of complex systems.

2.1 The PVM Event Mechanism

PVM realizes a distributed computation model which is characterized by the *reactive behaviour* of the concurrently operating PVM daemon processes, one on each host computer of the virtual machine. The daemons effectively carry out the PVM instructions of their local tasks and may interact with each other through asynchronous message–passing communication. There are in principle two different kinds of operations requiring the activity of a daemon (pvmd): a request of a local task to carry out some PVM instruction and the reception of a message from another pvmd. The daemon cannot influence from where, when, and which request or message will reach him, rather he has to wait for the next such event to come whenever he is idle. We can model this intuition faithfully by introducing an external function *event* which for some given pvmd might yield a PVM instruction or message as value. If *event(pvmd)* is defined and has the value *instr/mssg*, then the pvmd is going to execute/read *instr/mssg*. This is formalized in our PVM model by a rule of form

$$\textbf{if}\ \ event(pvmd) = instr/mssg\ \ \textbf{then}\ \ execute\_instr/read\_mssg$$

for each individual PVM instruction *instr* or PVM message *mssg*, where *execute_instr/read_mssg* represents the corresponding updates. An integrity constraint on the function *event* is that a defined value of *event(pvmd)* remains stable until the pvmd has evaluated the function. However, we assume 'destructive reading' such that *event(pvmd)* is reset to *undef* (resp. indicates the next event) as soon as the pvmd has read the current value.

As example we define the four rules which define the task administration and administration of message buffers in PVM.

2.1.1 PVM Task Administration

We complete the formalization of the PVM routine `pvm_mytid()` by the following rule defining the reaction of the pvmd when the routine is called by a given process *Process*:

`pvm_mytid()`

> **if** *event(pvmd)* = *mytid() from Process*
> **thenif** *tid(Process)* $\neq$ *undef*
>   **then** *return tid(Process) to Process*
>   **else extend** *TID* **by** *Tid* **with**
>   *tid(Process)* := *Tid*
>   *tids(pvmd)* := *insert(Tid, tids(pvmd))*
>   *return Tid to Process*
>   **endextend**

Similary, we have the following rule for the PVM routine `pvm_exit()` by which a *Task* can leave PVM through a request to the responsible pvmd:

`pvm_exit()`

> **if** *event(pvmd)* = *exit() from Task*
> **then**
>   **discard** *tid(Task)* **from** *TID*
>   *tids(pvmd)* := *delete(tid(Task), tids(pvmd))*

2.1.2 PVM Message Buffers

For the specification of PVM's message–passing interface—which offers point-to-point communication from one task to another as well as multicast to a set of tasks—we extend the basic model by abstract domains related to messages and buffers: $MESSAGE$, TAG, $DATA$, $BUFID$, $ENCODING$. To send a message, a task first packs the message into a send buffer and then calls one of the send functions. To selectively receive messages, a task invokes one of various receive functions specifying the *receive context*. Though PVM does not explicitly specify any limit to the size or number of messages, our specification could easily incorporate conditions which reflect constraints coming from physical limitations of the underlying hardware and software components.

The *basic* message-passing routines of PVM apply a simple communication model[4] that is based on two fundamental assumptions: for each task there is only one send buffer and one receive buffer; any message transfer between tasks is handled by the responsible pvmds. The following description of the message-passing model assumes two basic integrity constraints that are guaranteed by the corresponding routines embedded in the virtual machine: message-passing is *reliable* and *order-preserving*.

Message buffers are addressed through unique identifiers from $BUFID$, $0 \notin BUFID$. The content of a buffer may be any sequence of $DATA$ objects and is accessed by $cont : BUFID \rightarrow DATA^*$. An $encoding : BUFID \rightarrow ENCODING = \{PvmDataDefault, PvmDataRaw, PvmDataInPlace\}$, associated to send buffers, specifies the method used for packing messages: *PvmDataDefault* refers to External Data Representation (XDR); *PvmDataRaw* to the original data format; *PvmDataInPlace* means that the data items have to be copied directly out of the user's memory (for details cf. [19]). Two injective functions *sendbuf* and *recvbuf* from $TASK$ into $BUFID$ yield the current send and receive buffers of tasks.

The routine *pvm_initsend*() creates, for the task which wants to start a sending operation, an empty send buffer with the specified encoding scheme and returns a corresponding buffer identifier. A class of routines $pvm\_pk*(Pointer, Nitem, Stride)$—there is one pack routine for each individual data type $* \in \{byte, cplx, dcplx, double, float, int, long, short\}$[5]—packs the number $Nitem$ of data into the send buffer; $Pointer$ refers to the location of the first data item and $Stride$ to the relative distance to the next one. The formal definitions of these routines are given by the rules:

---

[4] or more sophisticated communication mechanisms additional routines and options, not discussed here, allow to tailor the basic model to individual application requirements; for example, this includes direct task-to-task communication, specific group functions, or multiple send and receive buffers.

[5] For packing strings a simpler routine is used which we do not describe here.

`pvm_initsend()`

$$
\begin{array}{l}
\textbf{if } event(x) = initsend(Encoding) \textit{ from } Task \\
\textbf{then} \\
\quad \textbf{extend } BUFID \textbf{ by } b \textbf{ with} \\
\quad\quad sendbuf(Task) := b \\
\quad\quad cont(b) := \langle\rangle \\
\quad\quad encoding(b) := Encoding \\
\quad\quad return\ \langle b\rangle\ to\ Task \\
\quad \textbf{endextend}
\end{array}
$$

`pvm_pk*()`

$$
\begin{array}{l}
\textbf{if } event(x) = pk^*(Pointer, Nitem, Stride) \textit{ from } Task \\
\textbf{then} \\
\quad cont(b)) := append(data^*(Pointer, Nitem, Stride), cont(b)) \\
\\
\textbf{where } b \equiv\ sendbuf(Task)
\end{array}
$$

We will complete the formalization of the PVM message–passing system in Section 3.

2.2 Further Examples of External Functions

The external function *event* in our PVM model is dynamic and thus directly reflects the way in which tasks interact with their local pvmd when they want PVM routines to be invoked. No pvmd has an own fixed PVM program; the instructions or messages which reach a pvmd come for him as determined by his environment to which he reacts. Using the external dynamic function *event* we abstract from the specific way how the daemon's walk through his sequence of instructions/messages is determined by the activities of his tasks.

The power of abstraction which is offered through the introduction of external functions does not depend on their being dynamic. Consider as example the formal definition given in [15] for the programming language Prolog as seen by the programmer. The four simple rules which define the full behaviour of Prolog for user–defined predicates make crucial use of two external functions *procdef* and *unify*.

The function *procdef* is supposed to provide for a given literal l and the given program *db* exactly the clauses in *db* which are relevant for l in the order in which they have to be applied. The whole non–trivial backtracking behaviour of Prolog (including optimizations like determinacy detection) can be described on the basis of such a function *procdef* without being more specific about the latter. *procdef* plays for that model the role of a "well–defined whole." If we consider Prolog without operations like *assert, retract* which modify the program, then *procdef* is a static external function. If we want to model also Prolog's program modification features then *procdef* becomes an internal dynamic function (see [7, 14]). Note that during the refinement process by which the Prolog model of [15] is linked in a provably correct way to the WAM implementation model in [16] the function *procdef* receives an explicit definition.

The function *unify* is supposed in [15] to provide for each pair of literals either a unifying substitution or the information that there is no such unification. This function describes the abstract behaviour of unification without being bound to any concrete unification algorithm. It also hides from the programmer the details about the representation of termes which appear in the refined WAM models of [16]. As a result the abstract PROLOG model of [15] and its refinement to the WAM model of [16] could easily be extended to constraint logic programming languages with or without types where unifiability appears as a particular case of constraints (see the formal evolving algebra definitions of PROLOG III [18], Protos–L [1], and CLP(R) [17]).

Other examples of external functions which contributed in a crucial way to the simplicity of the models under consideration are the following:

- the static find–catcher function defined in [15] leads to a concise formalization of the error–handling predicates *catch* and *throw* of Prolog.

- In the evolving algebra model for the IEEE VHDL Standard we have obtained a simple and uniform rule set for signal assignments by introducing for the inertial delay an external static function *reject* for which we give a natural and easily understandable recursive definition. Similarly, a transparent description is obtained for the propagation of signal values by introducing external functions for the so–called driving and effective values; the former is determined by a recursion on the signal sources, the latter by a recursion on port association elements from ports to signals. In both cases the recursive definitions replace rather complex algorithmic characterizations in the VHDL'93 language reference manual [28].

- In the abstract evolving algebra models of Occam (see [9]) which are the starting point for the correctness proof of a compilation scheme into Transputer instructions in [8] we have taken great advantage of the usual flowchart layout of programs; we define it by external functions which in the later refinement steps are replaced by recursive definitions of the compiling function. Considerable simplifications for both the specifications and the proofs have also been obtained there by leaving the evaluation and compilation of expressions and the implementation of values abstract, realized by appropriately restricted external functions.

- In the more theoretical example constituted by Lamport's mutual exclusion protocol, known as bakery algorithm, a tremendous simplification of the correctness proofs in the literature has been achieved in [12] by introducing two external functions, namely *Ticket* and *Go*, on which three natural conditions and an induction principle are imposed which imply the correctness of the protocol.

In all these cases the external functions allowed us to define a precise interface with respect to which the model under discussion works in a simple and transparent way. Clearly, if for such an abstract model we want to prove general properties about the behaviour of the system where external functions play a role, we have to state and assume their properties which we use. In order to guarantee an unchanged interface behaviour these properties have to be proved to be satisfied when the external functions are defined explicitly or implemented in later refinement steps or modified by changing requirements.

The most general *concept of modularity* which is present through the notion of external functions is deliberately kept open in the definition of evolving algebras. The resulting flexibility in using and dealing with different module structures is an advantage for real–life specification endeavours. Nothing prevents us from restricting this notion to specific and even syntactic concepts of compositionality where the need arises; an example where it turned out to be useful to stick to a simple automaton–theoretic concept of composition of evolving algebras through sequencing, juxtaposition, and feedback can be found in [5].

3 Locality for Dynamics

It is typical for large systems that their overall dynamical behaviour is determined by the dynamic behaviour of their components, i.e. by *local* changes. The evolving algebra methodology allows us to reflect this characteristic interplay between global and local dynamic system behaviour in a direct and faithful way by viewing (global) system states as static algebras and by providing the possibility to express local updates $f(t_1, \ldots, t_n) := t$ in a uniform way at any level of abstraction. This becomes particularly evident with attempts to model distributed systems. We are going to illustrate this through a formalization of the PVM message–passing interface.

Interactions between Pvmds Certain PVM instructions result in distributed operations involving two or more pvmds which perform some interaction through message–passing. We model inter–pvmd communication in the transition rules below by abstract updates of the form

'*forward* $\langle$*RequestMsg*$\rangle$ *to y*' respectively '*return* $\langle$*ReplyMsg*$\rangle$ *to x*'

where x and y refer to the interacting pvmds. They are supposed to trigger corresponding events for the receiving pvmds. For that reason we will have a number of communication related transition rules, distinguished by the suffix '*_req_msg*' for request messages and '*_rep_msg*' for reply messages, describing the reaction of a pvmd x when receiving a message from another pvmd y.

Distributed operations in which a pvmd requests another pvmd for service may produce considerable delays. In order to avoid blocking while waiting for the response to come, the requesting pvmd stores the "wait context" using a structure accessed by a unique *wait-id (wid)* from a domain WID. This wait-id is passed along with the request and returned with the reply. The wait context typically includes information about the requesting task ($req\_info : WID \rightarrow TASK$), the reply data ($rep\_info : WID \rightarrow REPLY^*$), and a request count ($waitcount : WID \rightarrow INT$, indicating the number of replies a pvmd is still waiting for).

The preceding concepts suffice to formalize the status information related to subroutines of PVM, as we are going to do in the next subsection.

3.1 PVM Status Information

PVM offers various kinds of routines providing status information about the virtual machine. To obtain the status of a host, a call of $pvm\_mstat()$ yields: *PvmOk* if the host is running, *PvmHostFail* if it is unreachable, or *PvmNoHost* if this host is not in the virtual machine. Since a host failure is to be considered as an external event, outside the scope of PVM, the resulting effect is modeled using an external dynamic function *hstatus*. When triggered by a requesting pvmd, *hstatus* does not immediately yield a result, as it usually takes some time to perform the distributed operation required to determine the status of a remote host. After a delay—corresponding to the duration of the simulated operation—*hstatus* replies by generating an *ext_hstatus* report event, returning the requested status information to the calling pvmd.

Note that an interaction between a pvmd and the external function *hstatus* appears to the pvmd almost the same way as an interaction with a remote pvmd. That is, it also requires to create a wait context structure, the wid of which is passed over in the function call and returned with the *ext_hstatus* report: $hstatus : HOST \times WID \rightarrow \{running, unreachable\} \times WID$.

To find out whether a particular host is in the virtual machine, a pvmd simply performs a look up on the host table. This test is expressed through the predicate $is\_in\_htable : HOST \rightarrow BOOL$:

pvm_mstat()

> **if** $event(pvmd) = mstat(Host)$ *from* $Task$
> **thenif** $is\_in\_htable(Host)$
>   **thenif** $Host = host(x)$
>   **then**
>   *return* $\langle PvmOk \rangle$ *to* $Task$
>   **else**
>   **extend** WID **by** wid **with**
> $rep\_info(wid) := Task$
>   *trigger* $hstatus(Host, wid)$
>   **endextend**
>
> **else** *return* $\langle PvmNoHost \rangle$ *to* $Task$

`ext_hstatus_report`

$$
\begin{array}{l}
\textbf{if } event(pvmd) = \\
\quad ext\_hstatus\_report\langle Status,\ Wid\rangle \\
\textbf{thenif } Status = running \\
\qquad \textbf{then} \\
\qquad\quad return\ \langle PvmOk\rangle\ to\ Task \\
\qquad \textbf{else} \\
\qquad\quad return\ \langle PvmHostFail\rangle\ to\ Task \\
\\
\textbf{where } Task \equiv\ rep\_info(Wid)
\end{array}
$$

We can now also explain the send and receive instructions of PVM.

3.2 PVM Send Routines

A message consists of receiver and sender tid, an integer tag, and the actual message data; these components are accessible through functions defined on *MESSAGE* with values in *TID* (*recvtid*, *sendtid*), *TAG* (*msgtag*), $DATA^*$ (*msgdata*). Thus each pvmd x holds a sequence $msgseq(x)$ of messages waiting—in the order of their arrival at x—to be received by one of its local tasks. An additional function $msgbuf : MESSAGE \rightarrow BUFID$ is used to hold the addresses of local buffers which are used by the pvmd to store the message data until these messages are delivered to the receiving tasks.

The routine $pvm\_send(Tid, Tag)$ puts the information, stored in the send buffer of the sending task, into a newly created *message* that is sent to *Tid* with label *Tag*. If the daemon of the receiver task is the same as that of the the sender task, sending the message in principle means to enqueue it into the daemon's message queue; otherwise, the message will be forwarded as intertask–message to the remote pvmd identified through *Tid* (causing a corresponding event for that pvmd).

Each time the pvmd x receives a message, it has to check whether the receiver task has specified a *receive context* that matches the labels of the message. The receive context of a task is expressed by a function $expecting : TASK \rightarrow \langle (TID + \{-1\}) \times (TAG + \{-1\})\rangle$ as a combination of options for the message *Tag* and sender *Tid*, where '-1' indicates matching every possible combination. If for some given task t of pvmd x $expecting(t)$ has a defined value, this means that t has been *suspended* on an attempt to receive a message which was not yet available; that is, a message of the specified type was not yet contained in $msgseq(x)$.

Note in the definition of `pvm_send()` that the message data are simply stored in a local buffer in case that sender and receiver work under the same pvmd; otherwise, they are stored as part of the message using the function *msgdata*. This meaning is formally defined by the following rules:

`pvm_send()`

$$
\begin{array}{l}
\textbf{if } event(pvmd) = send(Tid, Tag) \textit{ from } Task \\
\textbf{then} \\
\quad \textbf{extend } MESSAGE \textbf{ by } m \textbf{ with} \\
\quad\quad recvtid(m) := Tid \\
\quad\quad msgtag(m) := Tag \\
\quad\quad sendtid(m) := tid(Task) \\
\quad\quad \textbf{if } pvmd(Tid) = pvmd \\
\quad\quad \textbf{then} \\
\quad\quad\quad Enqueue(m, msgseq(pvmd), Type, Data, Receiver) \\
\quad\quad \textbf{else} \\
\quad\quad\quad msgdata(m) := cont(sendbuf(Task)) \\
\quad\quad\quad \textit{forward } intertask\_msg\langle m\rangle \textit{ to } pvmd(Tid) \\
\quad \textbf{endextend} \\
\\
\quad \textbf{where } \; Type \equiv \langle tid(Task), Tag\rangle, \\
\quad\quad\quad\quad Data \equiv cont(sendbuf(Task)), \\
\quad\quad\quad\quad Receiver \equiv task(Tid)
\end{array}
$$

$$
\begin{array}{l}
Enqueue(mssg, mssgs, type, data, receiver) \\
\equiv \; \textbf{extend } BUFID \textbf{ by } b \textbf{ with} \\
\quad\quad cont(b) := data \\
\quad\quad msgbuf(mssg) := b \\
\quad\quad \textbf{if } matching(type, expecting(receiver)) \\
\quad\quad \textbf{then} \\
\quad\quad\quad recvbuf(receiver) := b \\
\quad\quad\quad expecting(receiver) := undef \\
\quad \textbf{endextend} \\
\quad mssgs := append(mssg, mssgs)
\end{array}
$$

`intertask_msg()`

$$
\begin{array}{l}
\textbf{if } event(pvmd) = intertask\_msg\langle m\rangle \textit{ from } pvmd' \\
\textbf{then } Enqueue(m, msgseq(pvmd), Type, Data, Receiver) \\
\\
\textbf{where } \; Type \equiv \langle sendtid(m), msgtag(m)\rangle, \\
\quad\quad\quad Data \equiv msgdata(m), \\
\quad\quad\quad Receiver \equiv task(recvtid(m))
\end{array}
$$

The corresponding multicast routine $pvm\_mcast()$ can be handled in almost the same way.

3.3 PVM Receive Routines

PVM supports blocking and non–blocking receive routines. Blocking $recv()$ requested from $Task$ sets the receive buffer of $Task$ to the bufid of an expected message and returns bufid to $Task$, if an appropriate message is actually waiting in the message queue; otherwise, the given receive context is used to update the $expecting$ function of $Task$. Non–blocking $nrecv()$ does the same except for returning 0 to $Task$ instead of suspending the task, in case there is no message of the expected type in $msgseq(x)$. Formally this is expressed by two rules, where the function

$$\lceil \ : \ MESSAGE^* \times \langle TID \times (TID + \{-1\}) \times (TAG + \{-1\}) \rangle \rightarrow MESSAGE^*$$

selects, out of $msgseq(x)$, the messages corresponding to $expecting$:

`pvm_recv()`

$$
\begin{array}{l}
\textbf{if } event(pvmd) = recv(Tid, Tag) \ from \ Task \\
\textbf{thenif } msgseq(pvmd) \lceil \langle tid(Task), Tid, Tag \rangle = [\,] \\
\quad \textbf{then } expecting(Task) := \langle Tid, Tag \rangle \\
\quad \textbf{else } RecvMsg(Tid, Tag, Task) Through \ pvmd
\end{array}
$$

`pvm_nrecv()`

$$
\begin{array}{l}
\textbf{if } event(pvmd) = nrec(Tid, Tag) \ from \ Task \\
\textbf{thenif } msgseq(x) \lceil \langle tid(Task), Tid, Tag \rangle = [\,] \\
\quad \textbf{then } return \ \langle 0 \rangle \ to \ Task \\
\quad \textbf{else } RecvMsg(Tid, Tag, Task) \ Through \ pvmd
\end{array}
$$

$$
\begin{array}{ll}
& RecvMsg(Tid, Tag, Task) \ Through \ pvmd \\
\equiv & \textbf{let } msgseq(pvmd) \lceil \langle tid(Task), Tid, Tag \rangle = [m|Mssgs] \\
& return \ \langle bufid \rangle \ to \ Task \\
& recvbuf(Task) := bufid \\
& msgseq(pvmd) := delete \ m \ from \ msgseq(x) \\
& \\
& \textbf{where} \ \ bufid \equiv msgbuf(m)
\end{array}
$$

We conclude this section with two significant examples for the formalization of process control constructs in PVM, namely the creation and the deletion of tasks.

3.4 PVM Task Creation

The routine pvm_spawn() enables dynamic subtask creation. On initiating a spawn operation the spawning *task* becomes *parent* of the (*Ntask* many) new subtasks to be created, each of which is assumed to run a copy of a given executable *File* (possibly with a list of *Arg*uments). The spawning task may affect the selection of hosts to spawn on through a mode parameter: in *transparent* mode tasks are automatically executed on the most appropriate hosts w.r.t. certain load measures; in *architecture-dependent* mode the calling task specifies the architecture; in *low-level* mode it specifies a particular host.

Parameters *Flag* and *Where* are used to specify a combination of options as a sum of: *0 PvmTaskDefault*—PVM chooses where to spawn the processes; *1 PvmTaskHost*—the *Where* argument specifies a particular host to spawn on; *2 PvmTaskArch*— *Where* specifies a type of architecture to spawn on using ARCH; *4 PvmTaskDebug*—starts these processes up under debugger; *8 PvmTaskTrace*—the PVM calls in these processes will generate trace data[6].

In assigning tasks to hosts upon spawning, PVM's in principle non–deterministic choice depends in particular upon the environment (e.g. the operating system) and the internal load balancing scheme (which is transparent to the user). We abstract from details of this complex selection procedure by using a dynamic *external* function *hostselect* which we assume, as integrity constraint, to be consistent with the options set by the user (through *Flag, Where, Ntask*). For the result of *hostselect*, which is a list of the form

$$\langle pvmd_0, n_0 \rangle, \quad \langle pvmd_1, n_1 \rangle, \ldots, \langle pvmd_m, n_m \rangle$$

specifying a collection of pvmds and the number of tasks to be spawned by each of these pvmds, we require that the sum of the n_i, $i = 0, \ldots, m$, is less than or equal to *Ntask* and that $\forall i, j \in \{0, \ldots, m\} : \; i \neq j \Rightarrow pvmd_i \neq pvmd_j$.

The outcome of an attempt to spawn a task which has been assigned to a host depends on the availability of resources, and on whether a suitable copy of the relevant *File* is present on that host. We model the behaviour of the system using again a dynamic external function *try_to_spawn* which provides the necessary *SPAWNREPORT*. Since computing this function may be rather time consuming and thus is not an atomic action, spawning has two phases, modeled by separate rules: a spawn operation is triggered by calling *try_to_spawn* with the appropriate argument values computed using *hostselect*. On completion of this operation *try_to_spawn* returns, to the calling pvmd, the resulting list of pids, one for each successfully created process required to spawn a new task, by generating an *ext_spawn_report* event.

[6] This feature is not yet implemented, cf. [19].

`pvm_spawn()`

if $event(pvmd) = spawn(File, Args, Flag, Where, Ntask)$ *from* $Task$
thenif $hostselect(Flag, Where, Ntask, pvmd) = \langle local, remote\rangle$
  $\&\ local = \langle pvmd, n_0\rangle$
  $\&\ remote = \langle pvmd_1, n_1\rangle, \ldots, \langle pvmd_m, n_m\rangle$
 then
  let $tid = tid(Task)$
  extend WID **by** wid **with**
   spawning_task(wid) := tid
   tasks_to_be_spawned(wid) := $\langle\rangle$
   $waitcount(wid) := m + signum(n_0)$

   if $n_0 > 0$
   then *trigger* $try\_to\_spawn\langle \text{Params}, n_0, wid\rangle$
   var i **ranges over** $\{1, \ldots, m\}$
    if $m > 0$
    then *forward* $spawn\_req\_msg\langle \text{Params}, n_i, wid, tid\rangle$ *to* $pvmd_i$
  endextend

 where $\text{Params} \equiv File, Args, Flag, Where$

`ext_spawn_report⟨⟩`

if $event(pvmd) = ext\_spawn\_report\langle Wid, \langle Pid_1, ..., Pid_n\rangle\rangle$
then
 let tid = spawning_task(Wid)
 extend TID **by** $tid_1, \ldots, tid_n$ **with**
  let $Tids = tid_1, \ldots, tid_n$
  extend $TASK$ **by** $t_1, \ldots, t_n$ **with**
   $\vdots$
   $pvmd(tid_j) := pvmd$
   $Enroll(t_j, tid_j, tid, Pid_j)$
   $\vdots$
  endextend

  if $pvmd(tid) = pvmd$
  then $SpawnRepInfo(Tids, Wid)$
  else *return* $spawn\_rep\_msg\langle Tids, Wid\rangle$ *to* $pvmd(tid)$
 endextend

 where $Enroll(Task, Tid, Parent, Pid)$
  $\equiv tid(Task) := Tid$
   $pid(Task) := Pid$
   $parent(Task) := Parent$

When receiving an *ext_spawn_report*, the pvmd enrolls the reported processes as tasks. Depending on whether the task that has initiated the spawn request is local or not, the resulting list of new tids either is appended to the local wait context (in *SpawnRepInfo*) or it is returned to the corresponding remote pvmd. We omit the straightforward formalization of *SpawnRepInfo*(*Wid* , *Tids*) : the returned wait identifier *Wid* provides access to the local wait context where the request counter *waitcount*(*Wid*) and the list *rep_info(Wid)* of tids of successfully spawned tasks are updated; upon final completion, the result of the distributed spawn operation is returned to the calling task *req_info*(*Wid*).

The rules for interaction between the spawning task pvmd and the pvmds on selected remote hosts are:

$\texttt{spawn\_req\_msg}\langle\rangle$

> **if** $event(pvmd) = spawn\_req\_msg\langle Params, Wid, Tid\rangle$ *from* $pvmd'$
> **then**
>   **extend** WID **by** wid **with**
> $req\_info(wid) := Tid$
> $rep\_info(wid) := Wid$
>   *trigger* $try\_to\_spawn\langle Params, wid\rangle$
>   **endextend**
>
> **where** $Params \equiv File, Args, Flag, Where, N$

$\texttt{spawn\_rep\_msg}\langle\rangle$

> **if** $event(pvmd) = spawn\_rep\_msg\langle Tids, Wid\rangle$ *from* $pvmd'$
> **then** *SpawnRepInfo*(*Tids* , *Wid*)

Remark Note that the function *hostselect* used above is a particular interesting example which shows the integration potential of the evolving algebra methodology. This function is an external dynamic function for the PVM model constituted by our rules. It contains however a static subfunction which is defined by conditions on host selection which are given in the PVM manual.

3.5 PVM Task Deletion

The routine *pvm_kill*(*Tid*) causes the pvmd to kill the task identified by *Tid*. In the formal description a symbolic system command, *kill_process*, is used to express the resulting required interaction between PVM and the operating system. If the task to be killed resides on a remote processor, the local pvmd forwards a kill message to the corresponding remote pvmd. Upon receiving a kill_msg() a pvmd acts as if it had received a *pvm_kill* by one of its local tasks.

pvm_kill()

if *event(pvmd) = kill(Tid) from Task*
thenif *pvmd(Tid) = pvmd*
then
  delete Tid from TID
  kill_process(pid(task(Tid)))
else
  forward kill_msg⟨Tid⟩ to pvmd(Tid)

kill_msg⟨⟩

if *event(pvmd) = kill_msg⟨Tid⟩ from pvmd′*
then
  delete Tid from TID
  kill_process(pid(task(Tid)))

4 Appropriate Ground Models

As has been observed in [3, 21] evolving algebras fit in a special way when it comes to link a non–formal system description S (requirements specification) to a formal model S_0 which has to be recognized as faithful formalization of S. This is because one can tailor the level of abstraction of the formal model S_0 to be built in such a way that its domains and functions directly correspond to the basic objects and operations of S; this correspondence must be understandable by inspection because by definition there is no possibility to prove any correctness statement about S. Further refinements of the model S_0, once this *ground model* has been recognized as "correct", are obtained through possibly formal transformations leading to an implementation; since at this level only formal models are involved, in the ideal case the correctness of the transformations can be proved (by mathematical or theorem proving methods).

For an interesting although small example of such a system see the evolving algebra specification of the steam–boiler control which has been refined leading from the ground model to a running C^{++} program (see [2]). A more involved example is constituted by the problem of faithfully defining the standard for a programming language, i.e. in an easy to understand but precise and complete way. For the ISO Prolog Standard published in 1995 and the IEEE VHDL'93 Standard evolving algebra ground models appear in [15, 11].

5 Distributed Evolving Algebras

In this section we resume the preceding discussion by the definition of distributed evolving algebras, taken from [23] to which we refer for details.

A *distributed evolving algebra* $\mathcal{A}$ comes as a finite set of sequential evolving algebra programs, called *modules*, together with a finite set of concurrently operating *agents* executing these modules, and a collection of *initial states* of $\mathcal{A}$. The vocabulary of $\mathcal{A}$ is obtained by combining the vocabularies of the sequential evolving algebra programs of all modules. A function *Mod* specifies the mapping from agents to modules where the latter are identified through *module names*. In contrast to the function *Mod* and the set of agents, which may both change dynamically, module names are static nullary functions. In addtion to *Mod* there is a special nullary function *Self* which each agent interprets in a different way: an agent a interprets *Self* as a.

A distributed computation of $\mathcal{A}$ starting in some initial state S_0 of $\mathcal{A}$ and running through a sequence of states S_1 S_2 ... is defined by the collective actions of the various agents. (Note that each agent a has its own partial view of a given state S_i as reflected by the vocabulary of $Mod(a)$.) In a single computation step leading from state S_i to state S_{i+1} one or more agents may participate such that each of these agents contributes to the local state changes by making his own *move*. Since the moves of the subcomputations of the individual agents are linearly ordered in time, a *run* of $\mathcal{A}$ in principle is characterized by a partially ordered set of moves together with the associated agents and the resulting states. A rigorous semantical definition of the notion of runs of distributed evolving algebras is given in [23].

The Distributed *PVM Algebra* In our PVM model the agents are represented by the concurrently operating PVM daemons as identified through the function *pvmd* (taking the role of *Self*). The set of pvmds increases or decreases depending on the dynamically changing configuration of the virtual machine. As there are only two different kinds of pvmds, the *demiurge* on the master host (see Section 1.1) and the ordinary pvmds on the other hosts, we have only two different types of modules. For the the PVM features that have been discussed here, the differences in the functional behaviour of these two types of pvmds are irrelevant.

6 Conclusions

Continuing the work started in [10] we have defined a formal model for PVM from scratch. The goal was to illustrate some outstanding merits of the evolving algebra methodology to design and analysis of complex computer systems. This text has been written for the practitioner; we advice the theoretically inclined to study [23] where Gurevich provides a rigorous mathematical foundation of the semantics of evolving algebras.

References

[1] Ch. Beierle and E. Börger. A WAM extension for type-constraint logic programming: Specification and correctness proof. Research report IWBS 200, IBM Germany Science Center, Heidelberg, December 1991.

[2] Ch. Beierle, E. Börger, I. Đurđanović U. Glässer, and E. Riccobene. An evolving algebra solution to the steam-boiler control specification problem. Seminar on *Methods for Specification and Semantics* (Dagstuhl, June 1995), Report, 1995.

[3] E. Börger. Logic Programming: The Evolving Algebra Approach. In B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam, 391–395.

[4] E. Börger. Annotated bibliography on evolving algebras. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[5] E. Börger and G. Del Castillo. A formal method for provably correct composition of a real–life processor out of basic components (The APE100 reverse engineering project). In *Proc. of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95).*

[6] Egon Börger, Giuseppe Del Castillo, Paola Glavan, and Dean Rosenzweig. Towards a mathematical specification of the APE100 architecture: The APESE model. In B. Pehrson and I. Simon, editors, *Proc. of the IFIP 13th World Computer Congress 1994, Volume I: Technology and Foundations*, pages 396–401. Elsevier Science Publishers B. V., 1994.

[7] E. Börger and B. Demoen. A framework to specify database update views for Prolog. In M. J. Maluszynski, editor, *PLILP'91. Third International Symposium on Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1991.

[8] E. Börger and I. Đurđanović. Correctness of compiling Occam to Transputer code. 1995 (submitted).

[9] E. Börger, I. Đurđanović, and D. Rosenzweig. Occam: Specification and compiler correctness. part i: The primary model. In E.-R. Olderog, editor, *Proc. of PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489–508. North-Holland, 1994.

[10] E. Börger and U.Glässer. A formal specification of the PVM architecture. In B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam, 402–409.

[11] E. Börger, U. Glässer, and W. Mueller. Formal definition of an abstract VHDL'93 simulator by EA–machines. In C. Delgado Kloos and Peter T.

Breuer, editors, *Semantics of VHDL*, volume 307 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1995.

[12] E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[13] E. Börger and S. Mazzanti. A correctness proof for pipelining in RISC architectures. Manuscript, 1995.

[14] E. Börger and D. Rosenzweig. An analysis of Prolog database views and their uniform implementation. Research report CSE-TR-89-91, EECS, University of Michigan, Ann Arbor MI, 1991.

[15] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 1995.

[16] E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In L. C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Series in Computer Science and Artificial Intelligence. Elsevier Science B.V./North–Holland, 1995.

[17] E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP($\mathcal{R}$) programs. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[18] E. Börger and P. Schmitt. A formal operational semantics for languages of type Prolog III. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'90, 4th Workshop on Computer Science Logic*, volume 533 of *Lecture Notes in Computer Science*, pages 67–79. Springer, 1991.

[19] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, September 1994.

[20] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994. to appear.

[21] U. Glässer. Systems Level Specification and Modelling of Reactive Systems: Concepts, Methods, and Tools. In Proc. of the *Fifth International Conference on Computer Aided Systems Technology (EUROCAST'95)*, Innsbruck, Austria, May, 1995.

[22] Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[23] Yuri Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[24] Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.

[25] Y. Gurevich, J. Huggins, and R. Mani. The Generalized Railroad Crossing Problem: An Evolving Algebra Based Solution. EECS Department, University of Michigan–Ann Arbor, CSE Technical Report, CSE-TR-230-95, 1995.

[26] Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[27] Jim Huggins. Kermit: Specification and verification. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[28] The Institute of Electrical and Electronics Engineering. *IEEE Standard VHDL Language Reference Manual—IEEE Std 1076–1993*, New York, NY, USA, 1994.

[29] M. Marcotty and H.F. Ledgard. The World of Programming Languages. Springer-Verlag, 1986.

[30] Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992.

[31] David L. Parnas. Information distribution aspects of design methodology. In C. V. Freiman, editor, *Proc. of IFIP Congress 1971, Volume 1: Foundations and Systems*, pages 339–344. North-Holland, 1972.

[32] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20(4):531–545, 1994.

[33] Martin Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B*, pages 675–788, Elsevier, 1990.

[34] Niklaus Wirth. Algorithms & Data Structures. Prentice–Hall, 1975.

Correctness of compiling Occam to Transputer code

EGON BÖRGER & IGOR ĐURĐANOVIĆ

Dipartimento di Informatica
Università di Pisa
Cso Italia 40
I-56125 Pisa
boerger@di.unipi.it
fax: xx39–50–887226

University Paderborn
Fachbereich 17 – Informatik
Warburgerstr. 100
33098 Paderborn, Germany
igor@uni-paderborn.de
fax: xx49–5251–603338

This paper contributes to the development of a rigorous mathematical framework for the study of provably correct compilation techniques. The proposed method is developed through an implementation of a real–life non–toy imperative programming language with nondeterminism and parallelism – namely Occam – to a commercial machine, namely the Transputer. We provide a mathematical definition of the Transputer Instruction Set architecture for executing Occam together with a correctness proof for a general compilation schema of Occam programs into Transputer code.
We start from the ground model, an abstract processor, running a high and a low priority queue of Occam processes, which formalizes the semantics of Occam at the abstraction level of atomic Occam instructions. We develop here increasingly more refined levels of Transputer semantics, proving correctness (and when possible also completeness) for each refinement step. Along the way we collect our proof assumptions, a set of natural conditions for a compiler to be correct, thus making our proof applicable to a large class of compilers. As a by–product our construction provides a challenging realistic case study for proof verification by theorem provers.

1. INTRODUCTION

It is well known that a reliable compilation methodology which includes compiler verification requires that the semantics of both source and target language have been rigorously defined. In [BĐR:94] a high–level mathematical model for the truly concurrent semantics of Occam has been developed which captures the intuitive programmer's view of the dynamics of Occam in terms of atomic Occam instructions. In this paper we provide a mathematical model for the Transputer Instruction Set architecture. We use these two models to prove the correctness of the compilation scheme proposed in [Inmos:88] for the compilation of Occam programs into the Transputer instruction set.

<u>**Main Theorem**</u> Every compiler which satisfies the conditions listed in this paper compiles arbitrary Occam programs correctly into Transputer instructions.

The main problem in proving the theorem consists in bridging the gap between the abstraction levels of Occam and the Transputer. We relate the Occam ground model to the Transputer model by a series of stepwise refined intermediate models. At each refinement step we show the correctness and when possible also the completeness of the implementation. As a side product of our work for the correctness proof we obtain a detailed explanation of the rationale of the compilation scheme in [Inmos:88].

Several remarks have to be made to avoid a possible misunderstanding of the theorem.

- Correctness is to be understood as relative to the formal Occam and Transputer ground models; along the specification of these models it is made explicit and precise which parameters of the high–level model are correctly preserved through the refinement steps. Therefore it is crucial that these models are simple and transparent and can independently be justified, on pragmatic grounds, as adequate formalization of the programming language Occam and of the Transputer processor respectively.
- The correctness claim and its proof are not absolute but relative; indeed the mathematical models contain a certain number of interfaces to the environment which are supposed to work in accordance with those properties which are used as assumptions in the proof. Such relative correctness proofs are the best one can reach by rigorous methods, given the

huge complexity of the problems under study.

- The proof is a mathematical proof in the classical sense of the term, based upon human reasoning and insight. It is split into numerous steps which have been introduced in order to break down the complexity of the whole construction into manageable and well understood small pieces. In order to be faithful to the Transputer implementation of Occam (see [Inmos]) we have to meet the Compiler Writer's Guide [Inmos:88]; in order to achieve the desired correctness proof we have to structure the Compiler Writer's Guide by decomposing it through many refinement steps whose correctness is amenable to the precise *formulation* and to the *proof* of the relevant properties.
  We have also made a particular effort in order to achieve that most of our proofs become "local", i.e. have to do with well defined specific features and leave the whole rest abstract or unchanged; this locality makes the approach modular in a strong (not only syntactical) sense: we isolate orthogonal Transputer instruction set components and often can proceed with our proofs even instructionwise. The guiding principle for breaking complex statements into simpler ones has been to stop only where the proofs become routine exercises which can be carried out by an automatic theorem proving system. Clearly we concentrate our attention on a precise outline and full proof of the global proof strategy. The routine exercises which consist in carrying out simple inductions, case distinctions, etc. are left to the reader.
- We prove the correctness of a compiling specification. This means that we do not describe any specific compiler but formulate explicit conditions on the compilation function, namely those properties of the compilation process which we use for the correctness proof. These assumptions represent useful directives for correct compiler design and can also be used for variations of the design scheme for the underlying architecture, preserving correctness without need for testing of the unchanged components.

The overall structure of our refinement hierarchy is divided into two parts (sections 3. – 4. and 5. – 6.). It highlights essential points of provably correct compiler development, namely the implementation of the control structure and of environments (including auxiliary functions and dealing with relative addressing). The Occam ground model comprises a refinement of the concurrent Occam semantics to an abstract sequential processor which runs two queues of processes (one of low and one of high priority); in order to concentrate on Occam's distributed features — i.e. communication, parallelism and alternation — the Occam ground model has been based upon the usual layout of (imperative) programs as flowchart along which the process(es) (daemon(s)) are supposed to walk, each carrying along his own environment in which he executes at each node an atomic Occam instruction. In section 4. the generation of this flowchart is replaced by compilation into still abstract code. In section 6. the abstract code will be refined to Transputer code whose execution is based upon the Transputer model developed in section 5..

The refinement of the flowchart generation to code compilation consists in replacing the walking of daemons through the flowchart by moving instruction pointers through abstract code produced (together with the environment) by compilation. Thus it was natural to split this compilation again into two steps: compilation of the control structure and compilation of the environment. For the control structure compilation we first linearize the flowchart (by introducing goto–instructions) and then describe its machine internal representation by loading the result of a compilation function. For the compilation of the environment we first refine environments to be determined by the program structure and not any more by the daemons; we then implement environments by blocks of memory (obtaining eventually relocatable environment access by relative addresses of identifiers with respect to daemons as base addresses).

Section 6. refines the compilation from abstract to Transputer instructions. Here again we have four refinement steps: first we introduce the Transputer ground model consisting of various registers used for the execution of Transputer instructions.

Then we eliminate the environment from the run–time by building the relative addresses for identifiers into the compilation (with run–time calculation of the absolute address). As third step we implement abstract functions refining daemons to workspace addresses. At the end we make the Transputer code relocatable by introducing the technique of relative branching. At compile–time the instruction address offset (or distance) is calculated whereas at run–time this offset is used to obtain the absolute addresses.

Diagram 1 summarizes the overall structure of our refinement hierarchy. The Appendixes 7.2. and 7.3. summarize the complete rule system for the result of sections 4. and 6.

In contrast to other work on compiler correctness in the literature, we do not verify a series of compilers which compile between various intermediate languages to give a multi–stage compiler for the whole Occam language. Rather each refinement of our function *compile* compiles directly from the Occam source language and thus constitutes in itself a new compiler.

The refinement hierarchy and the correctness proofs work for the full Occam language. Since our main goal is to let the characteristic Occam features stand out in a transparent way along the whole implementation process we have decided to leave the evaluation and compilation of expressions and the implementation of values still abstract; one can add to our refinement chain fur-

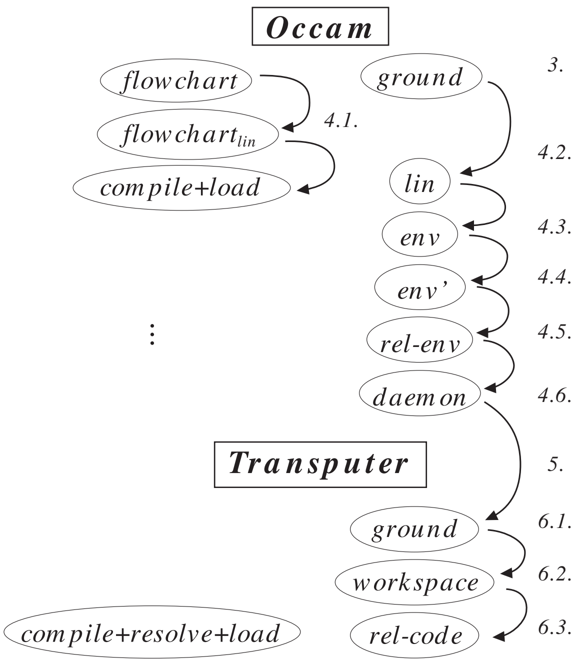


FIGURE 1. Refinement structure

ther levels which deal with this, including the particular Transputer scheme for encoding instructions and providing large operands (i.e. postfixing and prefixing). (See the careful treatment of expression compilation in [MMO:95], pages 38–49.) For notational convenience and without loss of generality we also work with only one processor (Transputer) but without loosing its multi–task capabilities (queue).

The methodology which allows us the appropriate refinements of the abstraction level, providing full mathematical rigor but avoiding heavy formal overhead, makes use of Gurevich's notion of evolving algebras [Gur:95]. One essential feature of the potential of the evolving algebra approach to specification and verification of large systems is the fact that evolving algebra models can be read and understood without any specific previous formal training. We invite the reader who does not know the notion of evolving algebra to read our models as "pseudo–code over abstract data"; that suffices for an understanding of the specification. To carry out the proofs, some more technical understanding of what constitutes computations by evolving algebras is needed however.

Section 2. summarizes the basic definitions and notation. Section 3. recalls the basic constituents of the Occam ground model which are the starting point of the refinements in section 4.. Along the way we use the chance to adapt the Occam oriented ground model to the needs of the Transputer, the target model of this paper. We pay attention that these modifications preserve the correctness theorems of [BĐR:94]. As a by–product this permits the reader to follow the present paper without knowing [BĐR:94].

We suppose the reader to be familiar with (the problems of) provably correct compiler development and to have some idea about Occam or at least about the notion of parallel computation.

2. EVOLVING ALGEBRA: PREREQUISITES AND NOTATION

In our specifications we use the notion of *evolving algebras*, see [Gur:95]. Evolving algebras represent a mathematically rigorous form of fundamental operational intuitions of computing. This permits to read and understand our description as 'pseudocode over abstract data', without any particular theoretical prerequisites. For the sake of completeness we review here our notation and refer the interested reader for the foundational justification to [Gur:95].

We treat abstract data as elements of (possibly not furthermore specified) sets (domains, *universes*). The operations allowed on universes will be represented by partial *functions*. Dynamic changes are obtained by executing *function updates* of form

$$f(t_1, \ldots, t_r) := t$$

whose execution is to be understood as *setting* (modifying) the value of the function f at given arguments. Note that the 0-ary functions play the role of *variables* in programming.

An *evolving algebra* is defined by a finite set of *transition rules* of form "**if** *Cond* **then** *Updates*" where *Cond* (condition or guard) is an expression, the truth of which triggers *simultaneous* execution of all updates in the finite set of *Updates*. Simultaneous execution helps us avoid fussing and coding to, say, interchange two values. Since functions may be partial, equality in the guards is to be understood as implying that both arguments are defined.

Unless explicitly declared to be static, functions are dynamic. For a given evolving algebra $\mathcal{A}$ a function must be dynamic if it can be updated in rules of $\mathcal{A}$, i.e. if it is an f of a functional update; otherwise this update would be syntactically incorrect. If we want to stress that a dynamic function f may change its value without being updated by a rule of $\mathcal{A}$ we declare this function as external.

In applications an evolving algebra usually comes together with a set of *integrity constraints*, i.e. extralogical axioms and/or rules of inference which specify the intended domains.

Our rules will always be constructed so that the guards imply *consistency* of updates.

Evolving algebras transform first order structures (abstract machine states). Thus they can be understood as *transition systems* whose states are first order structures. This intuitively clear semantics of evolving algebras has a rigorous definition given in [Gur:95].

In applications of evolving algebras one usually encounters *heterogenous* signatures with several universes, which may in general grow and shrink in time. Therefore we use the following update form to extend a universe:

extend A **by** $t_1, \ldots, t_r$ **with** *Updates* **endextend**

where *Updates* may (and should) depend on t_i's, setting the values of some functions on *newly created* elements t_i of A. [Gur:95] has shown how to reduce these domain extensions and heterogeneous structures to the basic model of a homogenous signature (with one universe).

The forms obviously reducible to the above basic syntax, which we shall freely use as abbreviations, are **where**, **let** and **if_then_else**.

As stated above all the updates appearing in a rule are executed simultaneously. In the rare cases where we need sequentiality we will use the update form:

seq *Updates* **endseq**

where the *Updates* are executed sequentially.

Instead of writing three rules which differ only in parts p_i of their guards and in updates u_i, we write one rule of form:

$$\begin{array}{l} \textbf{if } p_1 \mid p_2 \mid p_3 \\ \quad \wedge \ \ldots \\ \textbf{then } u_1 \mid u_2 \mid u_3 \\ \quad\quad \ldots \end{array}$$

We shall assume that we have the standard mathematical universes of booleans, integers, lists of whatever etc (as well as the standard operations on them) at our disposal without further mention. In general we will write *NAME* for the universe of *objects* of type *name*. We use notations $x_1.x_2.\ldots.x_r$, $x_1, x_2, \ldots, x_r$, $(x_1, x_2, \ldots, x_r)$ etc. for lists.

An evolving algebra, as given above, determines the dynamics of a transition system. We are usually only interested in states reachable from some designated *initial states*, which may be specified in various ways.

3. REVIEW OF THE $OCCAM_{ground}$ MODEL

We summarize here the final version of the Occam ground model developed in [BĐR:94] which is the starting point of the refinements in this paper. We make an effort to explain that ground model from scratch in order to enable the reader to understand this paper without knowing the details of [BĐR:94]. The reader who is familiar with [BĐR:94] can skip this section and come back to it should he want to check the justification for some slight technical (mostly notational) changes which we incorporate here in order to smoothen the transition from flowchart generation to compilation.

The starting point $OCCAM_{ground}$ of this paper is itself the result of various refinement steps defined in [BĐR:94]. They lead from a high–level truly concurrent model of Occam to abstract sequential processors running two queues of processes (for low and for high priority) with time–slicing and interrupt mechanism; these sequential processors (Transputers) run concurrently to other processors and external channels.

For notational convenience we restrict our attention here to only one processor and skip external channels; technically speaking this means only to suppress in certain functions and rules of our model the parameter ranging over the universe of *PROCESSOR*s. In order to avoid repetitions we will also deal with just one queue, the low priority queue which is subject to time–slicing in order to preserve completeness of the sequential implementation of concurrent runs; it would be easy to incorporate the interrupt handling by adding a new rule within the Fetch-Execute mechanism.

The flowchart

The usual layout of (imperative) programs as flowchart with nodes marked by atomic instructions is formalized by a set of *NODE*s with functions:

$$\begin{array}{rcl} next & : & NODE \times \mathbf{N} \rightarrow NODE \\ cmd & : & NODE \rightarrow CODE \end{array}$$

representing the edges and marks respectively. For notational convenience we use the following macros:

$$\begin{array}{rcl} next(n) & \stackrel{\text{def}}{=} & next(n, 0) \\ yes(n) & \stackrel{\text{def}}{=} & next(n, 0) \\ no(n) & \stackrel{\text{def}}{=} & next(n, 1) \end{array}$$

Processes are represented as daemons (elements of an abstract set *DAEMON*) which are placed into the flowchart by a dynamic function a (for "agent"):

$$loc \ : \ DAEMON \rightarrow NODE.$$

The processor

The processor is formalized by a queue of running (i.e. non sleeping) daemons, a currently active daemon and clock functions (for time–slicing). Formally — the set of of non sleeping processes, waiting to become active — $QUEUE \subseteq DAEMON^*$ comes with external functions:

$$\begin{array}{rcl} first, last & : & QUEUE \rightarrow DAEMON \\ rest & : & QUEUE \rightarrow QUEUE \end{array}$$

and a 0-ary dynamic function:

$$q \ : \ QUEUE$$

which represents the current incarnation of the queue. The unique currently active process (the "agent") is formalized by a 0-ary dynamic function:

$$a \ : \ DAEMON \cup \{nil\}.$$

The processor clock is formalized by an external function:

$$timer \ : \ \mathbf{N}$$

which is used to set the dynamic 0-ary function:

$$start \quad : \quad \mathbf{N}$$

when a process is taken out of the queue to become active. The dequeuing (activating the next process in the queue) is done by the macro[1]:

$$\begin{array}{lll} dequeue & \stackrel{\text{def}}{=} & \textbf{if} \text{ not } empty\ q \\ & & \textbf{then}\ a := first(q) \\ & & \qquad q := rest(q) \\ & & \qquad start := timer \\ & & \textbf{else}\ a := nil \end{array}$$

At places which are safe for time–slicing the currently active process $a \neq nil$ will be put back into the queue when its time is elapsed, i.e. exceeds the time–slicing *period*, a 0-ary external (implementation defined) function[2]:

$$period \quad : \quad \mathbf{N}$$

This is done by the rule[3]:

$$\boxed{\textsf{time–slice}}\ \begin{array}{l} \textbf{if}\ a \neq nil \\ \quad \wedge\ elapsed \\ \textbf{then seq}\ q := q.a \\ \qquad\qquad dequeue \\ \qquad \textbf{endseq} \end{array}$$

where

$$elapsed \stackrel{\text{def}}{=} timer - start > period$$

Beside of the queue q of running processes, there is another queue for administrating time–waiting processes[4]:

$$time\_q \quad : \quad QUEUE$$

If a daemon is in the time queue, then the dynamic function:

$$t_{min} \quad : \quad DAEMON \rightarrow \mathbf{N}$$

holds the minimal time he is waiting for. We will use the following macros[5] for (non–deterministic) insertion/deletion of daemons into/from the time-queue *time_q*:

$$\begin{array}{ll} time\text{-}insert & \stackrel{\text{def}}{=} \\ & \textbf{let}\ time\_q = x_1.\ldots.x_i.x_{i+1}.\ldots.x_r \\ & \textbf{let}\ t_{min}(x_i) \leq t_{min}(a) \leq t_{min}(x_{i+1}) \\ & time\_q := x_1.\ldots.x_i.a.x_{i+1}.\ldots.x_r \\ time\text{-}delete\ x & \stackrel{\text{def}}{=} \\ & \textbf{let}\ time\_q = x_1.\ldots.x_i.x.x_{i+1}.\ldots.x_r \\ & time\_q := x_1.\ldots.x_i.x_{i+1}.\ldots.x_r \end{array}$$

Dynamics of Processes

Daemons x are sitting at their current flowchart node (see 7.1.) $loc(x)$ waiting to be activated by the processor for execution of the instruction $cmd(loc(x))$ in this node. We abbreviate this condition for a daemon to get his instruction executed by the following:

$$cmd\ is\ c \stackrel{\text{def}}{=} cmd(loc(a)) = c \wedge \text{not } elapsed$$

Note that this condition is false if the **time–slice** rule can fire[6]. After having executed the instruction of a daemon the processor will proceed to the next instruction; we abbreviate this action by:

$$proceed \stackrel{\text{def}}{=} loc(a) := next(loc(a))$$

A daemon can become inactive either by time–slicing or because for some reason he has to wait (for a communication partner to be ready or for a time condition to become satisfied) and therefore "goes to sleep". Going to sleep means to become inactive without going back into the process queue:

$$sleep\ at\ n \stackrel{\text{def}}{=} loc(a) := n, dequeue$$

Being woken up then corresponds to return into the queue:

$$\begin{array}{lll} wakeup\ x & \stackrel{\text{def}}{=} & enqueue\ x \\ enqueue\ x & \stackrel{\text{def}}{=} & q := q.x \end{array}$$

The behavior of Occam programs can now be rigorously defined by giving for each atomic instruction rules which define the semantics of that instruction.

Declarations

In the flowchart model of Occam daemons carry their

[1] In [BĐR:94] we had introduced the dequeue update as independent rule "**if** $a = nil \wedge$ not *empty q* **then** *dequeue*" which was the only executable rule each time a was set to nil (by **time–slice**, **stop** or **end** rule or by sending a process to sleep). To smoothen the transition to the **Transputer** we incorporate here the effect of dequeuing directly into the relevant rules; it is obvious that this modification does not change the semantics. The present definition for dequeuing is in accordance with the Inmos manual [Inmos:88] which says that the special daemon nil indicates that no valid process is present. Our present definition reflects that when the **Transputer** runs out of processes to be executed nothing happens: the processor runs in *fetch* mode (see below) until some process shows up.

[2] The restriction to one processor comes up to have suppressed the *PROCESSOR* parameter for the functions q, a, *timer*, *period*, *start*. The restriction to one priority queue comes up to suppress the parameter q of *dequeue*.

[3] Note the sequential execution of enqueueing and dequeuing of a daemon. This is done to avoid problems with simultaneous execution when only one daemon exists.

[4] Note that in the **Transputer** there are two timer-queues, one for each priority. In [BĐR:94] we could formalize time queues simply as sequences *min_time* of pairs of daemons and their waiting time. For a smooth transition to the **Transputer** model it turns out to be advantageous to split *min_time* into a list *time-q* of daemons and a function t_{min} which records the waiting times. We leave it as an exercise to show that this implementation is correct.

[5] At the level of abstraction of the **Occam** interpreter in [BĐR:94] we could afford to abstract from the order of the processes in the timer queue. Since in reality the hardware is responsible for taking waiting processes out of the time queue, we have to make this sequentiality in the **Transputer** model explicit; see the **time_wakeup** rule below.

[6] The condition $a \neq nil$ in *cmd is* from [BĐR:94] could be skipped with the understanding that $cmd(undef) = undef$, $loc(nil) = undef$.

*ENV*ironment, associated to them by a dynamic function:

$$env \quad : \quad DAEMON \rightarrow ENV$$

This function is updated through execution of declaration instructions[7]. Without loss of generality we can replace the abstract notion of *ENV*ironment used in [BĐR:94] by defining *ENV* as $(ID \times (VAR \cup CHANNEL))^*$; clearly *VAR* and *CHANNEL* are supposed to be disjoint.

The semantics of declaration instructions is defined by the following simple rules:

$\boxed{\mathsf{decl\_var}(id_1, \ldots, id_r)}$
if *cmd is* $\mathsf{decl\_var}(id_1, \ldots, id_r)$
then extend *VAR* **by** $v_1, \ldots, v_r$ **with**
$env(a)$ **:=** $append(e, env(a))$
endextend
proceed
where $e = (id_r, v_r), \ldots, (id_1, v_1)$

For channel declarations one has to add the initialization of the channel agent (reader or writer) to *nil*; the agents needed for the asynchronous arrival of communications partners (see below) are formalized by a dynamic function:

$$agent \quad : \quad CHANNEL \rightarrow DAEMON \cup \{nil\}.$$

$\boxed{\mathsf{decl\_chan}(id_1, \ldots, id_r)}$
if *cmd is* $\mathsf{decl\_chan}(id_1, \ldots, id_r)$
then extend *CHANNEL* **by** $c_1, \ldots, c_r$ **with**
$env(a)$ **:=** $append(e, env(a))$
$agent(c_1)$ **:=** *nil*
...
$agent(c_r)$ **:=** *nil*
endextend
proceed
where $e = (id_r, c_r), \ldots, (id_1, c_1)$

Note that the Occam scoping scheme is respected: the later declaration overrides the previous one (see the definition of *bind* function below). The environment is restored once the daemon has left the scope of declaration:

$\boxed{\mathsf{decl\_end}(r)}$
if *cmd is* $\mathsf{decl\_end}(r)$
then let $env(a) = (id_r, v_r), \ldots, (id_1, v_1), e'$
$env(a)$ **:=** e'
proceed

Expressions

Expression evaluation is kept abstract using a function:

$$bind \quad : \quad ID \times ENV \rightarrow VAR \cup CHANNEL$$

for binding identifiers in a given environment and two functions for evaluating variables and expressions (in a given environment) to *VAL*ues which we keep abstract[8]:

$$eval \quad : \quad VAR \rightarrow VAL$$
$$eval \quad : \quad EXP \times ENV \rightarrow VAL.$$

As expected *bind* is recursively defined by:

$$bind(id, [\,H \,|\, T\,]) \quad = \quad \begin{cases} x & \text{if H = (id,x)} \\ bind(id, T) & \text{otherwise} \end{cases}$$

Notationally we use $\bar{v}, \bar{c}$ instead of $bind(v, env(a))$, $bind(c, env(a))$ respectively. Thus we have the following rules for assignment, timing, test of boolean condition:

$\boxed{\mathsf{ass}(v,t)}$ **if** *cmd is* $v := t$
then $eval(\bar{v})$ **:=** $eval(t, env(a))$
proceed

$\boxed{\mathsf{time}(v)}$ **if** *cmd is* TIME ? v
then $eval(\bar{v})$ **:=** *timer*
proceed

$\boxed{\mathsf{if}(b)}$ **if** *cmd is* $\mathsf{if}(b)$
then if $eval(b, env(a))$
then $loc(a)$ **:=** $yes(loc(a))$
else $loc(a)$ **:=** $no(loc(a))$

The rules for skip and stop are as follows:

$\boxed{\mathsf{skip}}$ **if** *cmd is* SKIP **then** *proceed*

$\boxed{\mathsf{stop}}$ **if** *cmd is* STOP **then** *dequeue*

Communication

The communicating daemons might arrive independently to the synchronizing channel. If a reader or writer arrives at a channel c before his communication partner, he will go to sleep after having notified to the channel his identity (updating the function $agent(c)$) and the place where he wants the input or the message he wants to output respectively. For this purpose in addition to the function *agent* two dynamic functions:

$$mssg \quad : \quad DAEMON \rightarrow VAL$$
$$place \quad : \quad DAEMON \rightarrow VAR$$

are introduced which are updated in the following two rules, where $idle\ c \stackrel{\text{def}}{=} agent(c) = nil$:

$\boxed{\mathsf{inp\_idle}(c,v)}$
if *cmd is* $c\,?\,v$
$\wedge$ *idle* $\bar{c}$
then *sleep at* $next(loc(a))$
$agent(\bar{c})$ **:=** a
$place(a)$ **:=** $\bar{v}$
$com\_mode(a)$ **:=** *input*

[7] As already in [BĐR:94] we skip the data-types of Occam and procedures as they are not in any way characteristic for the language and can be incorporated into our models in a standard way.

[8] We clearly assume these two evaluation functions to be related in the standard way.

$\boxed{\mathsf{out\_idle}(c,t)}$
if *cmd is* $c\,!\,t$
  $\wedge$ *idle* $\bar{c}$
then *sleep at next*$(loc(a))$
  $agent(\bar{c})$ **:=** a
  $mssg(a)$ **:=** $eval(t, env(a))$

Note that the auxiliary function:

$$com\_mode \quad : \quad DAEMON \rightarrow \{input, alt\_sleep, alt\_run\}$$

is set here to the value *input* in order to distinguish *ordinary* input requests from input requests which appear in the guard of an alternative (see the ALT-statement below). Since we consider here only internal channels, a daemon arriving as second communication partner to a non idle channel – formalized by $ready\ c \stackrel{\text{def}}{=} agent(c) \neq nil$ – completes the communication by placing the message to the place in question, by waking up the communication partner and by cleaning the channel[9]. This is formalized by the following two rules where *clear* c denotes the update $agent(c)$ **:=** *nil* which clears the channel once the communication is done:

$\boxed{\mathsf{inp\_ready}(c,v)}$
if *cmd is* $c\,?\,v$
  $\wedge$ *ready* $\bar{c}$
then $eval(\bar{v})$ **:=** $mssg(agent(\bar{c}))$
  wakeup $agent(\bar{c})$
  clear $\bar{c}$
  proceed

$\boxed{\mathsf{out\_ready}(c,t)}$
if *cmd is* $c\,!\,t$
  $\wedge$ *ready* $\bar{c}$
  $\wedge$ $com\_mode(agent(\bar{c})) = input$
then $eval(place(agent(\bar{c})))$ **:=** $eval(t, env(a))$
  wakeup $agent(\bar{c})$
  clear $\bar{c}$
  proceed

There is the special case of a daemon trying to output to a channel which received an input request through the guard of an alternative, formally which has been enabled by the first rule for the ALT-statement (alt_a, see below). In this case the output is allowed to be done only after the alternative in question has been really selected, but meantime, the outputting daemon has to announce his readiness (doing the updates of the out_idle rule) and to wake up the potentially input expecting communication partner; this includes to put him from the *alt_sleep* mode into the *alt_run* mode. This is formalized by the rule:

$\boxed{\mathsf{out\_alt}(c,t)}$
if *cmd is* $c\,!\,t$
  $\wedge$ *ready* $\bar{c}$
  $\wedge$ $com\_mode(agent(\bar{c})) \neq input$
then *sleep at next*$(loc(a))$
  $mssg(a)$ **:=** $eval(t, env(a))$
  $agent(\bar{c})$ **:=** a
  if $com\_mode(agent(\bar{c})) = alt\_sleep$
  then *time-delete* $agent(\bar{c})$
    wakeup $agent(\bar{c})$
    $com\_mode(agent(\bar{c}))$ **:=** alt_run

Alternation

In the sequential implementation of Occam the execution of ALT-statements is split into two phases:

- announcing — for each of the input requests $c_i\ ?\ v_i$ or time requirements TIME ? AFTER t_j where the corresponding boolean condition in the guard is *true* — the readiness to select the corresponding alternative once the corresponding input is ready or the time requirement is satisfied,
- selecting among the alternatives whose guards are satisfied.

The first phase is defined by the alt_a rule: the channels c_i appearing in guards $G_i = b_i : c_i?v_i$ with true boolean condition b_i are "enabled" by setting their agents to the executing daemon a:

$$\begin{array}{lcl} enable(b,c) & \stackrel{\text{def}}{=} & \textbf{if } eval(b, env(a)) \\ & & \quad \wedge\ agent(c) = nil \\ & & \textbf{then } agent(c) \textbf{ := } a \end{array}$$

The smallest among the enabled time requirements to be checked against *timer* is recorded into the function t_{min}. If no input is ready and none of the time requirements is satisfied yet, the daemon goes to sleep (setting his communication mode to *alt_sleep*) and inserts himself into the time-queue *time_q* (if there was at least one enabled time guard); it can be woken up by an outputting communication partner (see above the out_alt rule) or because the minimal waiting time t_{min} has been reached by *timer* (see time_wakeup rule below).

If, when enabling, a daemon finds at least one input to be ready or one time requirement to be satisfied – the latter is immediately true in case of the empty requirement, denoted by SKIP – he goes into communication mode *alt_run* and proceeds to the execution of the alt_s instruction after having "disabled" the channels which had been "enabled":

$$\begin{array}{lcl} disable(b,c) & \stackrel{\text{def}}{=} & \textbf{if } eval(b, env(a)) \\ & & \quad \wedge\ agent(c) = a \\ & & \textbf{then } agent(c) \textbf{ := } nil \end{array}$$

Note that the communication takes place only after the

[9] The restriction to internal channels means that we do not have to consider the **chan** rule of [BĐR:94], page 496.

corresponding alternative has been chosen.

$\boxed{\mathsf{alt\_a}(\vec{G})}$

if *cmd is* $\mathsf{alt\_a}(\vec{G})$
then
  $enable(b_1, \bar{c}_1)$
  ...
  $enable(b_p, \bar{c}_p)$
  if $\exists\, i\ (eval(b_i, env(a)) \wedge ready\ \bar{c}_i)$
    $\vee\ t\_wait < timer$
    $\vee\ \exists\, k\ (eval(b_k, env(a)))$
  then $com\_mode(a)$ **:=** alt_run
    proceed
  else $com\_mode(a)$ **:=** alt_sleep
    sleep at $next(loc(a))$
    if there is an enabled time guard
    then seq $t_{min}(a)$ **:=** t_wait
      time-insert
    endseq
where
  $G_i = b_i : c_i ?\, v_i$
  $G_j = b_j : \mathsf{TIME} ?\, \mathsf{AFTER}\, t_j$
  $G_k = b_k : \mathsf{SKIP}$
  $t\_wait = \begin{cases} \infty \text{ if } \forall\, j\ (\neg eval(b_j, env(a))), \text{ otherwise} \\ \min_j \{eval(t_j, env(a)) \mid eval(b_j, env(a))\} \end{cases}$
  $1 \le i \le p < j \le q < k \le r$

Obviously, the sentence "there is an enabled time guard" stands for the expression $\exists\, j\ (eval(b_j, env(a)))$, where $p < j \le q$. Note that here and in the sequel we use indices and "..." only to have a concise notation without contradicting the formal character of our model.

$\boxed{\mathsf{alt\_s\_com}} \mid \boxed{\mathsf{alt\_s\_time}} \mid \boxed{\mathsf{alt\_s\_skip}}\ (i, \vec{G})$

if *cmd is* $\mathsf{alt\_s}(\vec{G})$
  $\wedge\ eval(b_i, env(a))$
  $\wedge\ agent(\bar{c}_i) \neq a \mid timer > eval(t_i, env(a)) \mid$
then
  $disable(b_1, \bar{c}_1)$
  ...
  $disable(b_p, \bar{c}_p)$
  $loc(a)$ **:=** $next(loc(a), i)$
where
  $G_i = b_i : (\, c_i ?\, v_i \mid \mathsf{TIME} ?\, \mathsf{AFTER}\, t_i \mid \mathsf{SKIP}\,)$

The value p (as in previous rule) is the count of input guards appearing in ALT construct. (Note our "|" notation explained in the prerequisites section 2..)

$\boxed{\mathsf{time\_wakeup}}$

if $time\_q$ not *empty*
  $\wedge\ timer > t_{min}(first(time\_q))$
then $com\_mode(first(time\_q))$ **:=** alt_run
  wakeup $first(time\_q)$
  time-delete $first(time\_q)$

Note that by our definition of insertion into the time queue processes in that queue are ordered by the time they are waiting for.

$\boxed{\textbf{Parallelism}}$

In executing a PAR-statement the newly created daemons have to be linked to the currently executing daemon using a function:

$$father\ :\ DAEMON \rightarrow DAEMON$$

and the currently executing daemon – who will go to sleep – has to record how many processes he has created, updating a function:

$$count\ :\ DAEMON \rightarrow \mathbf{N}$$

The two rules to execute (begin and end of) PAR-statements are as follows[10]:

$\boxed{\mathsf{par(r)}}$ **if** *cmd is* $\mathsf{par}(r)$
then extend *DAEMON* **by** $x_1 \ldots x_r$ **with**
  q **:=** $q.x_1. \ldots .\, x_r$
  ...
  $father(x_i)$ **:=** a
  $loc(x_i)$ **:=** $next(loc(a), i)$
  $env(x_i)$ **:=** $env(a)$
  ...
  $count(a)$ **:=** r
  sleep at $next(loc(a))$
endextend
where $1 \le i \le r$

$\boxed{\mathsf{end}}$ **if** *cmd is* end
then *dequeue*
  r **:=** $r - 1$
  if $r = 1$
  then *wakeup* $father(a)$
where $r = count(father(a))$

4. FLOWCHART AND ENVIRONMENT COMPILATION

In this section we compile into still abstract code the Occam control structure (which is embodied in the flowchart of the ground model) and the environment (which in the ground model is associated to the daemons). In 4.1. the flowchart is linearized (by introducing goto-instructions instead of multiple in/outgoing edges). In 4.2. the resulting sequence of nodes, marked by atomic Occam instructions, is described by a compilation function whose output is loaded into memory. In 4.3. environments are refined as determined recursively by the program structure, resulting in their still abstract definition by the compilation function. In 4.4. this definition is refined by placing variables and channels into memory using an auxiliary function which computes the needed environment size recursively along the program structure. This allows us in 4.5. to make

[10] Since we concentrate the attention to one processor we do not consider the PLACED PAR statement which is semantically different from the PAR-statement, it is allowed to be used only once – at the beginning of the Occam program – to "place" daemons on different processors; the daemons are not supposed to terminate and are not linked to the father.

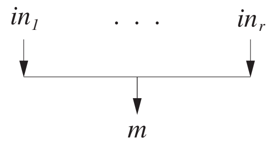
the memory access relocatable by introducing identifier addresses which are relative to daemons as base addresses.

4.1. Flowchart linearization

The goal of our paper is to prove the correctness of compiling specifications and not the correctness of compiler optimization techniques. Therefore the attention is focused on what happens to *atomic* Occam instructions during the compilation. As a consequence the use of the tree structure for the optimization of the code compiled for the given Occam program is not relevant for our analysis. This is the reason why we deviate from the usual practice in compilers and linearize the given flowchart at the very beginning of our analysis (instead of keeping the linearization of the generated instruction blocks for the end of the compilation).

In the sequel let S be an arbitrary Occam program and $flowchart(S)$ the flowchart by which the ground model $OCCAM_{ground}$ is initialized (see 7.1.). We transform $flowchart(S)$ into a sequence $lin(S)$ of nodes which are marked with atomic Occam instructions or with new goto-instructions whenever those are needed to avoid multiple in/outgoing edges. We refine correspondingly the $OCCAM_{ground}$ – rules for branching instructions – i.e. if, par, alt_s; adding also a new goto-rule we obtain a new evolving algebra $OCCAM_{lin}$ which if initialized by $lin(S)$ is equivalent to $OCCAM_{ground}$ initialized by $flowchart(S)$.

The transformation follows standard techniques: through the linearization nodes which (at run time) are the next ones might physically not be any more the next ones; for such cases one has to introduce goto instructions. More formally: multiple incoming edges to a node m from nodes $in_1, \ldots, in_r$



are replaced by new nodes $n_1, \ldots, n_r$ marked by the new instruction goto(m); multiple outgoing edges, say from a node n with $cmd(n) = instr$ to nodes $m_1, \ldots, m_q$ are replaced by assigning a new instruction $instr(m_1, \ldots, m_q)$ to n with a modified rule which will assume the intended flow control. The precise form depends on the four possible cases for $instr$ where multiple outgoing edges can occur in $flowchart(S)$, namely IF, WHILE, PAR, ALT.

For a smooth transition (in 4.2.) from the linearized flowchart to the use of a compilation function whose result is loaded into $NODE$ we treat the parameters for the modified instructions as "labels" to which an auxiliary function:

$$labeled\_loc \quad : \quad LABEL \rightarrow NODE$$

will associate the intended target – location. In this way the new $OCCAM_{lin}$-function:

$$next \quad : \quad NODE \rightarrow NODE$$

becomes monadic and is related to the old binary $OCCAM_{ground}$ function $next$ by:

$$next(n, i) \quad = \quad \begin{cases} next(n) & \text{if } i = 0 \\ labeled\_loc(l_i) & \text{otherwise} \end{cases}$$

In the case of nodes with an if instruction, one out-

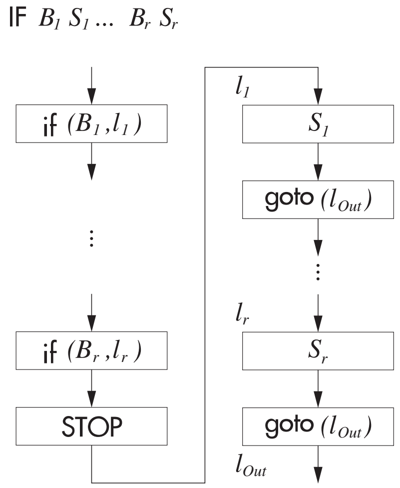


FIGURE 2. Flowchart Linearization for IF

going edge is chosen as *next*-exit, the other exit m is implemented by using a new node marked with the new instruction goto(m). For conventional reasons the choice of the *next*-exit depends on whether the if instruction comes from an IF (see Fig. 2) or from a WHILE (see Fig. 3) statement in the original Occam program; the *next*-exit is the no-exit in the first and the yes-exit in the second case. For flowcharts correspond-

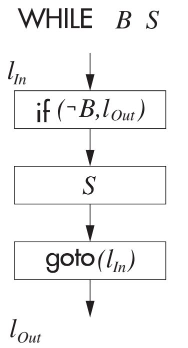


FIGURE 3. Flowchart Linearization for WHILE

ing to a statement of form PAR $S_1 \ldots S_r$ (see Fig. 4) or ALT $G_1\, S_1 \ldots G_r\, S_r$ (see Fig. 5) the linear sequences constructed by induction for the subprograms are connected in the order in which the latter appear in the Occam program. The correct transfer of control to the entry node of $lin(S_i)$ or $lin(G_i, S_i)$ labeled by l_i is implemented by the additional assignment of $labeled\_loc(l_i)$ as location in the modified par and alt rule. Formally

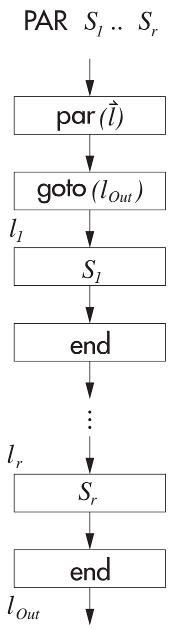


FIGURE 4. Flowchart Linearization for PAR

this linearized flowchart can be generated by a simple system of EA rules, similar to the flowchart generating rules given in appendix 7.1.. We leave this as an exercise to the reader. The new $OCCAM_{lin}$ algebra has new rules for three refined instructions (if, alt_s and par of the preceding section) and for one new instruction (goto). In the branching rules the additional parameter l is introduced together with a location update by $labeled\_loc(l)$.

$\boxed{\text{if(b,l)}}$ **if** *cmd is* if(b, l)
then if $eval(b, env(a))$
  then $loc(a)$:= $labeled\_loc(l)$
  else *proceed*

$\boxed{\text{goto(l)}}$ **if** *cmd is* goto(l)
then $loc(a)$:= $labeled\_loc(l)$

$\boxed{\text{par}(\vec{l})}$ **if** *cmd is* par$(l_1, \ldots, l_r)$
then extend *DAEMON* **by** $x_1 \ldots x_r$ **with**
  $q := q.x_1. \ldots .x_r$
  ...
  $father(x_i)$:= a
  $loc(x_i)$:= $labeled\_loc(l_i)$
  $env(x_i)$:= $env(a)$
  ...
  $count(a)$:= r
  sleep at $next(loc(a))$
endextend
where $1 \leq i \leq r$

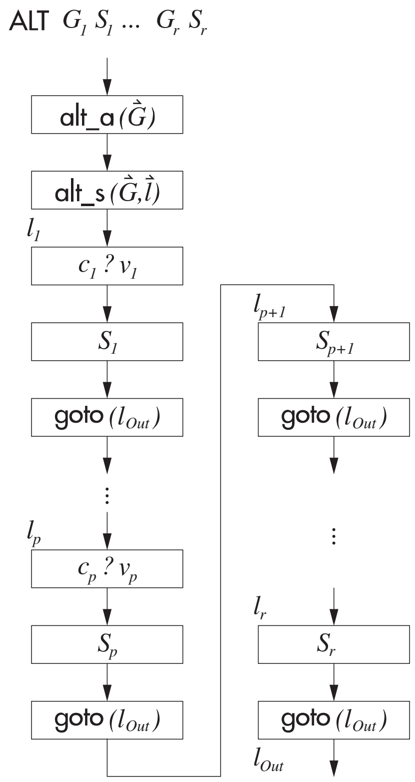


FIGURE 5. Flowchart Linearization for ALT

$\boxed{\text{alt\_s\_com}}$ | $\boxed{\text{alt\_s\_time}}$ | $\boxed{\text{alt\_s\_skip}}$ $(i, \vec{G}, \vec{l})$
if *cmd is* alt_s$(\vec{G}, \vec{l})$
  $\wedge\ eval(b_i, env(a))$
  $\wedge\ agent(\bar{c}_i) \notin \{ nil, a \} \mid timer > eval(t_i, env(a)) \mid$
then
  $disable(b_1, \bar{c}_1)$
  ...
  $disable(b_p, \bar{c}_p)$
  $loc(a)$:= $labeled\_loc(l_i)$
where
  $\vec{l} = l_1, \ldots, l_r$
  $\vec{G} = G_1, \ldots, G_r$
  $G_i = b_i\ :\ (\ c_i\ ?\ v_i \mid \text{TIME ? AFTER}\ t_i \mid \text{SKIP}\ )$

The value p is the count of input guards appearing in the ALT construct.

It is easy to define the notion of corresponding runs and of the equivalence between $OCCAM_{ground}$ on *flowchart*(S) and $OCCAM_{lin}$ on $lin(S)$ where homonymous rules — corresponding to each other via $next(n, i) = labeled\_loc(l_i)$ — have the same effect. This allows us to prove the following proposition:

PROPOSITION 4.1. *Corresponding runs of $OCCAM_{ground}$ on flowchart(S) and of $OCCAM_{lin}$ on $lin(S)$ are equivalent.*

COROLLARY 4.2. *Corresponding runs of $OCCAM_{ground}$ on flowchart(S) and of $OCCAM_{lin}$ on $lin(S)$ preserve daemons together with their environ-*

ment and the values of variables appearing there, communication traces, termination, deadlock and divergence.

4.2. Compilation and Loading of programs

In this section we describe the generation of the linearized flowchart $lin(S)$ through an abstract compilation function:

$$compile \;:\; STATEMENT \rightarrow (CODE \cup LABEL)^*$$

In order to achieve a simple definition of *compile* we will use the natural and quite standard recursion on the program structure. Therefore we separate the definition of $compile(S)$ from its "loading", i.e. from the generation of nodes which will be marked by the atomic instructions in $compile(S)$. As a consequence we separate now the syntactical status of labels l — to be names for parameters of instructions goto, if, par, alt_s — from their semantical interpretation as nodes $labeled\_loc(l)$ associated to them during the loading process. The loading itself is described by two new rules which constitute the $LOAD$ algebra: they assign instructions to the next free node – "memory location" *loading_loc* – and assign to labels the current loading location *loading_loc* as value (to which value the next compiled instruction will be associated).

Formally the $LOAD$ algebra has the two new dynamic functions:

$$\begin{aligned} loading\_loc &: NODE \\ load\_prg &: (CODE \cup LABEL)^* \end{aligned}$$

for the current loading location and the remaining program to be loaded which are thought to be initialized by *begin* – the initial node where the program execution starts – and $compile(S)$ respectively. The two rules are:

$\boxed{\text{load\_label(l,t)}}$
if $load\_prg = [\,l\,|\,t\,]$
  $\wedge\; l \in LABEL$
then $labeled\_loc(l)$ **:=** $loading\_loc$
    $load\_prg$ **:=** t

$\boxed{\text{load\_cmd(c,t)}}$
if $load\_prg = [\,c\,|\,t\,]$
  $\wedge\; c \in CODE$
then $cmd(loading\_loc)$ **:=** c
    $load\_prg$ **:=** t
    extend $NODE$ **by** n **with**
     $next(loading\_loc)$ **:=** n
     $loading\_loc$ **:=** n
    endextend

In defining now the compilation function by induction on S we tacitly assume that each time where labels appear they are distinct from each other and fresh for the compilation process; this assumption reflects the fact that semantically the labels have to represent different nodes in $lin(S)$, generated during the construction of the (linearized) flowchart. We abstain from the routine formalization of this assumption.

$$\begin{aligned} compile(\mathsf{SKIP}) &= \mathsf{SKIP} \\ compile(\mathsf{STOP}) &= \mathsf{STOP} \\ compile(v := t) &= v := t \\ compile(\mathsf{TIME}\,?\,v) &= \mathsf{TIME}\,?\,v \\ compile(c\,!\,t) &= c\,!\,t \\ compile(c\,?\,v) &= c\,?\,v \end{aligned}$$

$compile(\mathsf{VAR}\ id_1, \ldots, id_r : S)$ =
  $\mathsf{decl\_var}(id_1, \ldots, id_r)$,
  $compile(S)$,
  $\mathsf{decl\_end}(r)$

$compile(\mathsf{CHAN}\ id_1, \ldots, id_r : S)$ =
  $\mathsf{decl\_chan}(id_1, \ldots, id_r)$,
  $compile(S)$,
  $\mathsf{decl\_end}(r)$

$compile(\mathsf{IF}\ B_1\, S_1\ \ldots\ B_r\, S_r)$ =
  ...
  $\mathsf{if}(B_i, l_i)$,
  ...
  $compile(\mathsf{STOP})$,
  ...
  l_i,
  $compile(S_i)$,
  $\mathsf{goto}(l_{Out})$,
  ...
  l_{Out}
where $l_1, \ldots, l_r, l_{Out}$ are new labels
   $1 \leq i \leq r$.

$compile(\mathsf{WHILE}\ B\, S)$ =
  l_{In},
  $\mathsf{if}(\neg B, l_{Out})$,
  $compile(S)$,
  $\mathsf{goto}(l_{In})$,
  l_{Out}
where l_{In}, l_{Out} are new labels.

$compile(\mathsf{PAR}\ S_1 \ldots S_r)$ =
  $\mathsf{par}(l_1, \ldots, l_r)$,
  $\mathsf{goto}(l_{Out})$,
  ...
  l_i,
  $compile(S_i)$,
  end,
  ...
  l_{Out}
where $l_1, \ldots, l_r, l_{Out}$ are new labels
   $1 \leq i \leq r$.

$compile(\mathsf{SEQ}\ S_1 \ldots S_r)$ =
$compile(S_1)$,
...,
$compile(S_r)$

$compile(\mathsf{ALT}\ G_1\, S_1 \ldots G_r\, S_r)$ =
$\mathsf{alt\_a}(\vec{G})$,
$\mathsf{alt\_s}(\vec{G}, \vec{l})$,
...
l_i,
$compile(c_i\ ?\ v_i)$,
$compile(S_i)$,
$\mathsf{goto}(l_{Out})$,
...
l_j,
$compile(S_j)$,
$\mathsf{goto}(l_{Out})$,
...
l_{Out}
where $\vec{l} = l_1, \ldots, l_r$ are new labels
$\vec{G} = G_1, \ldots, G_r$
$G_i = b_i\ :\ c_i\ ?\ v_i$
$G_j = b_j\ :\ (\ \mathsf{TIME}\ ?\ \mathsf{AFTER}\, t_j$ or $\mathsf{SKIP}\ )$
$1 \leq i \leq p < j \leq r$.

Using the convention on fresh labels during the computation of $compile(S)$ it is easy to see that the following is true:

Proposition 4.3. *The LOAD algebra, started with $load\_prg = compile(S)$, computes a sequence of nodes marked with* Occam *instructions which is isomorphic to $lin(S)$.*

Proof. Induction on S, observing that labels l_i occurring during the recursive computation of $compile(S)$ correspond to the nodes $labeled\_loc(l_i)$ associated to them by the loading rules.

Corollary 4.4. *Let $OCCAM_{compile}$ be the union of $OCCAM_{lin}$ and the LOAD algebra. The runs of $OCCAM_{lin}$ on $lin(S)$ and of $OCCAM_{compile}$ started with $load\_prg = compile(S)$ are equivalent. Therefore they preserve daemons together with their environments and the values of the variables appearing there, communication traces, termination, deadlock and divergence.*

You would probably have expected that we define $OCCAM_{compile}$ by using the modification of $OCCAM_{lin}$ where the all rules get the additional guard $load\_prg =$ [ ]. As a matter of fact it is irrelevant whether applications of $OCCAM_{lin}$ rules and of $LOAD$ rules are interleaved or not. ∎

Remark 4.1. Once the flowcharts have been linearized and the programs are compiled and loaded, it is of no help any more to speak of nodes as placeholders for instructions. In the following we switch therefore to the following new notation and naming which brings us closer to the intuition of the Transputer memory:

$$NODE = LOC \quad next = +1 \quad begin = 0.$$

Since at this level of abstraction we do not care about efficient use of locations, we assume $+1 : LOC \rightarrow LOC$ to be a total (if you wish even injective) function on locations (we will also use the binary + function defined in the usual way from +1). As a consequence the extend update in the load_cmd rule can be replaced equivalently by:

$$loading\_loc \quad := \quad loading\_loc + 1.$$

As further step towards the Transputer we introduce the Transputer store abstractly by a function:

$$content \ :\ LOC \rightarrow VAL$$

which yields the value stored in a location. From now on we consider programs to be "stored in memory" by requiring cmd to be subfunction of $content$. Clearly these cosmetic changes (pure data refinements) of the $OCCAM_{compile}$ algebra do not effect the truth of proposition 4.3.

4.3. Environment Compilation

This section is devoted to the compilation of the environment which however still remains abstract and will be refined in the next section.

For the interpreter model $OCCAM_{ground}$ it did pay out to let daemons carry their environment; daemons only have to extend or shrink their current environment upon execution of a declaration begin or end instruction. In order to obtain efficient code the management of environments is assigned as much as possible to the compiler. Indeed for the Occam language an environment is really determined by the structure of the piece of program to which it belongs, namely it is the result of the still active declarations which lie on the path of locations (instructions) traversed by the executing daemon.

Therefore the environments can be computed in advance by the compilation function and be used at run time by the daemons as context for the execution of atomic Occam instructions. This means that $env(a)$ is replaced by an environment parameter e of the atomic Occam instructions, computed by $compile$.

This is the idea for the following refinement of $OCCAM_{compile}$ into the evolving algebra $OCCAM_{env}$. The function $compile$ receives as additional parameter the environment within which the (sub)program has to be executed:

$$compile \ :\ STATEMENT \times ENV \rightarrow ((CODE \times ENV) \cup LABEL)^*.$$

For better readability we will write $code(e)$ instead of $(code,\ e)$ and suppress the environment parameter for code where it doesn't matter (i.e. for SKIP, STOP, goto, end).

The rules decl_var, decl_chan, decl_end of $OCCAM_{compile}$ in section 3. are deleted because their effect is now computed by the following clauses which refine the clauses for declarations of the function *compile* of the preceding section. As we did already for labels, we abstain from the routine formalization of the creation of "new" variables and channels.

$$
\begin{aligned}
& compile(\mathsf{VAR}\ id_1, \ldots, id_r : S, e) = compile(S, e') \\
& \textbf{where}\ e' = append(e_{new}, e) \\
& \qquad e_{new} = (id_r, v_r), \ldots, (id_1, v_1) \\
& \qquad v_1, \ldots, v_r \text{ are "new" variables}
\end{aligned}
$$

$$
\begin{aligned}
& compile(\mathsf{CHAN}\ id_1, \ldots, id_r : S, e) = \\
& \quad \mathsf{init\_chan}(id_1, \ldots, id_r, e'), \\
& \quad compile(S, e') \\
& \textbf{where}\ e' = append(e_{new}, e) \\
& \qquad e_{new} = (id_r, c_r), \ldots, (id_1, c_1) \\
& \qquad c_1, \ldots, c_r \text{ are "new" channels}
\end{aligned}
$$

Note that our compilation of declarations "allocates" statically variables and channels. During the run–time they are *reused*, therefore the initialization of channels has to be re–done every time a channel is "created". Obviously this cannot be done at compile time, therefore we have to generate an init_chan instruction with a rule which executes the updates (initializations) of the decl_chan instruction of $OCCAM_{compile}$:

$$
\begin{aligned}
& \textbf{if}\ cmd\ is\ \mathsf{init\_chan}(c_1, \ldots, c_r, e) \\
& \textbf{then}\ agent(\bar{c_1}) := nil \\
& \qquad \ldots \\
& \qquad agent(\bar{c_r}) := nil \\
& \textbf{where}\ \bar{c_i} = bind(c_i, e)
\end{aligned}
$$

Note also that the generation of the instruction decl_end in $OCCAM_{compile}$ in section 3. can be eliminated because e is the "restored" environment of e'.

The compilation of atomic Occam instructions whose semantics depends upon the environment generates corresponding instructions which refine the instructions generated in the preceding subsection by passing to them also the environment parameter:

$$
\begin{aligned}
compile(v := t, e) &= \mathsf{ass}(v, t, e) \\
compile(\mathsf{TIME}\,?\,v, e) &= \mathsf{time}(v, e) \\
compile(c\,!\,t, e) &= \mathsf{out}(c, t, e) \\
compile(c\,?\,v, e) &= \mathsf{inp}(c, v, e)
\end{aligned}
$$

The rules for those refined instructions are obtained from corresponding $OCCAM_{compile}$ rules of section 3. by using the generated environment parameter e instead of $env(a)$ (except for the instructions SKIP, STOP, goto, end where nothing does change because their semantics does not depend upon the environment):

$\boxed{\mathsf{ass}(v,t,e)}$

$$
\begin{aligned}
& \textbf{if}\ cmd\ is\ \mathsf{ass}(v, t, e) \\
& \textbf{then}\ eval(\bar{v}) := eval(t, e) \\
& \qquad proceed
\end{aligned}
$$

$\boxed{\mathsf{time}(v,e)}$

$$
\begin{aligned}
& \textbf{if}\ cmd\ is\ \mathsf{time}(v, e) \\
& \textbf{then}\ eval(\bar{v}) := timer \\
& \qquad proceed
\end{aligned}
$$

$\boxed{\mathsf{if}(b,l,e)}$

$$
\begin{aligned}
& \textbf{if}\ cmd\ is\ \mathsf{if}(b, l, e) \\
& \textbf{then if}\ eval(b, e) \\
& \qquad \textbf{then}\ loc(a) := labeled\_loc(l) \\
& \qquad \textbf{else}\ loc(a) := next(loc(a))
\end{aligned}
$$

Analogously for the refined rules for input and output and for alt_a and alt_s. For WHILE, IF, ALT, SEQ, PAR we have the same compilation as in $OCCAM_{compile}$ (see section 3.), adding to *compile* and to the generated instructions the parameter e. We assume as before that each generated label is "new".

$$
\begin{aligned}
& compile(\mathsf{WHILE}\ B\ S, e) = \\
& \quad l_{In}, \\
& \quad \mathsf{if}(\neg B, l_{Out}, e), \\
& \quad compile(S, e), \\
& \quad \mathsf{goto}(l_{In}), \\
& \quad l_{Out}
\end{aligned}
$$

$$
\begin{aligned}
& compile(\mathsf{IF}\ B_1\, S_1\ \ldots\ B_r\, S_r, e) = \\
& \quad \ldots \\
& \quad \mathsf{if}(B_i, l_i, e), \\
& \quad \ldots \\
& \quad compile(\mathsf{STOP}, e), \\
& \quad \ldots \\
& \quad l_j, \\
& \quad compile(S_j, e), \\
& \quad \mathsf{goto}(l_{Out}), \\
& \quad \ldots \\
& \quad l_{Out} \\
& \textbf{where}\ 1 \leq i, j \leq r
\end{aligned}
$$

Analogously $compile(\mathsf{SEQ}\ S_1, \ldots, S_r)$ is refined introducing the parameter e.

$$
\begin{array}{l}
compile(\mathsf{ALT}\ G_1\,S_1 \ldots G_r\,S_r, e) \ = \\
\quad \mathsf{alt\_a}(\vec{G}, e), \\
\quad \mathsf{alt\_s}(\vec{G}, \vec{l}, e), \\
\quad \ldots \\
\quad l_i, \\
\quad compile(c_i\,?\,v_i, e), \\
\quad compile(S_i, e), \\
\quad \mathsf{goto}(l_{Out}), \\
\quad \ldots \\
\quad l_j, \\
\quad compile(S_j, e), \\
\quad \mathsf{goto}(l_{Out}), \\
\quad \ldots \\
\quad l_{Out} \\
\textbf{where } \vec{l} = l_1, \ldots, l_r \\
\qquad \vec{G} = G_1, \ldots, G_r \\
\qquad G_i = b_i\ :\ c_i\,?\,v_i \\
\qquad G_j = b_j\ :\ (\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j \text{ or } \mathsf{SKIP}\ ) \\
\qquad 1 \le i \le p < j \le r
\end{array}
$$

Analogously $compile(\mathsf{PAR}\ S_1, \ldots, S_r)$ is refined by introducing the parameter e. In the refined rule for par the update of the environment function env is deleted.

It is easy (see the proof below) to define the correspondence of runs of $OCCAM_{compile}$ and the refined evolving algebra $OCCAM_{env}$ started with $compile(S)$ and $compile(S, [\,])$ respectively.

PROPOSITION 4.5. *Corresponding runs of $OCCAM_{compile}$ and of $OCCAM_{env}$, started with $compile(S)$ and $compile(S, [\,])$ respectively, are equivalent. Therefore they preserve daemons, environments, the values of variables appearing there, communication traces, termination, deadlock and convergence.*

Proof. By induction on the structure of S. The linear structure of the compiled code is the same in both algebras, except for the declaration code of $OCCAM_{compile}$. No such code is generated in $OCCAM_{env}$ where instead $compile$ – assisted by the channel initializing instruction – provides the environment computed in $OCCAM_{compile}$ at run time by the declaration code. Homonymous rules in both algebras have the same effect. The environment $env(a)$ within which an instruction is executed in $OCCAM_{compile}$ is the same as the environment parameter e of the corresponding instruction in the corresponding run of $OCCAM_{env}$. ∎

In the sequel our compilation function will undergo further refinements. Besides instructions and environments several other entities will be generated like labels, daemons, etc. We use them as parameters in very the same way as we did with e. Therefore they are to be understood without further mentioning to go together with the introduction or refinement of corresponding dynamic functions which will be updated correctly in the $LOAD$ algebra.

REMARK 4.2. Note that for the compilation of environments in $OCCAM_{env}$ we assume fresh variables to have value *undef*, as is the case when new variables are created in $OCCAM_{compile}$. In any case it is hardly the compiler which should be blamed if a variable used by a program is wrongly initialized.

4.4. Refining variables and channels

In this section we implement variables and channels as locations. Thus the abstract environments become mappings to blocks of memory. Formally we require:

$$VAR, CHANNEL \ \subseteq\ LOC$$

and refine the variable evaluation function *eval* and the function *agent* to be subfunctions of *content*. Due to the static memory allocation for Occam programs we can store environments internally as blocks of memory. As preparation for the refinement to relocatable memory access in the next subsection we provide the compilation function with an additional parameter m for the next free memory position starting from which the environment will be stored in a contiguous memory area:

$$
\begin{array}{lll}
compile & : & STATEMENT \times ENV \times LOC \\
& \to & ((CODE \times ENV) \cup LABEL)^*
\end{array}
$$

The recursive definition of the placement of environments into memory by the compilation function is assisted by an auxiliary function:

$$env\_size \ :\ STATEMENT \to \mathbf{N}$$

which yields the number of memory locations needed by a subprogram for its variables and channels. This function is also recursively defined; the definition takes advantage from the decision to place environments into contiguous memory positions. We first give this recursive definition and then show how the refinement of *compile* affects the defining clauses for declarations and PAR-statements. Through declarations, the environments are extended:

$$
\begin{array}{lll}
env\_size(\mathsf{VAR}\ id_1, \ldots, id_r\ :\ S) & = & r + env\_size(S) \\
env\_size(\mathsf{CHAN}\ id_1, \ldots, id_r\ :\ S) & = & r + env\_size(S).
\end{array}
$$

Some statements don't introduce new variables, therefore in those cases we have:

$$
\begin{array}{lll}
env\_size(\mathsf{SKIP}) & = & 0 \\
env\_size(\mathsf{STOP}) & = & 0 \\
env\_size(\mathsf{TIME}\,?\,v) & = & 0 \\
env\_size(c\,?\,v) & = & 0
\end{array}
$$

For output and assignment statements additional (temporary) variables might be needed through the still abstract evaluation. Thus we require:

$$env\_size(c\,!\,t) \quad = \quad eval\_size(t)$$
$$env\_size(v := t) \quad = \quad eval\_size(t)$$

where *eval_size* is an auxiliary function which will be specified together with expression evaluation.

Sequential execution of SEQ components requires that the reserved place must cover the maximal space requirement among the components. Therefore:

$$env\_size(\mathsf{SEQ}\ S_1 \ldots S_r) \quad = \quad max_{i=1}^{r}\ env\_size(S_i)$$

Also statements operating on boolean expression(s) may need some additional temporary variables for evaluation; this explains the definition of *eval_size* for ALT, IF and WHILE:

$$env\_size(\mathsf{ALT}\ G_1\,S_1 \ldots G_r\,S_r) = \\ max_{i=1}^{r}\ (eval\_size(b_i), env\_size(S_i))$$

$$env\_size(\mathsf{IF}\ B_1\,S_1, \ldots B_r\,S_r) = \\ max_{i=1}^{r}\ (eval\_size(B_i), env\_size(S_i))$$

$$env\_size(\mathsf{WHILE}\ B\,S) = \\ max(\ eval\_size(B), env\_size(S)\ )$$

In the case of a PAR program statement, its component statements run *concurrently*, so their variables/channels must be present concurrently. Therefore the *env_size* sums the size of the components:

$$env\_size(\mathsf{PAR}\ S_1 \ldots S_r) = \sum_{i=1}^{r} env\_size(S_i).$$

During the compilation of declarations the first free memory location has to be updated (advanced to the next location) to express the effect of the "location in use". Thus the compilation of variable declarations of the preceding section is refined as follows.

$$\begin{array}{l} compile(\mathsf{VAR}\ id_1, \ldots, id_r\ :\ S, e, m) = \\ \quad compile(S, e', m + r) \\ \textbf{where} \\ \quad e' = append(e_{new}, e) \\ \quad e_{new} = (id_r, m + r - 1), \ldots, (id_1, m). \end{array}$$

The refinement of the compilation function is similar for all other cases except for PAR. During the compilation of PAR-statements we have to pass now to each son a different free memory location for his private declarations. The requirement that all variables/channels have to be placed in a contiguous block comes handy now. We simply give each son enough space (determined using the value of *env_size* for the subprograms of the sons). Hence the compilation of PAR statements of the preceding subsection is refined as follows:

$$\begin{array}{l} compile(\mathsf{PAR}\ S_1 \ldots S_r, e, m) = \\ \quad \mathsf{par}(\vec{l}), \\ \quad \mathsf{goto}(l_{Out}), \\ \quad \ldots \\ \quad l_i, \\ \quad compile(S_i, e, m_i), \\ \quad \mathsf{end}, \\ \quad \ldots \\ \quad l_{Out} \\ \textbf{where}\ \vec{l} = l_1, \ldots, l_r \\ \quad m_1 = m \\ \quad m_j = m_{j-1} + env\_size(S_{j-1}) \\ \quad 1 \le i \le r \\ \quad 2 \le j \le r \end{array}$$

The environment space splitting is shown in figure 6. The space is reserved by the $LOAD$ algebra by setting *loading_loc* initially to $env\_size(S)$, and passing 0 as first free location for the compilation function. In this way the instructions are loaded immediately after the space dedicated to the environment, i.e. starting at $env\_size(S)$. The refinement $OCCAM_{env'}$

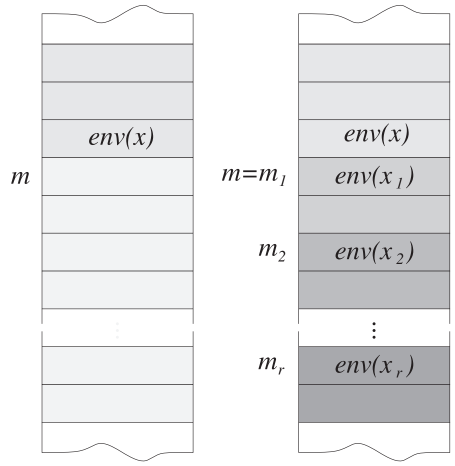


FIGURE 6. Splitting of space among new daemons x_i

of $OCCAM_{env}$ obtained by placing environments into memory is a pure data refinement for *compile* and for the initialization by $compile(s, [\,], 0)$. It does not involve new rules.

Proposition 4.6. *Proposition 4.5 is true for $OCCAM_{env'}$ modulo the refinement of variables and channels.*

4.5. Relative addressing for environments

In this section we formalize a standard technique to introduce relative addressing for bindings of identifiers. The goal is to make the environment relocatable.

In the Occam ground model the role of daemons is that of agents who execute Occam programs. In the

Transputer this role is reduced to that of holding the base address of the corresponding environment. Indeed the compilation function does not need to know (and in a real system cannot know) where the memory block for the environment will begin at run time.

For the compilation it is sufficient "to assume" a parameter x as base address with respect to which addresses are calculated as distances $n - x$ from x. At run time the actual address can then be calculated by adding this distance to the current base address, namely the active daemon a in our model. Making the addressing of locations mapped to identifiers relative to a daemon x as base address means that the execution of a piece of code "by x" is done by the Transputer through a computation in which the locations mapped to identifiers are accessed through x.

To realize this idea we augment the compilation function of the preceding and the binding function of section 3. with an additional parameter for the environment base address ("daemon") and place daemons into memory by requiring $DAEMON \subseteq LOC$:

$$\begin{array}{rcl} compile & : & STATEMENT \times ENV \times LOC \times DAEMON \\ & \rightarrow & ((CODE \times ENV \times DAEMON) \cup LABEL)^* \\ bind & : & ID \times ENV \times DAEMON \rightarrow LOC \end{array}$$

For the refined code produced by the new compilation function we put again the new daemon parameter as argument of the original code — writing *code(x)* instead of *(code, x)* — and suppress the parameter x in instructions where it plays no role.

The new function *compile* is obtained from the function of $OCCAM_{env'}$ by passing the new parameter x uniformly from arguments to rules whenever the base address (daemon) does not change. The scheme is illustrated for assignment. The compilation of assignments in subsection 4.3. is refined as follows:

$$compile(v := t, e, m, x) \quad = \quad \mathsf{ass}(v, t, e, x)$$

The new rule for assignment instructions will use the compile-time parameter x to calculate the distance and from that the correct run-time address of the correct binding of v and the variables in t (see below). New daemons appear only as consequence of the execution of PAR instructions where the new compilation function will compute these new base addresses. It is natural to define the i^{th} son as the memory position m_i which had been assigned by the compilation function of $OCCAM_{env'}$ for his private declarations. Hence the compilation of PAR statements of the preceding subsection is refined as follows:

$$\begin{array}{l} compile(\mathsf{PAR}\ S_1 \ldots S_r, e, m, x) \ = \\ \quad \mathsf{par}(x, \vec{m}, \vec{l}), \\ \quad \mathsf{goto}(l_{Out}), \\ \quad \ldots \\ \quad l_i, \\ \quad compile(S_i, e, m_i, m_i), \\ \quad \mathsf{end}, \\ \quad \ldots \\ \quad l_{Out} \\ \textbf{where} \\ \quad \vec{l} = l_1, \ldots, l_r \\ \quad \vec{m} = m_1, \ldots, m_r \\ \quad m_1 = m \\ \quad m_j = m_{j-1} + env\_size(S_{j-1}) \\ \quad 1 \le i \le r \\ \quad 2 \le j \le r \end{array}$$

The new rule for $\mathsf{par}(x, \vec{m}, \vec{l})$ will use the compile time distance $x_i - x$ between father x and son x_i to calculate the correct run-time base address $a + (x_i - x)$ for each i.

Similarly the new binding function computes the distance:

$$bind(id, e, x) \quad = \quad bind(id, e) - x$$

between the absolute addresses $bind(id, e)$ – computed by the binding function of $OCCAM_{env'}$ – and the base address x. In this way the relation between compile time and run time addresses is characterized by the simple equation (see Fig. 7):

run-time address = compile-time relative address + a

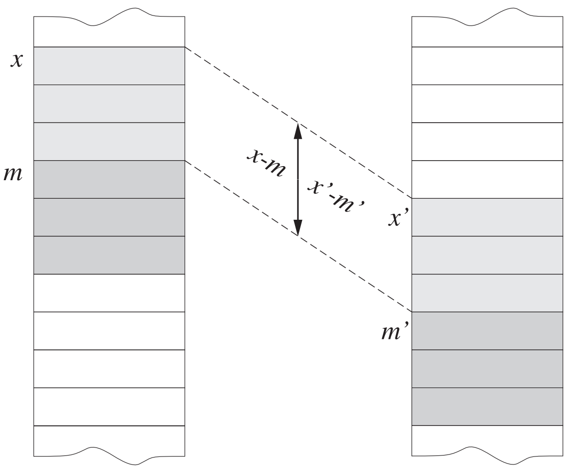


FIGURE 7. Assumed and Real addresses

Therefore the rules of the new evolving algebra $OCCAM_{rel-env}$ with relocatable environments are obtained from the rules of $OCCAM_{env'}$ as follows. Whenever $bind(id, e)$ is used it is replaced by $bind(id, e, x)$ with the appropriate x, given by the rule context.

Whenever $content(\bar{v})$ is used, it is replaced by $content(\bar{v}+a)$. A typical example is the assignment rule of subsection 4.3. which is refined as follows:

$\boxed{\mathsf{ass}(\mathrm{v,t,e,x})}$
if *cmd is* $\mathsf{ass}(v,t,e,x)$
then $content(\bar{v}+a)$:= $eval(t,e,x)$
  proceed

The relative addressing of a location associated to an identifier v in an environment e with respect to the compile-time base address x is reflected by the fact that at run time the distance $\bar{v} = bind(v,e,x)$ of the identifier's address from the compile time base address x is added to the run time base address, here the value of the active daemon a. Note that the expression evaluation function *eval* which has indirectly to call *bind* to get the values of the variables which occur in the expression has to be extended by an argument for the value of the daemon which has been assumed at compile time:

$$eval \quad : \quad EXP \times ENV \times DAEMON \rightarrow VAL.$$

The same refinement is done for time and input. The new rule for $\mathsf{par}(x,\vec{m},\vec{l})$ refines the corresponding rule in $OCCAM_{env'}$ (compare the formulation in subsection 4.1.) by computing the new daemons from the compile time parameters $x, \vec{m}$, i.e. the abstract creation of new daemons is replaced by a definition of the new base addresses they represent. The definition of the run time value x'_i for the i^{th} son adds the run-time value of the creating father – namely a – to the distance $x_i - x$ of the i^{th} son from its father which has been computed by the compilation function.

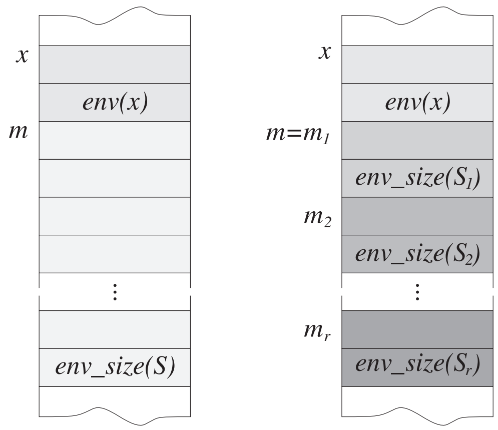


FIGURE 8. "Creation" of daemons in PAR

$\boxed{\mathsf{par}(x,\vec{x},\vec{l})}$
if *cmd is* $\mathsf{par}(x,\vec{x},\vec{l})$
then q := $q.x'_1 \ldots .x'_r$
  ...
  $father(x'_i)$:= a
  $loc(x'_i)$:= $labeled\_loc(l_i)$
  ...
  $count(a)$:= r
  sleep at $next(loc(a))$
where $\vec{l} = l_1,\ldots,l_r$
  $\vec{x} = x_1,\ldots,x_r$
  $x'_i = a + x_i - x$
  $1 \le i \le r$

Figure 8-a shows the interpretation of daemons as base addresses, figure 8-b illustrates the optimization of the next section where the father x takes over the role of his first son. The effect of the relative addressing is expressed by the following lemma.

Lemma 4.7. *Let m, x, m', x' be such that $m - x = m' - x'$. Then the computations of $OCCAM_{rel-env}$ started with $compile(S,[\,],m,x)$ and $compile(S,[\,],m',x')$ respectively are equivalent.*

Proof. The claim follows by induction an S and induction on the computations using the definition of the new binding function and the definition of new daemons for the PAR construct, which have been given in terms of distances to x. ■

Corollary 4.8. *Corresponding runs of $OCCAM_{env'}$ and $OCCAM_{rel-env}$, started with $compile(S,[\,],0)$ and $compile(S,[\,],0,0)$ respectively, are equivalent, preserving daemons, environments, the values of variables appearing there, communication traces, termination, deadlocks and convergence.*

4.6. Optimization for PAR statements

Figure 8 shows that when a daemon x creates sons $x_1,\ldots,x_k$, his next free memory position m coincides with the first free memory position m_1 of his first son x_1. Due to the fact that the father is sleeping while his sons are running we can reuse the father to do the work of his first son; indeed due to our compilation schema the environment of the first son is adjacent to the father's environment. In this way we can save the creation of one daemon. This idea can be realized by the following refinement of $OCCAM_{rel-env}$ to an optimized evolving algebra $OCCAM_{daemon}$. The assignment of the environment space is refined by not creating x_1 any more. The compilation function is refined on the PAR statement by eliminating the first element from $\vec{m}$ and $\vec{l}$ — i.e. creating only $k-1$ instead of k sons — and by eliminating the instruction $\mathsf{goto}(l_{Out})$, i.e. by letting the father proceed with the execution of the subprogram of the old first son. As a consequence the last (finishing) son must take up the role of his father instead of waking him up.

This implies refinements of the rules for the instructions par and end of the preceding subsection (see below).

$compile$(PAR $S_1 \ldots S_r, e, m, x$) =
  par($x, \vec{m}, \vec{l}, l_{Out}$),
  l_1,
  $compile(S_1, e, m, x)$,
  end,
  l_2,
  $compile(S_2, e, m_2, m_2)$,
  end,
  ...
  l_r,
  $compile(S_r, e, m_r, m_r)$,
  end,
  l_{Out}
where
  $\vec{l} = l_2, \ldots, l_r$
  $\vec{m} = m_2, \ldots, m_r$
  $m_1 = m$
  $m_i = m_{i-1} + env\_size(S_{i-1})$
  $2 \leq i \leq r$

When the last son finishes his work — i.e. when he executes the end instruction — he has to reset his base address to the base address of his father and to proceed with the program of his father. For this purpose the goto-parameter l_{Out} has to be made available. We store it through a function:

$$end\_par \quad : \quad DAEMON \rightarrow LABEL$$

The par-instruction is refined to include l_{Out} as parameter which upon execution of the par-rule is assigned to $end\_par(a)$; upon execution of the refined end-rule it will be used to let the last son take up the work of his father.

$\boxed{\text{par}(x, \vec{x}, \vec{l}, l_{Out})}$
if *cmd is* par($x, \vec{x}, \vec{l}, l_{Out}$)
then $q := q.x'_1. \ldots .x'_r$
  ...
  $father(x'_i) := a$
  $loc(x'_i) := labeled\_loc(l_i)$
  ...
  $count(a) := r$
  $end\_par(a) := l_{Out}$
  proceed
where $\vec{l} = l_1, \ldots, l_r$
  $\vec{x} = x_1, \ldots, x_r$
  $x'_i = a - x + x_i$
  $1 \leq i \leq r$

$\boxed{\text{end}}$
if *cmd is* end
then if $r > 1$
  then *dequeue*
    $r := r - 1$
  else $a := x_f$
    $loc(x_f) := a_{Out}$
where $r = count(x_f)$
  $x_f = father(a)$
  $l_{Out} = end\_par(x_f)$
  $a_{Out} = labeled\_loc(l_{Out})$

PROPOSITION 4.9. *Let $OCCAM_{daemon}$ be the optimization of $OCCAM_{rel-env}$, obtained by saving one son in the compilation of* PAR *statements. Runs of $OCCAM_{daemon}$ are equivalent to the corresponding runs of $OCCAM_{rel-env}$. Therefore environments, the values of variables appearing there, communication traces, deadlocks and divergence are preserved.*

Proof. The runs in the two algebras are in one-to-one correspondence: the same rules are applied although by different agents. Indeed fathers work for their first sons and they do this correctly, as assured by the refined clause for the *compile* function: x goes to execute the code of his first son (because the goto instruction has been eliminated); the last son instead of killing himself takes up the role of his father by reseting himself (the base address) to that of his father and his location to where the father had stopped to do his own job. The compilation of the variables of x_1 is done correctly because the environment produced for x_1 is the same as before, it is only accessed from the father's base address x instead of x_1. ■

We can summarize the results of this section by the following theorem:

THEOREM 4.10. *$OCCAM_{daemon}$ is a correct and complete implementation of $OCCAM_{ground}$, i.e. for each* Occam *program, its $OCCAM_{ground}$ semantics is the same as its $OCCAM_{daemon}$ semantics.*

5. Transputer (GROUND) MODEL

In this section we introduce the signature of the Transputer ground model which is relevant for the execution of the Transputer code to which the abstract code of the preceding section will be refined in the next section.

We introduce here the registers *Wreg* (for the current agent) and *Ireg* (for the pointer to the current instruction), the three (stack like) registers *Areg,Breg,Creg* for the management of instruction operands and the registers for the implementation of the queues q and $time\_q$. We also define which Transputer instructions are safe for time–slicing and introduce the instruction fetch mechanism which also takes care of the Transputer time–slicing mechanism.

5.1. Queue Implementation

Two special–purpose Transputer registers[11]:

$$FPtrReg, BPtrReg \quad : \quad DAEMON \cup \{ nil \}$$

known as *front/back* (queue) pointer registers, together with the function[12]:

$$next \quad : \quad DAEMON \rightarrow DAEMON$$

implement the abstract queues of $OCCAM_{daemon}$.

As a matter of fact, the latter can be recovered by the following definition of the list $queue(l)$:

$$q \;=\; queue(FPtrReg)$$
$$queue(l) \;=\; \begin{cases} l & \text{if } l = BPtrReg \\ l.queue(next(l)) & \text{otherwise} \end{cases}$$

(Remember that we use the dot for list notation.) Clearly $first(q) = FPtrReg$, $last(q) = BPtrReg$, $rest(q) = queue(next(FPtrReg))$. The queue is said to be empty if $FPtrReg = nil$; we assume that the queue registers are initialized with nil. The abstract updates for enqueueing and dequeuing of the Occam algebra in section 3. are now implemented by the following macro:

$$\begin{array}{ll} enqueue\ x \stackrel{\text{def}}{=} & BPtrReg := x \\ & \textbf{if } FPtrReg = nil \\ & \textbf{then } FPtrReg := x \\ & \textbf{else } next(BPtrReg) := x \end{array}$$

The macro *dequeue* formalizes the dequeuing mechanism. Note that if the process queue is empty, then the cpu is idle, waiting for some daemon to appear (either by completing an external communication or by an application of the time_wakeup rule).

$$\begin{array}{ll} dequeue \stackrel{\text{def}}{=} & \textbf{if } FPtrReg = nil \\ & \textbf{then } a := nil \\ & \textbf{else } a := FPtrReg \\ & \quad loc(a) := loc(FPtrReg) \\ & \quad start := timer \\ & \quad \textbf{if } FPtrReg = BPtrReg \\ & \quad \textbf{then } FPtrReg := nil \\ & \quad \textbf{else } FPtrReg := next(FPtrReg) \end{array}$$

Note that strictly speaking the update $loc(a) := loc(FPtrReg)$ is unecessary; we include it already here because in subsection 5.3. we refine $loc(a)$ to be the register *Ireg* which is independent of a.

It is a routine exercise to formalize and to prove the correctness and completeness statement for this implementation of queues of $OCCAM_{daemon}$. Note that this refinement is essentially a data refinement.

5.2. Time queue

The administration of time is implemented in the Transputer by Timer–hardware. It takes care of incrementing the *timer* function(s) and administrating the time queue.

The time queue is implemented by a 0-ary function[13] (a reserved location (register) Time Pointer Location):

$$TPtrLoc \quad : \quad DAEMON \cup \{ nil \}$$

together with the above indicated function *next*. Note that we are allowed to reuse the same *next* function for scheduler and timer queue because these two queues are always disjoint. Assume that an ALT process is waiting for a communication partner and for the time; at the moment he is woken up by his communication partner he is thrown out from the timer queue and put into the scheduler queue (see out_alt and OUT_alt rules). Clearly *time_q* can be recovered in the standard way from $TPtrLoc = first(time\_q)$ using *next*. In the Transputer we find for efficiency reasons also another reserved location (register):

$$TNextReg \quad : \quad LOC$$

to hold the time the process in *TPtrLoc* is waiting for (i.e. $TNextReg = t_{min}(TPtrLoc)$).

It is routine to refine the notions of *time-insert* and *time-delete* accordingly.

When the time recorded in *TNextReg* is reached the refined time_wakeup rule fires and wakes up the corresponding waiting daemons. The refined version of the time_wakeup rule of section 3. is the following:

$$\begin{array}{ll} \boxed{\text{time\_wakeup}} & \\ \textbf{if } TPtrLoc \neq nil \wedge timer > TNextReg & \\ \textbf{then } com\_mode(TPtrLoc) := alt\_run & \\ \qquad enqueue\ TPtrLoc & \\ \qquad time\text{-}delete\ TPtrLoc & \end{array}$$

We leave again as an exercise to the reader to formalize and prove the correctness and the completeness statement for this implementation of time queues — which is again a pure data refinement. Note that the time_wakeup rule runs concurrently to the other rules, thus reflecting that an independent piece of hardware is responsible for timing.

Note that on the basis of the notions of process queue and time queue and with appropriate register initialization we can define deadlock as process and timer queue being empty without any process waiting for an external communication.

5.3. Registers

The Transputer has three general purpose registers:

$$Areg, Breg, Creg \quad : \quad VAL$$

which behave like a stack, allowing us to access directly only the top of the stack, *Areg*[14]. Therefore we will use the following stack–like macros for accessing registers:

[11] Really two for each priority

[12] Not to be confused with the function *next* in the flowcharts of $OCCAM_{ground}$.

[13] Really for each priority

[14] By requiring that the registers take the values in *VAL* we abstract from the hardware implementation of registers; e.g. the

$push(v) \stackrel{\text{def}}{=} Areg := v,\ Breg := Areg,\ Creg := Breg$
$pop \stackrel{\text{def}}{=} Areg := Breg, Breg := Creg,\ Creg := random$
$pop(v) \stackrel{\text{def}}{=} v := Areg,\ pop$

In *pop* we use a 0-ary external function *random* which takes values in *VAL*. The update of *Creg* by *random* in *pop* formalizes the non–deterministic behavior of the hardware which may assign any value to *Creg*.

The Transputer has two special–purpose registers:

$$Wreg, Ireg \quad : \quad VAL$$

to hold the current agent and a pointer to the current instruction. From the *workspace (pointer) register Wreg* and the *instruction (pointer) register Ireg* we can recover a and *loc* of $OCCAM_{daemon}$ as follows:

$$\begin{aligned} a &\stackrel{\text{def}}{=} Wreg \\ loc(a) &\stackrel{\text{def}}{=} Ireg \end{aligned}$$

Correspondingly the macro *proceed* of previous sections becomes:

$$proceed \stackrel{\text{def}}{=} Ireg := Ireg + 1$$

Similarly the definitions and rules where a and $loc(a)$ appear have to be interpreted with *Wreg* and *Ireg* (see in particular the dequeue rules). For the time being it

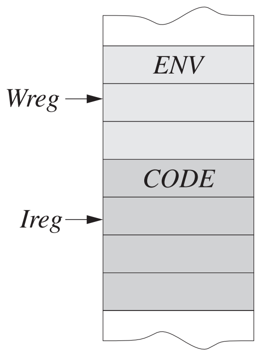


FIGURE 9. Example of *Wreg* and *Ireg*

suffices to assume that the general purpose Transputer registers range over a subdomain of *LOC* so that the function *content* can be applied to registers.

Since this introduction of Transputer registers is a pure data refinement, with respect to an appropriate register initialization it preserves the correctness and the completeness statement.

5.4. Time slicing

As a rule the scheduling of processes is a time consuming operation. In fact before actually starting a process one has to restore his context, represented by certain processor registers which have to be preserved on rescheduling. For the sake of efficiency this context — in a narrow sense — has been reduced in the Transputer to *Wreg* and *Ireg* only. The introduction of the notion of safe places defines "safe" points in a program starting from where the code doesn't depend on the context — in a broad sense — including values of *Areg*, *Breg* and *Creg* registers which have been generated before reaching such a point. It is the duty of the compilation function to meet the proposed requirement for being a safe place when the code is generated. At such safe places time-slicing can be applied correctly.

For time-slicing it is natural that its execution is coupled to the *fetch* phase in which, anyway, the CPU has to recognize the type of instruction to be executed. We formalize this by introducing a 0-ary function:

$$mode \quad : \quad \{fetch, execute\}$$

together with a function[15]:

$$cmd \quad : \quad CODE$$

which holds the instruction fetched by the CPU for execution so that $cmd = content(Ireg - 1)$.

Since all the rules defined so far have to do with the execution of instructions they are put under the additional guard $mode = execute$ by refining the macro *cmd is* of section 3. to:

$$cmd\ is\ c \stackrel{\text{def}}{=} cmd = c \wedge mode = execute \wedge Wreg \neq nil$$

and by adding to each rule the update $mode := fetch$ and deleting the update *proceed*. The time–slice rule of section 3. is refined correspondingly as follows:

```
[time–slice]
if mode = fetch ∧ Wreg ≠ nil
then if elapsed
        ∧ safe(content(Ireg))
     then seq enqueue Wreg
              dequeue
          endseq
     else cmd := content(Ireg)
          proceed
          mode := execute
```

Under the assumption that in an infinite run there are infinitely many *safe* instructions which are executed this new rule is a correct implementation of the previous time–slice rule. We reassume this section in the following proposition.

PROPOSITION 5.1. *The refinement of $OCCAM_{daemon}$ by implementing process and time queues and by introducing time–slicing as part of the fetch mechanism is correct and complete.*

Proof. It is easy to check comparing homonymous rules that the $OCCAM_{daemon}$ update *proceed* is handled correctly by the time–slicing rule. By definition a

implementation of the workspace register introduced below, in **T800** uses only 30 bits for values and one bit for priority.

[15] Not to be confused with the function *cmd* of the Occam algebra.

"safe" place in a program is represented by code which doesn't depend on the "context" in the broad sense generated before reaching that place but only on the "context" in the narrow sense. The "context" in the broad sense is represented in the Transputer by *Wreg*, *Ireg* and the three stack registers *Areg*, *Breg*, *Creg* for holding the parameters. The claim therefore follows by induction on the number of times when the time has elapsed for time–slicing, using the fact that Occam processes which run in parallel are independent and in particular do not compete for resources.

Note that the Transputer designers have defined the jump instruction (as well as the loop end instruction `LEND`) to be safe. (Note that our compilation function prevents pieces of code which follow a jump instruction to depend on the values of *Areg*, *Breg*, *Creg* unless these registers are set by that piece of code.) The compilation of WHILE constructs contains a jump so that no execution of a compiled WHILE program on the Transputer can diverge without containing infinitely many executions of jumps and therefore time–slicing. ■

6. COMPILATION INTO Transputer CODE

In this section we refine the *compile* function of section 4. to output instructions which are executed in the Transputer model defined in section 5.. We do this in three steps: first we compile into code which still uses the abstract auxiliary functions of $OCCAM_{daemon}$ but implements the relative addressing of variables and channels by the run–time calculation of the absolute addresses from the distance from the base address which has been calculated at compile–time. Then we implement the abstract auxiliary Occam functions by generalizing *Wreg* from the daemon's base address to the process workspace it represents (points to). We refine the *compile* function according to the workspace implementation and finally make the code relocatable by applying the technique of relative branching (relative addressing for target positions of branching instructions).

Our definition of the *compile* function really describes a specific compiler. However this definition is by no means unique. Any definition will do which satisfies the constraints which we use for our correctness proof.

In this section we make extensive use of the locality principle built into evolving algebras which allows us to proceed instructionwise. Instruction per instruction we define the new function *compile* together with the corresponding rule(s) for the execution of the produced target code and prove the correctness with respect to $OCCAM_{daemon}$. These local proofs are mostly trivial and altogether will prove the following theorem:

Theorem 6.1. *($TRANSPUTER_{ground}$ correctness) Arbitrary runs of $OCCAM_{daemon}$ on $compile_{daemon}$ $(S, [], 0, 0)$ are implemented correctly by corresponding runs of $TRANSPUTER_{ground}$ on $compile_{ground}(S, [], 0, 0)$ where $TRANSPUTER_{ground}$ is the evolving algebra defined in section 5. and 6.1. and where $compile_{daemon}$ and $compile_{ground}$ are the respective compile functions.*

From the following definition of the new rules for $TRANSPUTER_{ground}$ which execute the code produced by the new function *compile* it will be clear how to define the "correspondence" of runs on the basis of "local" correspondences of single $OCCAM_{daemon}$ instructions to short sequences of $TRANSPUTER_{ground}$ instructions.

Note that the theorem cannot be extended to completeness of corresponding runs due to the particular Transputer strategy for the computation of ALT.

6.1. Abstract Transputer compilation

In this section we define the compilation to Transputer instructions which still uses abstract auxiliary $OCCAM_{daemon}$ functions.

We proceed stepwise, defining for each Occam statement S the value of *compile* together with the $TRANSPUTER_{ground}$ rules for the execution of the code. Each time we show that this implements correctly the semantics of S as compiled to and executed in $OCCAM_{daemon}$.

Declarations

The compilation of variable declarations remains the same as in $OCCAM_{daemon}$. For the channel declarations (see subsection 4.3.) we have to compile the pseudo instruction init_chan for the initialization of channels to *nil*. This is realized by first loading *nil* into the register *Areg* (using the `MINT` instruction) and then storing it from there to the channel (using the local storing instruction `STL`) with appropriate address:

$$
\begin{aligned}
& compile(\mathsf{CHAN}\ id_1, \ldots, id_r : S, e, m, x) = \\
& \quad compile(\mathsf{init\_chan}(\vec{id}), e', m, x), \\
& \quad compile(S, e', m + r, x) \\
& \textbf{where } \vec{id} = id_1, \ldots, id_r \\
& \quad\quad e' = (id_r, m + r - 1), \ldots, (id_1, m), e
\end{aligned}
$$

$$
\begin{aligned}
& compile(\mathsf{init\_chan}(\vec{c}), e, m, x) = \\
& \quad \ldots \\
& \quad \mathtt{MINT}, \\
& \quad \mathtt{STL}\ bind(c_i, e, x), \\
& \quad \ldots \\
& \textbf{where } \vec{c} = c_1, \ldots, c_r \\
& \quad\quad 1 \le i \le r
\end{aligned}
$$

if *cmd is* `MINT` **then** *push*(*nil*)

if *cmd is* `STL` n **then** *pop*(*content*(*Wreg* + n))

Note that the binding function (see section 4.5.) yields the offset $bind(id, e, x)$ calculated at compile time with respect to the compile–time base address x; this relative address is used at run time by the instruction `STL` to calculate the absolute address by adding the current base address *Wreg*. In this way the environment becomes spurious at run-time.

Looking at the rule for init_chan in $OCCAM_{daemon}$ (see section 4.3.) it should become clear that the following correctness lemma holds for declaration statements:

LEMMA 6.2. (CORRECTNESS LEMMA) *For declaration statements S of* Occam, *the execution of the rules of $OCCAM_{daemon}$ on $compile_1(S, e, m, x)$ is correctly implemented by the execution of the rules of $TRANSPUTER_{ground}$ on $compile_2(S, e, m, x)$ where $compile_1$, $compile_2$ denote the function compile of $OCCAM_{daemon}$ and $TRANSPUTER_{ground}$ respectively.*

Assignment

We still keep the expression evaluation abstract by using an abstract instruction eval which stores the evaluation result in *Areg*. The compilation of $v := t$ has then to make sure that from *Areg* this value gets stored into the right variable. Remembering that for evaluation of variables we have $eval(v, e, x) = content(\bar{v} + Wreg)$, where $\bar{v} = bind(v, e, x)$, makes clear that the following compilation of assignment statements satisfies the correctness lemma 6.2 (see the compilation of assignments in section 4.5.); similarly for time statements where instead of an expression the current value of *timer*[16] is loaded into *Areg*.

$$\begin{array}{l} compile(v := t, e, m, x) = \\ \quad \mathsf{eval}(t, e, x) \\ \quad \texttt{STL}\ bind(v, e, x) \end{array}$$

$$\begin{array}{l} compile(\mathsf{TIME}\,?\,v, e, m, x) = \\ \quad \texttt{LDTIMER} \\ \quad \texttt{STL}\ bind(v, e, x) \end{array}$$

if *cmd is* eval(t, e, x)
then $push(eval(t, e, x))$

if *cmd is* LDTIMER
then $push(timer)$

There is no Transputer equivalent for the *no-operation* instruction SKIP, therefore it disappears:

$$compile(\mathsf{SKIP}, e, m, x) =$$

The STOP instruction is directly implemented by the Transputer instruction STOPP (see the stop rule in section 3.):

$$compile(\mathsf{STOP}, e, m, x) = \texttt{STOPP}$$

if *cmd is* STOPP **then** *Areg, Breg, Creg* := *random*
dequeue

Note that in our compilation schema the process executing STOPP will never be woken up, thus rendering the necessity of storing *Ireg* in the workspace — as it is really done in the Transputer — superfluous.

Obviously the correctness lemma 6.2 holds for the new compilation of SKIP and STOP.

The compilation of SEQ remains unchanged.

[16] The *timer* function for the priority of the currently active process is used.

Input

The compilation of an input statement starts with instructions LDLP which loads the absolute variable/channel addresses into the first two stack registers. Then the instruction IN uses the information stored there and actually takes the message from the channel and stores it into the variable.

$$\begin{array}{l} compile(c\,?\,v, e, m, x) = \\ \quad \texttt{LDLP}\ bind(c, e, x), \\ \quad \texttt{LDLP}\ bind(v, e, x), \\ \quad \texttt{IN} \end{array}$$

if *cmd is* LDLP n **then** $push(Wreg + n)$

The corresponding rules are the following:

IN_idle

if *cmd is* IN $\wedge$ $content(Breg) = nil$
then $content(Breg) := Wreg$
$place(Wreg) := Areg$
$com\_mode(Wreg) := input$
dequeue

IN_ready

if *cmd is* IN $\wedge$ $content(Breg) \neq nil$
then $content(Areg)) := mssg(content(Breg))$
$content(Breg) := nil$
enqueue $content(Breg)$

Clearly the rules IN_idle and IN_ready directly implement the $OCCAM_{daemon}$ rules inp_idle(x, c, v) and inp_ready(x, c, v) for internal channels, with a being *Wreg*, $\bar{c}$ being *Breg* and $\bar{v}$ being *Areg* (loaded by LDLP) (see the input rules in section 3. and 4.3.). This proves the correctness lemma 6.2 for the compilation of input commands. [17]

Output

Also in the output case[18] we abstract from the memory organization (see footnote 17). Note however that we could have use a specialized OUTW instruction which explicitly uses a unit size.

$$\begin{array}{l} compile(c\,!\,t, e, m, x) = \\ \quad \texttt{LDLP}\ bind(c, e, x), \\ \quad \mathsf{eval}(t, e, x) \\ \quad \texttt{OUT} \end{array}$$

As in the input case, the following rules OUT_idle, OUT_ready and OUT_alt directly implement the $OCCAM_{daemon}$ rules out_idle(x, c, t), out_ready(x, c, t) and out_alt(x, c, t); this proves the correctness lemma

[17] Since at this level of abstraction we don't deal with the organization of memory into bytes, words, etc., we omit the compilation of the message size which is taken here as unit size.

[18] Note that when the compiler does also the compilation of expressions then the expression evaluation would first generate code for the expression evaluation and then will load the address of the channel (and then reorder the values in the registers).

6.2 for the compilation of output commands.

OUT_idle
if *cmd is* OUT
  $\wedge$ *content*(*Breg*) = *nil*
then *content*(*Breg*) := *Wreg*
  mssg(*Wreg*) := *Areg*
  dequeue

OUT_ready
if *cmd is* OUT
  $\wedge$ *content*(*Breg*) $\neq$ *nil*
  $\wedge$ *com_mode*(*content*(*Breg*)) = *input*
then *content*(*place*(*content*(*Breg*))) := *Areg*
  content(*Breg*) := *nil*
  enqueue content(*Breg*)

OUT_alt
if *cmd is* OUT
  $\wedge$ agent $\neq$ *nil*
  $\wedge$ *com_mode*(agent) $\neq$ *input*
then *dequeue*
  agent := *Wreg*
  mssg(*Wreg*) := *Areg*
  if *com_mode*(agent) = *alt_sleep*
  then *time-delete* agent
    com_mode(agent) := *alt_run*
    enqueue agent
where agent = *content*(*Breg*)

Conditionals

For IF and WHILE we compile the pseudo instruction if by using an unconditional jump instruction:

$$\begin{array}{l} compile(\mathsf{IF}\ B_1\,S_1\ \ldots\ B_r\,S_r, e, m, x) \ = \\ \quad \ldots \\ \quad compile(\mathsf{if}(B_i, l_i), e, m, x), \\ \quad \ldots \\ \quad compile(\mathsf{STOP}, e, m, x), \\ \quad \ldots \\ \quad l_j, \\ \quad compile(S_j, e, m, x), \\ \quad \mathtt{J}\ l_{Out}, \\ \quad \ldots \\ \quad l_{Out} \\ \textbf{where}\ 1 \leq i, j \leq r \end{array}$$

$$\begin{array}{l} compile(\mathsf{if}(B, L), e, m, x) \ = \\ \quad \mathsf{eval}(\neg B, e, x), \\ \quad \mathtt{CJ}\ L \end{array}$$

The negated boolean expression ($\neg B$) is evaluated because CJ performs the conditional branch if the value in *Areg* is *false*:

if *cmd is* CJ *L*
then if *Areg* = *false*
  then *Ireg* := *labeled_loc*(*L*)
  else *Areg* := *Breg*
    Breg := *Creg*
    Creg := *random*

$$\begin{array}{l} compile(\mathsf{WHILE}\ B\,S, e, m, x) \ = \\ \quad l_{In}, \\ \quad compile(\mathsf{if}(\neg B, l_{Out}), e, m, x), \\ \quad compile(S, e, m, x), \\ \quad \mathtt{J}\ l_{In}, \\ \quad l_{Out} \end{array}$$

Clearly, this compilation of IF and WHILE satisfies the correctness lemma 6.2 (see their compilation in section 4.2.).

Time–Slicing

The pseudo instruction goto is implemented by the Transputer instruction J and serves as *safe* place to perform time–slicing.

Statistical analysis of program behavior has taught us the well known locality rule for optimizations, saying that *the locations which will be used in the next execution step won't be far away from the data used in the last step.* The rule can be split into two rules, one for data and one for code. From this viewpoint branching instructions are expensive. They destroy the locality of code because the next instruction can be located very far away from the previous instruction location; implicitly they also destroy the locality of data because it is highly improbable that the context for code which is far away from the last instruction will have anything in common with the current context. From the point of view of locality branching is similar to rescheduling. Therefore branching places are a *natural* place for rescheduling of processes[19], i.e. to define function *safe* to be *true* only for the jump instruction.

if *cmd is* J *L* **then** *Ireg* := *labeled_loc*(*L*)

To prove the correctness of this implementation of time slicing it is sufficient to consider the following. The only instruction in Occam which can produce a loop is WHILE. The compilation of WHILE however contains a jump which will be executed at the end of each iteration. Therefore no compiled Occam program, when executed on the Transputer, can diverge without containing an infinite number of executions of jumps and thereby of time slicing.[20]

[19] The conditional jump differs from the unconditional jump because its execution destroys the locality only in one of the two possible cases. That is the reason why usually the locality preserving non jumping case is favored.

[20] Note that there are however simple **Transputer** programs which produce infinite runs without any unconditional jump. Here is an example, implementing an unconditional jump

For ALT-statements we have to compile the two pseudo-instructions alt_a and alt_s. The compilation of alt_a results in "enabling" code, the compilation of alt_s in code for "disabling" and the selection of one alternative.

ALT announce

The parallelism which is built into evolving algebras through the simultaneous execution of possibly many updates allowed us in $OCCAM_{daemon}$ to formulate alt_a as *one* computation step of the following simultaneous actions for the guards (see section 3.):

1. check whether among the communication guards there are some with true boolean condition; if yes, enable the corresponding channel;
2. compute the new minimal time requirement $t_{min}(a)$ appearing in time guards with true boolean condition;
3. check whether there is at least one SKIP command with true boolean condition;
4. depending on the result of 1 – 3 proceed to compute the communication mode (*com_mode*) for a or otherwise insert a into the time queue and send him to sleep.

The Transputer has to do these computations in some order. *compile* will produce for each communication guard the channel enabling instruction `ENBC` (preceded by instructions `LDLP` to load the channel $bind(c,e,x)$ and the evaluated boolean condition $\mathsf{eval}(b,e,x)$). Upon execution `ENBC` will also record into an auxiliary function:

$$alt\_ready \;:\; DAEMON \rightarrow \{true, false\}$$

whether there is already an outputting partner waiting for the communication. (An initialization instruction, namely `ALT` or `TALT`, will set this auxiliary function at the beginning to *false*.) For each time guard *compile* produces the time enabling instruction `ENBT` (preceded by instructions to evaluate the time argument $\mathsf{eval}(t,e,x)$ and the boolean condition $\mathsf{eval}(b,e,x)$). Upon execution `ENBT` records into an auxiliary function:

$$time\_enabled \;:\; DAEMON \rightarrow \{true, false\}$$

whether there is at least one enabled time guard and updates $t_{min}(a)$ if the encountered enabled time value is smaller then the previously encountered ones. (The appropriate initialization of *time_enabled*(a) to *false* is again done by the initialization instruction `TALT`.) For each SKIP guard *compile* produces an `ENBS` instruction (preceded by an instruction $\mathsf{eval}(b,e,x)$) which will set *alt_ready*(a) to *true* if *eval*(b,e,x) is *true*.

The compilation of alt_a terminates with an alt–waiting instruction `ALTWT` or `TALTWT` whose execution computes (through evaluation of *alt_ready*, t_{min} and *time_enabled*) whether the current daemon is to be sent to sleep (and into timer queue) or whether he can proceed to execute the code for alt_s.

This explains the following definitions for the compilation of alt_a and alt_s instructions[21] and proves the correctness lemma 6.2 for them. (A technical detail: in the initializing instructions `ALT` and `TALT` *com_mode*(a) is set to *alt_run* which will be changed to *alt_sleep* if `ALTWT` or `TALTWT` finds out that indeed the process has to go to sleep.) For an explanation of the update of the auxiliary function *alt_choice* in `ALTWT` and `TALTWT` see below.

$compile(\mathsf{alt\_a}(\vec{G}), e, m, x) =$
  `ALT`,
  ...
  `LDLP` $bind(c_i, e, x)$,
  $\mathsf{eval}(b_i, e, x)$,
  `ENBC`,
  ...
  $\mathsf{eval}(t_j, e, x)$,
  $\mathsf{eval}(b_j, e, x)$,
  `ENBT`,
  ...
  $\mathsf{eval}(b_k, e, x)$,
  `ENBS`,
  ...
  `ALTWT`
where
  $\vec{G} = G_1, \ldots, G_r$
  $G_i = b_i \;:\; c_i\,?\,v_i$
  $G_j = b_j \;:\; \mathsf{TIME}\,?\,\mathsf{AFTER}\, t_j$
  $G_k = b_k \;:\; \mathsf{SKIP}$
  $1 \le i \le p < j \le q < k \le r$

In case $l \neq p$, the same sequence of instructions but with `TALT` and `TALTWT` instead of `ALT` and `ALTWT` respectively is generated.

if *cmd is* `TALT` | `ALT`
then *time_enabled*(*Wreg*) := *false* |
  com_mode(*Wreg*) := *alt_run*
  alt_ready(*Wreg*) := *false*

if *cmd is* `ENBC`
then if *Areg* = *true*
  then if *content*(*Breg*) = *nil*
    then *content*(*Breg*) := *Wreg*
    elsif *content*(*Breg*) ≠ *Wreg*
    then *alt_ready*(*Wreg*) := *true*
  Breg := *Creg*
  Creg := *random*

Note that the Transputer scheme keeps *Areg* untouched by "enabling" instructions `ENBC`, `ENBT` and `ENBS` in order to offer the possibility to collect the boolean guard

by a conditional jump with guard which is constantly false: L, `LDC 0`, `CJ` L. Such programs do not appear as result of our compilation function.

[21] For the sake of definiteness we choose as order communication guards followed by time guards followed by SKIP guards. Since completeness cannot be preserved when a particular order is chosen, any order will do for the correctness claim. A suggestion of INMOS [Inmos:88] for making `ALT` appear non deterministic is to let the function *compile* reorder the alternatives randomly.

appearing there.

if *cmd is* ENBT
then if $Areg = true$
  then if $time\_enabled(Wreg) = false$
    then $time\_enabled(Wreg) := true$
      $t_{min}(Wreg) := Breg$
    elsif $Breg < t_{min}(Wreg)$
    then $t_{min}(Wreg) := Breg$
  $Breg := Creg$
  $Creg := random$

if *cmd is* ENBS
then if $Areg = true$
  then $alt\_ready(Wreg) := true$

if *cmd is* TALTWT|ALTWT
then if $t_{min}(Wreg) \geq timer \wedge |$
    $alt\_ready(Wreg) = false$
  then if $time\_enabled(Wreg)$
    then *time-insert*
    $com\_mode(Wreg) := alt\_sleep$
    dequeue
  $Areg, Breg, Creg := random$
  $alt\_choice(Wreg) := nil$

The selection of an alternative through alt_s is done in the Transputer by searching sequentially through all communication, time and skip guards for the first one which is ready. This deterministic Transputer strategy is responsible for the loss of completeness of $TRANSPUTER_{ground}$ with respect to $OCCAM_{daemon}$; only correctness can be preserved.

A realization of this deterministic Transputer strategy needs also a mechanism to prevent the processor from getting involved in a further ready alternative once the first one has been encountered. We formalize this by an auxiliary function:

$$alt\_choice \ : \ DAEMON \rightarrow LABEL \cup \{nil\}$$

which will hold (the label for) the first ready alternative and has to be initialized to *nil* **before** starting the execution of the alt_s code. This is why *alt_choice* is initialized in the rule for the last instruction issued by *compile* for alt_a (see above).

compile will produce for each guard an instruction which covers the disjunctive choice among readiness through communication, time or skip guards; these instructions are DISC, DIST, DISS and each of them will be preceded by instructions which load the corresponding channel or time value, the boolean guard and the label marking the code for the alternative. The last instruction generated for the compilation of alt_s, namely ALTEND, branches to the selected first (ready) alternative by setting *Ireg* to the location recorded in $alt\_choice(Wreg)$.

$\boxed{\text{ALT select}}$

This explains the following compilation [22] of alt_s and proves the correctness lemma 6.2 for ALT–statements.

$compile(\mathsf{alt\_s}(\vec{G}, \vec{l}), e, m, x) =$
  ...
  LDLP $bind(c_i, e, x)$,
  $\mathsf{eval}(b_i, e, x)$,
  LDC l_i,
  DISC,
  ...
  $\mathsf{eval}(t_j, e, x)$,
  $\mathsf{eval}(b_j, e, x)$,
  LDC l_j,
  DIST,
  ...
  $\mathsf{eval}(b_k, e, x)$,
  LDC l_k,
  DISS,
  ...
  ALTEND
where
  $\vec{l} = l_1, \ldots, l_r$
  $\vec{G} = G_1, \ldots, G_r$
  $G_i = b_i \ : \ c_i \, ? \, v_i$
  $G_j = b_j \ : \ \mathsf{TIME\,?\,AFTER}\, t_j$
  $G_k = b_k \ : \ b_k \ : \ \mathsf{SKIP}$
  $1 \leq i \leq p < j \leq q < k \leq r$

if *cmd is* DISC
then if *Breg*
  then if $content(Creg) = Wreg$
    then $content(Creg) := nil$
      $Areg := false$
    elsif $content(Creg) \neq nil$
      $\wedge\ alt\_choice(Wreg) = nil$
    then $alt\_choice(Wreg) := Areg$
      $Areg := true$
    else $Areg := false$
  else $Areg := false$
  $Breg, Creg := random$

if *cmd is* DIST
then if *Breg*
  then if $Creg < timer$
      $\wedge\ alt\_choice(Wreg) = nil$
    then $alt\_choice(Wreg) := Areg$
      $Areg := true$
    else $Areg := false$
  else $Areg := false$
  $Breg, Creg := random$

[22] To avoid repetition we skip the compilation of PRI ALT; it would come up to respect the order in which the guards appear in the source program.

if *cmd is* DISS
then if *Breg*
    then if $alt\_choice(Wreg) = nil$
        then $alt\_choice(Wreg) := Areg$
            $Areg := true$
        else $Areg := false$
    else $Areg := false$
    $Breg, Creg := random$

if *cmd is* ALTEND
then $Ireg := labeled\_loc(alt\_choice(Wreg))$

if *cmd is* LDC n **then** $push(n)$

Parallelism

For PAR statements we have to compile the two pseudo instructions par and end. The Transputer has to start the processes sequentially, for definiteness we stick here to the order in which they appear in the Occam PAR statement. The start instruction STARTP is preceded by instructions which load the location l_i and the (relative) base address $x_i - x$ (of x_i with respect to his father); upon execution of STARTP the run–time base address of the i^{th} son is calculated by adding the run–time base address *Wreg* of his father to the relative base address $x_i - x$ which at this moment is loaded into *Areg*. This explains the following definition:

$compile(\mathsf{par}(x, \vec{x}, \vec{l}, l_{Out}), e, m, x)$ =
    $count(Wreg) := r$,
    $end\_par(Wreg) := l_{Out}$,
    ...
    LDC l_i,
    LDLP $x_i - x$
    STARTP,
    ...
where $\vec{l} = l_1, \ldots, l_r$
    $\vec{x} = x_1, \ldots, x_r$
    $1 \leq i \leq r$

if *cmd is* STARTP
then $loc(Wreg + Areg) := labeled\_loc(Breg)$
    enqueue $Wreg + Areg$

The ENDP instruction terminates the process, causing rescheduling except when the process to be terminated is the last son. Note that ENDP is preceded by an LDLP instruction which loads into *Areg* the negative value of the distance between son and the father which has to be added to the current value of *Wreg*, namely $-(x_i - x)$ $= x - x_i$. This explains the following compilation of the end instruction.

$compile(\mathsf{end}(x_i), e, m, x)$ = LDLP $x - x_i$, ENDP

if *cmd is* ENDP
then if $count(x_f) > 1$
    then *dequeue*
        $count(x_f) := count(x_f) - 1$
    else $Ireg := labeled\_loc(end\_par(x_f))$
        $Wreg := x_f$
where $x_f = Wreg + Areg$

The arguments above show that this compilation of the instructions par and end correctly implements their abstract versions in $OCCAM_{daemon}$ (see section 4.5.) and thus prove the correctness lemma 6.2 for PAR statements. The proof of the theorem 6.1 follows by induction on S putting together the preceding local correctness lemmata.

6.2. Compilation of Workspace

In this section we implement our abstract functions. The definition of a daemon as process base address will be extended to daemons as workspace address. This term describes a memory block which is associated with a process. It is used to implement the environment together with all the functions which are needed by the process at run time. We will refine the function *compile* accordingly and show that the resulting evolving algebra $TRANSPUTER_{workspace}$ implements $TRANSPUTER_{ground}$ correctly.

6.2.1. Defining the abstract functions

The abstract functions which are used in our rules are here mapped into the memory by defining them relative to daemons. It would be a routine exercise to prove the correctness for a scheme which simply reserves for each of the eleven abstract functions and each daemon x some space, say $content(workspace(x, i))$, which is kept separate from the space for the environment of x. More sophistication is needed for the Transputer layout with optimized use of memory characterized by re-use of space for different non conflicting purposes. In accordance with the organization of environments as falling stack — where local variables of process x are addressed as positive offsets from $x = Wreg$ — locations with negative offsets from x are used for recording the values $f_i(x)$ of the abstract functions. The values which are likely to be used most often are put closest to x, namely $loc(x)$ and $next(x)$ which are needed when x is in the queue:

$$next(x) = content(x - 2) \qquad loc(x) = content(x - 1)$$

Since every daemon x can execute at any time only one Occam statement, functions which belong to different statements can be mapped to the same location without creating a conflict among them. This is the case for $mssg(x)$, $end\_par(x)$ and $alt\_choice(x)$ whose values are stored at $Wreg = x$:

$$mssg(x), end\_par(x), alt\_choice(x) = content(x)$$

Note that as consequence of this decision in [Inmos:88] to use the location *Wreg* as an extra 'register" for certain instructions one has to take care to avoid any conflicting use of this location for storing the first local variable of the daemon $x = Wreg$. This will be done by allocating in these cases an extra workspace slot for the use of *Wreg* to record one of $mssg(x)$, *end_par*, $alt\_choice(x)$; see below the introduction of *Wreg* adjustment instructions `AJW` into the compilation of output, ALT and PAR statements.

The three functions *place*, *alt_ready*, *com_mode* can be encoded in a consistent way by one function whose value is mapped to location $content(x-3)$. To avoid conflicts we introduce one new value *alt_ready* by use of which we can restore the three functions as follows:

$$alt\_ready(x) = \begin{cases} false & \text{if } content(\text{x-3}) \in \{\ alt\_sleep,\ alt\_run\} \\ true & \text{if } content(\text{x-3}) = alt\_ready \\ undef & \text{otherwise} \end{cases}$$

$$com\_mode(x) = \begin{cases} alt\_sleep & \text{if } content(\text{x-3}) = alt\_sleep \\ alt\_run & \text{if } content(\text{x-3}) \in \{\ alt\_run,\ alt\_ready\ \} \\ input & \text{otherwise} \end{cases}$$

$$place(x) = \begin{cases} undef & \text{if } content(\text{x-3}) \in \\ & \quad \{\ alt\_sleep,\ alt\_run, alt\_ready\ \} \\ content(x-3) & \text{otherwise} \end{cases}$$

For processes x executing the code for an ALT statement with time guards the locations $x-4$, $x-5$ are used to encode the relevant values of the functions *time_enabled* and t_{min}:

$$\begin{aligned} time\_enabled(x) &= content(x-4) \\ t_{min}(x) &= content(x-5) \end{aligned}$$

Note that the compilation schema defined below optimizes the use of memory by distinguishing ALT statements with and without time guards. For a PAR executing process x location $x+1$ is used to encode the son counter:

$$count(x) = content(x+1)$$

This use of location $x+1$ for workspace makes a corresponding workspace adjustment instruction necessary (see below definition of *env_size* and the refinement of *compile*). Note that we still keep the function *labeled_loc* here; it will disappear in section 6.3.1..

The layout of these functions into consecutive positions $x+1, \ldots, x-5$ realizes an optimized use of memory: each time a daemon is in the queue, locations $x-1$ and $x-2$ are used. When a daemon executes input/output for variables, also $x-3$ is needed, for values also x is used. When a daemon executes an ALT statement, locations $x, \ldots, x-3$ are used. In addition also $x-4, x-5$ are used if ALT contains a time guard. For a daemon executing a PAR statement locations $x+1, x+2$ are used. It is easy to formalize these observations for a proof of the following interesting

LEMMA 6.3. (WORKSPACE OPTIMIZATION LEMMA) *Assuming a proper initialization it never happens in runs of* $TRANSPUTER_{workspace}$ *that there are any holes between used locations.*

In order to incorporate this implementation of functions by *content* correctly into the rules of $TRANSPUTER_{ground}$ we have to reserve for each daemon x enough space to encode his functions without touching the space of other daemons.

For the convenience of the reader we collected the final rules in appendix (see 7.2.).

6.2.2. Workspace Size

In this section we refine the function *env_size* (which calculates the space needed by a daemon for his environment) to the real Transputer workspace by coupling it with a function *spec_size* which takes into account the special space needed for the implementation of the abstract functions as defined above. We introduce two new functions:

$$\begin{aligned} spec\_size &: STATEMENT \rightarrow \mathbf{N} \\ work\_size &: STATEMENT \rightarrow \mathbf{N} \\ work\_size(S) &= env\_size(S) + spec\_size(S) \end{aligned}$$

spec_size calculates the space to be reserved for the abstract functions. *spec_size* for declarations is already covered by *env_size*, therefore:

$$\begin{aligned} spec\_size(\mathsf{VAR}\ id_1, \ldots, id_r : S) &= spec\_size(S) \\ spec\_size(\mathsf{CHAN}\ id_1, \ldots, id_r : S) &= spec\_size(S) \end{aligned}$$

For each statement it must be foreseen that the process is in the queue; this requires two locations:

$$\begin{aligned} spec\_size(\mathsf{SKIP}) &= 2 \\ spec\_size(\mathsf{STOP}) &= 2 \\ spec\_size(v := t) &= 2 \\ spec\_size(\mathsf{TIME}\,?\,v) &= 2 \end{aligned}$$

The input statements use in addition the functions *place* and *com_mode*, both implemented by $content(x-3)$:

$$spec\_size(c\,?\,v) = 3$$

Output statements require the function *mssg* and *com_mode*. As we want to output the value of a variable without actually having the variable, a temporary variable is allocated using the function *mssg* and its address is stored in *place*. Since *com_mode* and *place* are mapped to the same location we define:

$$spec\_size(c\,!\,t) = 4$$

Due to the sequential execution of component statements of SEQ, IF and WHILE, *spec_size* is defined there as maximum of the *spec_size* of components.

$$spec\_size(\mathsf{SEQ}\ S_1 \dots S_r) = max_{i=1}^r\ spec\_size(S_i)$$

$$spec\_size(\mathsf{IF}\ B_1\, S_1, \dots B_r\, S_r) = max_{i=1}^r\ (spec\_size(S_i))$$

$$spec\_size(\mathsf{WHILE}\ B\, S) = spec\_size(S)$$

In ALT statements we distinguish two cases: case (a) without timer guard(s): the functions *alt_ready*, *alt_choice* and *com_mode* are needed; case (b) with timer guard(s), where also *time_enabled* and t_{min} are needed. Therefore:

$$spec\_size(\mathsf{ALT}\ G_1\, S_1 \dots G_r\, S_r) = \begin{cases} max_{i=1}^r\ (4, spec\_size(S_i)) & \text{case (a)} \\ max_{i=1}^r\ (6, spec\_size(S_i)) & \text{case (b)} \end{cases}$$

For PAR statements we need the additional functions *count* and *end_par*. Because of the father doing his first son's job we must check the first component statement:

$$spec\_size(\mathsf{PAR}\ S_1 \dots S_r) = max(4, spec\_size(S_1))$$

We must extend also *env_size* because each son (except the first who is covered by his father) has to receive enough space not only for his environment but also for his functions:

$$env\_size(\mathsf{PAR}\, S_1 \dots S_r) = env\_size(S_1) + \sum_{i=2}^{r} work\_size(S_i)$$

6.2.3. *Workspace Adjustment*

In this section we refine the function *compile* with respect to the implementation of the abstract Occam functions by the special workspace assigned to daemons. This refinement has to prevent conflicting requests for already used locations x and $x+1$ in the case of daemon x executing statements which make use of Occam functions encoded there. (We will see that in general what is and has to be prevented are conflicting requests for the use of already used locations.) The three cases are output, ALT, and PAR statements.

In the case of a daemon x executing an output statement we must prevent that $content(x) = mssg(x)$ overwrites a local variable stored in $content(x)$. Function *compile* takes care of this by generating an instruction which provides an extra location for the encoding of $mssg(x)$: the instruction `AJW` adjusts the workspace address by n, in this case by minus one, making temporarily location $x-1$ location x and restoring the original workspace address once the output is done:

$$\begin{aligned} & compile(c\,!\,t, e, m, x) = \\ & \quad \texttt{LDLP}\ bind(c, e, x), \\ & \quad \mathsf{eval}(t, e, x) \\ & \quad \texttt{AJW}\ -1 \\ & \quad \texttt{OUT} \\ & \quad \texttt{AJW}\ +1 \end{aligned}$$

The rule which executes the new instruction is:

$$\textbf{if}\ cmd\ is\ \texttt{AJW}\ n\ \textbf{then}\ Wreg := Wreg + n$$

Note that the definition of *spec_size* for output statements provides the three locations which are needed to make this workspace sliding conflict free. This proves the correctness lemma for the implementation of output statements in $TRANSPUTER_{workspace}$.

This proves the following lemma:

LEMMA 6.4. (CORRECTNESS LEMMA) *For output statements S of* Occam*, the execution of the rules of $TRANSPUTER_{ground}$ on $compile_1(S, e, m, x)$ is correctly implemented by the execution of the rules of $TRANSPUTER_{workspace}$ on $compile_2(S, e, m, x)$ where $compile_1$, $compile_2$ denote the function compile of $TRANSPUTER_{ground}$ and $TRANSPUTER_{workspace}$ respectively.*

A daemon x who executes an ALT statement will make use of $alt\_choice(x)$, stored as $content(x)$. We can proceed as in the case of output statements to prevent the use of location x as first position for the environment part which contains the local variables. Since any alternative might be chosen by executing ALT, *compile* emits a workspace adjusting instruction at the entry of each branch. Note that in accordance with the initial workspace adjustment by -1 the compilation of alt_a and alt_s receives the workspace pointer parameter $x-1$ to ensure the correct access to the environment part.

Note that the definition of *env_size* for ALT statements makes sure that the space which is necessary for a conflict free workspace adjustment has been provided.

Since we don't know in advance which branch will be chosen by ALT, we have to generate the re-sliding instruction in each branch:

$$\begin{aligned} & compile(\mathsf{ALT}\ G_1\, S_1 \dots G_r\, S_r, e, m, x) = \\ & \quad \texttt{AJW}\ -1, \\ & \quad compile(\mathsf{alt\_a}(\vec{G}), e, m, x-1), \\ & \quad compile(\mathsf{alt\_s}(\vec{G}, \vec{l}), e, m, x-1), \\ & \quad \dots \\ & \quad l_i, \\ & \quad \texttt{AJW}\ +1, \\ & \quad compile(c_i\, ?\, v_i, e, m, x), \\ & \quad compile(S_i, e, m, x), \\ & \quad \texttt{J}\ l_{Out}, \\ & \quad \dots \\ & \quad l_j, \\ & \quad \texttt{AJW}\ +1, \\ & \quad compile(S_j, e, m, x), \\ & \quad \texttt{J}\ l_{Out}, \\ & \quad \dots \\ & \quad l_{Out} \\ & \textbf{where}\ \vec{l} = l_1, \dots, l_r \\ & \qquad \vec{G} = G_1, \dots, G_r \\ & \qquad G_i = b_i\ :\ c_i\, ?\, v_i \\ & \qquad G_j = b_j\ :\ (\ \mathsf{TIME}\,?\,\mathsf{AFTER}\, t_j\ \text{or}\ \mathsf{SKIP}\ ) \\ & \qquad 1 \le i \le p < j \le r \end{aligned}$$

We can now prove the correctness lemma 6.4 for ALT statements. Note that at the moment when `ALTEND` is executed the slided *Wreg* points to the correct location where the reference to the chosen alternative is stored.

For a daemon x executing a PAR statement we have to consider two possible sources for conflicting reuse of memory. One is similar to the situation we know already from output and ALT statements: location $Wreg = x$ is used to store $end\_par(x)$ and location $x+1$ for $count(x)$. Conflicting use of these two locations for local variables is avoided by a workspace adjustment through the instruction $\texttt{AJW} - 2$. The second conflict to prevent is a conflict between x and his sons. First of all each son has to receive enough space for his own values of the relevant abstract Occam functions; this is provided by the refinement of m_i in the refinement of *compile* which includes the special workspace (see figure 11). But note that also the workspace conflict between x acting as father and x acting as his first son has to be prevented in case that his son is again responsible for a PAR statement. Here is an example:

```
PAR
   PAR
      p
      q
      r
L:  s
L':
```

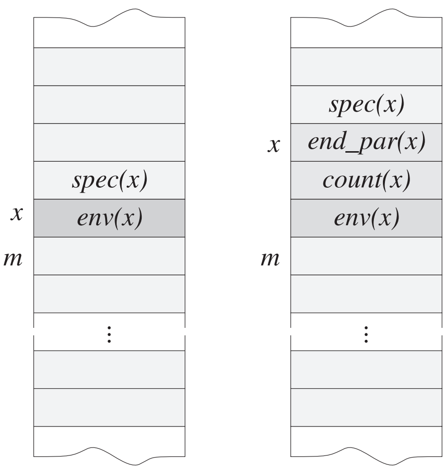


FIGURE 10. Sliding of the father

The overwriting of the values $content(x) = \texttt{L}$[23] and $content(x+1) = 3$ belonging to the father by the values $end\_par(x_1) = \texttt{L}'$ and $count(x_1) = 2$ belonging to the first son $x_1 = x$ is avoided by the initial workspace pointer adjustment $\texttt{AJW} - 2$ (see figure 10). Note that here as for ALT statements the adjusted value $x-2$ becomes the workspace pointer passed as parameter for the compilation of par and end.

$$
\begin{aligned}
&compile((\mathsf{PAR}\ S_1 \ldots S_r), e, m, x) \ = \\
&\quad \texttt{AJW}\ -2 \\
&\quad compile(\mathsf{par}(x, \vec{m}, \vec{l}, l_{Out}), e, m, x-2), \\
&\quad compile(S_1, e, m_1, x-2), \\
&\quad compile(\mathsf{end}(x-2), e, m, x-2), \\
&\quad l_2, \\
&\quad compile(S_2, e, m_2, m_2), \\
&\quad compile(\mathsf{end}(m_2), e, m_2, x-2), \\
&\quad \ldots \\
&\quad l_r, \\
&\quad compile(S_r, e, m_r, m_r), \\
&\quad compile(\mathsf{end}(m_r), e, m_r, x-2), \\
&\quad l_{Out}, \texttt{AJW}\ +2 \\
&\textbf{where}\ \vec{l} = l_2, \ldots, l_r \\
&\qquad \vec{m} = m_2, \ldots, m_r \\
&\qquad m_1 = m \\
&\qquad m_i = \ m_{i-1} + env\_size(S_{i-1}) + \\
&\qquad\qquad spec\_size(S_i) \\
&\qquad 2 \le i \le r
\end{aligned}
$$

Together with the implementation of the abstract functions *count* and *end_par* the first two abstract assignment instructions in the definition of $compile(\mathsf{par}(x, \vec{x}, \vec{l}, l_{Out}), e, m, x)$ in $TRANSPUTER_{ground}$ are replaced by:

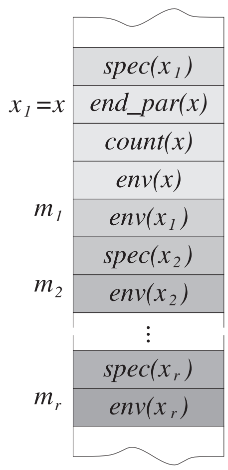


FIGURE 11. Sliding of sons

$\texttt{LDC}\ r$,
$\texttt{STL}\ 1$,
$\texttt{LDC}\ l_{Out}$
$\texttt{STL}\ 0$

As we have seen each process upon its creation must be (initially) slided into the plus direction pointing to the base address of the environment as shown in figure 12. For the very first process — the daemon Demiurge — we must do this manually, therefore the initial state of

[23] Note that the labels `L` and `L'` we have inserted here are not part of the Occam program but will be generated by *compile* to mark the exit of a PAR statement.

the loading algebra is refined by:

$$\begin{array}{rcl} load\_prg & = & \mathtt{AJW}\ + spec\_size(S) \\ & & compile(S, [\,], 0, 0) \\ & & \mathtt{AJW}\ - spec\_size(S) \end{array}$$

The preceding reasoning proves the correctness lemma

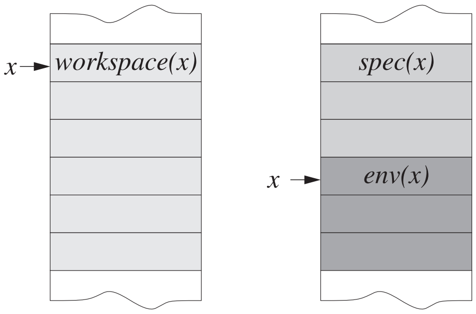


FIGURE 12. Initial sliding

6.4 for PAR statements.

THEOREM 6.5. *The implementation of* $TRANSPUTER_{ground}$ *in* $TRANSPUTER_{workspace}$ *is correct. Corresponding runs of the two evolving algebras on arbitrary* Occam *programs* S *are equivalent.*

Proof. By induction on S. The cases where there is no conflict among the abstract functions are covered by *compile* providing enough workspace $work\_size(S)$. The non trivial case where a conflict might have occurred are covered by the above correctness lemmata for output, ALT and PAR statements. ∎

6.3. Relocatable code

The real Transputer code uses relative branching instructions and doesn't have labels. In this section we eliminate labels from the compiled code (and thereby the function *labeled_loc* from our rules) in two steps: first we refine the function *compile* to incorporate relative addressing of instructions, then we add a label resolution phase to the loading procedure which is based upon location distances.

It is easy to apply the technique of relative addressing to instructions. What matters for the execution of a branching instruction is to know its distance from the place where the target instruction resides. Here is the refinement for the compilation of WHILE commands and the related jump rule:

$$\begin{array}{l} compile(\mathsf{WHILE}\ B\ S, e, m, x)\ = \\ \quad l_{In}, \\ \quad compile(\mathsf{if}(\neg B, l_{Out}), e, m, x), \\ \quad compile(S, e, m, x), \\ \quad \mathtt{J}\ l_{In} - l_{Out}, \\ \quad l_{Out} \end{array}$$

if *cmd is* $\mathtt{J}\ l_1 - l_2$
then $Ireg := Ireg + (labeled\_loc(l_1) - labeled\_loc(l_2))$

Clearly the correctness lemma holds for this refinement of WHILE statements.

For alt_s we have a similar refinement of *compile* and of the related ALTEND rule (where again the auxiliary function *labeled_loc* disappears). Note that the jumping action takes place in ALTEND, therefore from l_{here} (forward) to the chosen alternative with label l_i.

$$\begin{array}{l} compile(\mathsf{alt\_s}(\vec{G}, \vec{l}), e, m, x)\ = \\ \quad \ldots \\ \quad \mathtt{LDLP}\ bind(c_i, e, x), \\ \quad \mathsf{eval}(b_i, e, x), \\ \quad \mathtt{LDC}\ l_i - l_{here}, \\ \quad \mathtt{DISC}, \\ \quad \ldots \\ \quad \mathsf{eval}(t_j, e, x), \\ \quad \mathsf{eval}(b_j, e, x), \\ \quad \mathtt{LDC}\ l_j - l_{here}, \\ \quad \mathtt{DIST}, \\ \quad \ldots \\ \quad \mathsf{eval}(b_k, e, x), \\ \quad \mathtt{LDC}\ l_k - l_{here}, \\ \quad \mathtt{DISS}, \\ \quad \ldots \\ \quad \mathtt{ALTEND}, \\ \quad l_{here} \\ \textbf{where}\ \vec{l} = l_1, \ldots, l_r \\ \qquad \vec{G} = G_1, \ldots, G_r \\ \qquad G_i = b_i\ :\ c_i\,?\,v_i \\ \qquad G_j = b_j\ :\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j \\ \qquad G_k = b_k\ :\ \mathsf{SKIP} \\ \qquad 1 \le i \le p < j \le q < k \le r \end{array}$$

if *cmd is* ALTEND
then $Ireg := Ireg + alt\_choice(Wreg)$

The same technique applies to the refinement of *compile* on IF statements:

$$\begin{array}{l} compile(\mathsf{IF}\ B_1\, S_1\ \ldots\ B_r\, S_r, e, m, x)\ = \\ \quad \ldots \\ \quad compile(\mathsf{if}(B_i, l_i), e, m, x), \\ \quad \ldots \\ \quad compile(\mathsf{STOP}, e, m, x), \\ \quad \ldots \\ \quad l_i, \\ \quad compile(S_i, e, m, x), \\ \quad \mathtt{J}\ l_{Out} - l_{i+1}, \\ \quad \ldots \\ \quad l_{Out} \\ \textbf{where}\ l_{r+1} = l_{Out} \\ \qquad 1 \le i \le r \end{array}$$

For the corresponding environment of *compile* on $\mathsf{if}(B, L)$ we have no name l_{Out} and therefore create a new label l_{here} to mark the place where the instruction pointer *Ireg* will point to when the Transputer is executing the branching instruction (see *fetch* phase in 5.4.). Thus we have the following refinement of *compile* on $\mathsf{if}(B, L)$ and of the rule for conditional jump (note that the auxiliary function *labeled_loc* disappears from the rule):

$$\begin{aligned} &compile(\mathsf{if}(B, L), e, m, x) \ = \\ &\quad \mathsf{eval}(\neg B, e, x), \\ &\quad \mathtt{CJ}\ L - l_{here}, \\ &\quad l_{here} \end{aligned}$$

if *cmd is* `CJ` $l_1 - l_2$
then if $Areg = false$
  then $Ireg := Ireg +$
      $(labeled\_loc(l_1) - labeled\_loc(l_2))$
  else $Areg := Breg$
    $Breg := Creg$
    $Creg := random$

Again it is obvious that this refinement satisfies the correctness lemma for IF statements. For the compilation of PAR we can proceed in a similar way replacing the starting address l_i of the i^{th} son by the distance $l_i - l_i'$ to the location l_i' of the instruction `STARTP` which actually starts the process. The rule for `STARTP` is refined correspondingly by adding the relevant distance to the current value of *Ireg*; here too *labeled_loc* disappears.

There is a slight additional complication. For the execution of the jump in the instruction `ENDP` the absolute (return) address is needed. It can be computed upon entering the PAR code when the distance $l_{Out} - l_{here}$ is loaded into *Areg* to get stored into $content(x)$: it suffices to add to this compile–time value the current value of *Ireg*. This is done by the new instruction `LDPI` which is inserted after `LDC` $l_{Out} - l_{here}$.

$$\begin{aligned} &compile(\mathsf{par}(x, \vec{x}, \vec{l}, l_{Out}), e, m, x) \ = \\ &\quad \mathtt{LDC}\ r, \\ &\quad \mathtt{STL}\ 1, \\ &\quad \mathtt{LDC}\ l_{Out} - l_{here} \\ &\quad \mathtt{LDPI}, \\ &\quad l_{here}, \\ &\quad \mathtt{STL}\ 0, \\ &\quad \ldots \\ &\quad \mathtt{LDC}\ l_i - l_i', \\ &\quad \mathtt{LDLP}\ x_i - x \\ &\quad \mathtt{STARTP}, \\ &\quad l_i', \\ &\quad \ldots \\ &\textbf{where}\ \vec{l} = l_1, \ldots, l_r \\ &\qquad\quad \vec{x} = x_1, \ldots, x_r \\ &\qquad\quad 1 \leq i \leq r \end{aligned}$$

if *cmd is* `LDPI`
then $Areg := Ireg + Areg$

if *cmd is* `STARTP`
then $loc(Wreg + Areg) := Breg + Ireg$
  enqueue $Wreg + Areg$

Let $TRANSPUTER_{rel-code}$ be the refinement of $TRANSPUTER_{workspace}$ by introducing relative addressing of instructions for relocatable code as defined in this section. The preceding arguments prove the following:

LEMMA 6.6. *$TRANSPUTER_{rel-code}$ is a correct implementation of $TRANSPUTER_{workspace}$.*

6.3.1. Resolving Labels

The rules of $TRANSPUTER_{rel-code}$ show that at run–time only the distances $labeled\_loc(l_1) - labeled\_loc(l_2)$ between labeled locations are used but not any more the labels l_1, l_2 themselves. We therefore eliminate in this section the appearance of labels from the code which is loaded into the Transputer memory.

The idea is to transfer the computation of the distances between labeled locations from the execution time of Transputer instructions to the very moment where the values $labeled\_loc(l_i)$ become known. Before loading we therefore pre-process the compiled program $compile(S, [\,], 0, 0)$ by a variant *LOAD-LABEL* of the *LOAD* algebra, whose function is to compute the values of the function *labeled_loc* — followed by the evaluation of the distances:

$$labeled\_loc(l_1) - labeled\_loc(l_2).$$

Let *LOAD-LABEL* be obtained from the algebra *LOAD* (see section 4.2.) by adding the guard $mode =$ *load-label* to the rules and by replacing the update:

$$cmd(labeled\_loc) := c$$

by the update:

$$tmp\_prg := append(tmp\_prg, c)$$

where $tmp\_prg : (CODE \cup LABEL)^*$ is a new 0-ary function. *LOAD-LABEL* algebra is initialized by:

$$\begin{aligned} load\_prg &= \mathtt{AJW}\ spec\_size(S) \\ & compile(S, [\,], 0, 0) \\ & \mathtt{AJW}\ -spec\_size(S) \\ tmp\_prg &= [\,] \\ loading\_loc &= 0 \end{aligned}$$

LOAD-LABEL computes $labeled\_loc(l)$ for each label l appearing in *load_prg* and stores a copy of *load_prg* in *tmp_prg*. Upon termination of *LOAD-LABEL* the following rule switches to the computation of distances between labeled locations:

if $mode = load\text{-}label \wedge load\_prg = [\,]$
then $loading\_loc := 0$
  $mode := compute\text{-}distances$

The following rule replaces $l_1 - l_2$ by $labeled\_loc(l_1) - labeled\_loc(l_2)$ in the code produced by *compile*:

if $mode = compute\text{-}distances \wedge tmp\_prg \neq [\,]$
then $loading\_loc := loading\_loc +$
  $tmp\_prg := tail(tmp\_prg)$
  if $first(tmp\_prg) = c\ l_1 - l_2$
  then $load\_prg := append(load\_prg,$
      $c\ labeled\_loc(l_1) - labeled\_loc(l_2))$
  else $load\_prg := append(load\_prg, first(tmp\_prg))$

When this rule cannot be applied any more, the following rules switches to the algebra obtained from $LOAD$ by adding the condition $mode = load$ to the guard and by deleting the rule load_label:

$$\textbf{if } mode = compute\text{-}distances \land tmp\_prg = [\,]$$
$$\textbf{then } mode := load$$

Where the loading rule cannot be applied any more, the following rule switches to the rules of $TRANSPUTER_{rel-code}$:

$$\textbf{if } mode = load \land load\_prg = [\,]$$
$$\textbf{then } mode := fetch$$

Let $TRANSPUTER$ be the evolving algebra consisting of all the rules described above added to the rules of $TRANSPUTER_{rel-code}$. It is an exercise to prove the following lemma:

LEMMA 6.7. *$TRANSPUTER$ implements $TRANSPUTER_{rel-code}$ correctly.*

The proof of the theorem 6.1 follows by induction on S putting together the single local correctness lemmata stated and proved above. The 4.10 theorem, the 6.1 theorem and the 6.5 theorem prove the Main theorem.

CONCLUSION AND OUTLOOK

We have built a formal model for the Transputer instruction set architecture and have used it to formulate and prove a correctness theorem for a general compilation schema of Occam programs into Transputer code. Occam and Transputer here served as non trivial paradigm taken from the real world and independent from the compiler verification project. The method developed in this paper to support correct compiler design is general and can be applied *mutatis mutandis* to other architectures and other programming languages which exhibit the characteristic features of distributed computing, namely nondeterminism and concurrency.

There are several directions for future work which we consider challenging and worthwhile to be investigated. One is to refine the Transputer actions to sequences of hardware actions which are executed by a hardware interpreter; other processors might also be interesting for this line of research. Another direction is to figure out whether our proofs can be carried out in interesting deductive frameworks. In this context it would be especially interesting to compare our approach with the work done on the subject within the PROCOS II Esprit Basic Research Action. A directly related investigation is Müller–Olm's forthcoming doctoral dissertation (see [MMO:95] where a code generator correctness proof for a sequential sublanguage of Occam is given by delivering increasingly more abstract levels starting from the Transputer). We guess that some of the evolving algebras developed here are models of the algebraic laws employed for Müller–Olm's proofs. In general it could be interesting to investigate the relation between evolving algebras and Hoare's refinement algebra approach to prove correctness of compiling specifications (see [Hoare:91], [Hoare et al.:87], [Hoare et al.:93]). A challenging project is to verify our proofs by theorem provers. Note that the corresponding computer verification of the WAM correctness proof of [BöRo:95b] in *KIV* and *ISABELLE* at present looks promising.

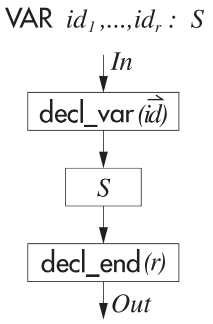
Acknowledgments

We thank the following colleagues for criticism and useful comments on previous versions of this paper: Peter Baumann, Markus Mueller-Olm, Kirsten Winter, Wolf Zimmermann and two anonymous referees. We also thank Schloß Dagstuhl for many weekends we have been offered there to work on this paper.

7. APPENDIX

7.1. Generating Flowcharts (see section 4.1)

Given two nodes, $Begin$ and End, such that $next(Begin) = End$, the following rules will generate the flowchart for an Occam program $S = cmd(Begin)$ (assuming that there are no further nodes or function values, and that S belongs to the fragment of Occam treated in the main text).



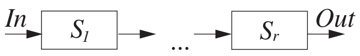
$$\begin{array}{l}
\textbf{if } cmd(n) = \mathsf{VAR}\ id_1, \ldots, id_r\ :\ S \\
\textbf{then extend } NODE \textbf{ by } n_1, n_2 \textbf{ with} \\
\quad cmd(n) := \mathsf{decl_var}(\vec{id}) \\
\quad next(n) := n_1 \\
\quad cmd(n_1) := S \\
\quad next(n_1) := n_2 \\
\quad cmd(n_2) := \mathsf{decl_end}(r) \\
\quad next(n_2) := next(n) \\
\textbf{endextend} \\
\textbf{where } \vec{id} = id_1, \ldots, id_r
\end{array}$$

if $cmd(n)$ = CHAN $id_1, \ldots, id_r$: S
then extend $NODE$ **by** n_1, n_2 **with**

$cmd(n)$:= decl_chan($\vec{id}$)
$next(n)$:= n_1
$cmd(n_1)$:= S
$next(n_1)$:= n_2
$cmd(n_2)$:= decl_end(r)
$next(n_2)$:= $next(n)$

endextend
where $\vec{id} = id_1, \ldots, id_r$

SEQ $S_1 \ldots S_r$

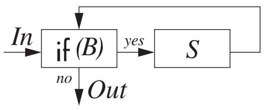


if $cmd(n)$ = SEQ $S_1 \ldots S_r$
then extend $NODE$ **by** $n_2, \ldots, n_r$ **with**

...
$cmd(n_i)$:= S_i
$next(n_i)$:= n_{i+1}
...

endextend
where $n_1 = n$
$n_{r+1} = next(n)$
$1 \leq i \leq r$

WHILE B S

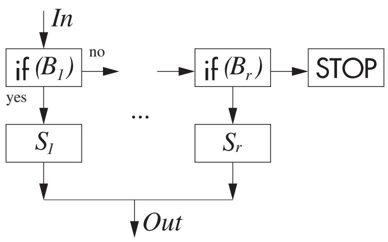


if $cmd(n)$ = WHILE B S
then extend $NODE$ **by** n_1 **with**

$cmd(n)$:= if(B)
$no(n)$:= $next(n)$
$yes(n)$:= n_1
$cmd(n_1)$:= S
$next(n_1)$:= n

endextend

IF B_1 $S_1 \ldots B_r$ S_r

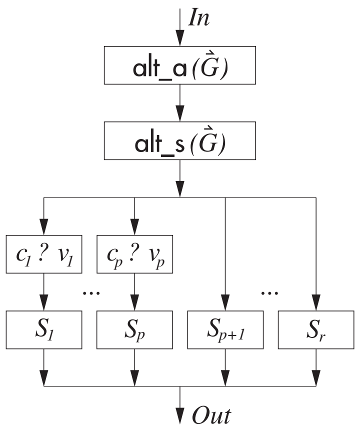


if $cmd(n)$ = IF B_1 $S_1 \ldots B_r$ S_r
then extend $NODE$ **by** $n_1, \ldots, n_r,$ $m_2, \ldots, m_{r+1}$ **with**

...
$cmd(m_i)$:= if(B_i)
$yes(m_i)$:= n_i
$no(m_i)$:= m_{i+1}
$cmd(n_i)$:= S_i
$next(n_i)$:= $next(n)$
...
$cmd(m_{r+1})$:= STOP

endextend
where $m_1 = n$
$1 \leq i \leq r$

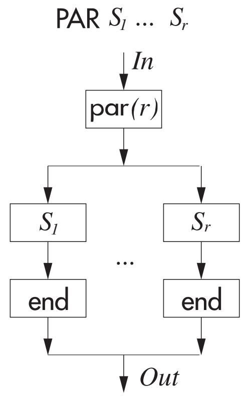
ALT G_1 $S_1 \ldots G_r$ S_r



if $cmd(n)$ = ALT G_1 $S_1 \ldots G_r$ S_r
then extend $NODE$ **by** $n_0, \ldots, n_r,$ $m_1, \ldots, m_p$ **with**

$cmd(n)$:= alt_a($\vec{G}$)
$next(n)$:= n_0
$cmd(n_0)$:= alt_s($\vec{G}$)
...
$next(n_0, i)$:= n_i
$cmd(n_i)$:= c_i ? v_i
$next(n_i)$:= m_i
$cmd(m_i)$:= S_i
$next(m_i)$:= $next(n)$
...
$next(n_0, j)$:= n_j
$cmd(n_j)$:= S_j
$next(n_j)$:= $next(n)$
...

endextend
where $\vec{G} = G_1, \ldots, G_r$
$G_i = b_i$: c_i ? v_i
$G_j = b_j$: (TIME ? AFTER t_j or SKIP)
$1 \leq i \leq p < j \leq r$



if $cmd(n) = \mathsf{PAR}\ S_1 \ldots S_r$
then extend $NODE$ **by** $n_1, m_1 \ldots, n_r, m_r$ **with**
    $cmd(n) := \mathsf{par}(r)$
    $\ldots$
    $next(n, i) := n_i$
    $cmd(n_i) := S_i$
    $next(n_i) := m_i$
    $cmd(m_i) := \mathsf{end}$
    $\ldots$
  endextend
where $1 \leq i \leq r$

7.2. Summary of $OCCAM_{daemon}$ (section 4.6)

The Processor

$$content : LOC \rightarrow VAL$$
$$+1 : LOC \rightarrow LOC$$

$$\begin{aligned} a &: DAEMON \cup \{nil\} \\ QUEUE &\subseteq DAEMON^* \\ first, last &: QUEUE \rightarrow DAEMON \\ rest &: QUEUE \rightarrow QUEUE \\ q &: QUEUE \end{aligned}$$

$enqueue\ x \stackrel{\text{def}}{=} q := q.x$

$dequeue \stackrel{\text{def}}{=}$ **if** not $empty\ q$
  then $a := first(q)$
    $q := rest(q)$
    $start := timer$
  else $a := nil$

$$\begin{aligned} timer &: \mathbf{N} \\ start &: \mathbf{N} \\ period &: \mathbf{N} \\ elapsed &\stackrel{\text{def}}{=} timer - start > period \end{aligned}$$

$\boxed{\text{time–slice}}$ **if** $a \neq nil$
  $\wedge\ elapsed$
 then seq $q := q.a$
   $dequeue$
  endseq

$$\begin{aligned} time\_q &: QUEUE \\ t_{min} &: DAEMON \rightarrow \mathbf{N} \end{aligned}$$

$time\text{-}insert \stackrel{\text{def}}{=}$
  let $time\_q = x_1. \ldots .x_i.x_{i+1}. \ldots .x_r$
  let $t_{min}(x_i) \leq t_{min}(a) \leq t_{min}(x_{i+1})$
  $time\_q := x_1. \ldots .x_i.a.x_{i+1}. \ldots .x_r$

$time\text{-}delete\ x \stackrel{\text{def}}{=}$
  let $time\_q = x_1. \ldots .x_i.x.x_{i+1}. \ldots .x_r$
  $time\_q := x_1. \ldots .x_i.x_{i+1}. \ldots .x_r$

$$\begin{aligned} proceed &\stackrel{\text{def}}{=} loc(a) := loc(a) + 1 \\ sleep\ at\ n &\stackrel{\text{def}}{=} loc(a) := n, dequeue \\ wakeup\ x &\stackrel{\text{def}}{=} enqueue\ x \\ cmd\ is\ c &\stackrel{\text{def}}{=} content(loc(a)) = c \wedge \text{not } elapsed \end{aligned}$$

Compilation & Loading

$$\begin{aligned} compile &: STATEMENT \times ENV \times LOC \times DAEMON \\ &\rightarrow ((CODE \times ENV \times DAEMON) \cup LABEL)^* \end{aligned}$$

$$\begin{aligned} loading\_loc &: NODE \\ load\_prg &: ((CODE \times ENV \times DAEMON) \cup LABEL)^* \end{aligned}$$

$\boxed{\text{load\_label(l,t)}}$
if $load\_prg = [\,l\,|\,t\,] \wedge l \in LABEL$
then $labeled\_loc(l) := loading\_loc$
  $load\_prg := t$

$\boxed{\text{load\_cmd(c,t)}}$
if $load\_prg = [\,c\,|\,t\,] \wedge c \in CODE$
then $content(loading\_loc) := c$
  $load\_prg := t$
  $loading\_loc := loading\_loc+$

Declarations

$compile(\mathsf{VAR}\ id_1, \ldots, id_r : S, e, m, x) =$
  $compile(S, e', m + r, x)$
where $e' = append(e_{new}, e)$
  $e_{new} = (id_r, m + r - 1), \ldots, (id_1, m).$

$compile(\mathsf{CHAN}\ id_1, \ldots, id_r : S, e, m, x) =$
$\quad \mathsf{init\_chan}(\vec{id}, e', x),$
$\quad compile(S, e', m+r, x)$
where $e' = append(e_{new}, e)$
$\qquad e_{new} = (id_r, m+r-1), \ldots, (id_1, m).$

$$\begin{array}{rcl}
ENV & = & (ID \times (VAR \cup CHANNEL))^* \\
bind & : & ID \times ENV \times DAEMON \\
 & \to & VAR \cup CHANNEL \\
bind(id, [H \mid T], x) & = & \begin{cases} o - x & \text{if } H = (id,o) \\ bind(id, T, x) & \text{otherwise} \end{cases} \\
\bar{v} & = & bind(v, e) \\
\bar{c} & = & bind(c, e)
\end{array}$$

$\boxed{\mathsf{init\_chan}(\vec{c}, e, x)}$
if *cmd is* $\mathsf{init\_chan}(c_1, \ldots, c_r, e, x)$
then $agent(\bar{c_1})$ **:=** nil
$\quad \ldots$
$\quad agent(\bar{c_r})$ **:=** nil

Expressions

$$\begin{array}{rcl}
eval & : & VAR \to VAL \\
eval(v) & \stackrel{\text{def}}{=} & content(v) \\
eval & : & EXP \times ENV \times DAEMON \to VAL
\end{array}$$

$compile(v := t, e, m, x) = \mathsf{ass}(v, t, e, x)$

$\boxed{\mathsf{ass}(v,t,e,x)}$
if *cmd is* $\mathsf{ass}(v, t, e, x)$
then $content(\bar{v} + a)$ **:=** $eval(t, e, x)$
$\quad$ *proceed*

$compile(\mathsf{TIME}\,?\,v, e, m, x) = \mathsf{time}(v, e, x)$

$\boxed{\mathsf{time}(v,e,x)}$
if *cmd is* $\mathsf{time}(v, e, x)$
then $content(\bar{v} + a)$ **:=** $timer$
$\quad$ *proceed*

Conditionals

$compile(\mathsf{WHILE}\ B\, S, e, m, x) =$
$\quad l_{In},$
$\quad \mathsf{if}(\neg B, l_{Out}, e, x),$
$\quad compile(S, e, m, x),$
$\quad \mathsf{goto}(l_{In}),$
$\quad l_{Out}$

$compile(\mathsf{IF}\ B_1\, S_1\ \ldots\ B_r\, S_r, e, m, x) =$
$\quad \ldots$
$\quad \mathsf{if}(B_i, l_i, e, x),$
$\quad \ldots$
$\quad \mathsf{STOP},$
$\quad \ldots$
$\quad l_j,$
$\quad compile(S_j, e, m, x),$
$\quad \mathsf{goto}(l_{Out}),$
$\quad \ldots$
$\quad l_{Out}$
where $1 \le i, j \le r$

$\boxed{\mathsf{if}(b,l,e,x)}$
if *cmd is* $\mathsf{if}(b, l, e, x)$
then if $eval(b, e, x)$
$\quad$ **then** $loc(a)$ **:=** $labeled\_loc(l)$
$\quad$ **else** *proceed*

$\boxed{\mathsf{goto}(l)}$
if *cmd is* $\mathsf{goto}(l)$
then $loc(a)$ **:=** $labeled\_loc(l)$

$compile(\mathsf{SKIP}, e, m, x) = \mathsf{SKIP}$

$\boxed{\mathsf{skip}}$ **if** *cmd is* SKIP **then** *proceed*

$compile(\mathsf{STOP}, e, m, x) = \mathsf{STOP}$

$\boxed{\mathsf{stop}}$ **if** *cmd is* STOP **then** *dequeue*

Input

$$\begin{array}{rcl}
idle\ c & \stackrel{\text{def}}{=} & agent(c) = nil \\
clear\ c & \stackrel{\text{def}}{=} & agent(c) := nil \\
ready\ c & \stackrel{\text{def}}{=} & agent(c) \neq nil
\end{array}$$

$compile(c\,?\,v, e, m, x) = \mathsf{inp}(c, v, e, x)$

$\boxed{\mathsf{inp\_idle}(c, v, e, x)}$
if *cmd is* $\mathsf{inp}(c, v, e, x) \wedge idle\ \bar{c}$
then *sleep at* $next(loc(a))$
$\quad agent(\bar{c})$ **:=** a
$\quad place(a)$ **:=** $\bar{v} + a$
$\quad com\_mode(a)$ **:=** $input$

$\boxed{\mathsf{inp\_ready}(c, v, e, x)}$
if *cmd is* $\mathsf{inp}(c, v, e, x) \wedge ready\ \bar{c}$
then $content(\bar{v} + a)$ **:=** $mssg(agent(\bar{c}))$
$\quad$ *wakeup* $agent(\bar{c})$
$\quad$ *proceed*
$\quad$ *clear* $\bar{c}$

Output

$compile(c\,!\,t, e, m, x) = \mathsf{out}(c, t, e, x)$

$\boxed{\mathsf{out\_idle}(c, t, e, x)}$

if *cmd is* $\mathsf{out}(c, t, e, x) \wedge$ *idle* $\bar{c}$
then *sleep at next*$(loc(a))$
  $agent(\bar{c}) := a$
  $mssg(a) := eval(t, e, x)$

$\boxed{\mathsf{out\_ready}(c, t, e, x)}$

if *cmd is* $\mathsf{out}(c, t, e, x)$
  $\wedge$ *ready* $\bar{c}$
  $\wedge\ com\_mode(agent(\bar{c})) = input$
then $content(place(agent(\bar{c}))) := eval(t, e, x)$
  wakeup $agent(\bar{c})$
  proceed
  clear $\bar{c}$

$\boxed{\mathsf{out\_alt}(c, t, e, x)}$

if *cmd is* $\mathsf{out}(c, t, e, x)$
  $\wedge$ *ready* $\bar{c}$
  $\wedge\ com\_mode(agent(\bar{c})) \neq input$
then *sleep at next*$(loc(a))$
  $mssg(a) := eval(t, e, x),\ agent(\bar{c}) := a$
  if $com\_mode(agent(\bar{c})) = alt\_sleep$
  then *time-delete* $agent(\bar{c})$
    wakeup $agent(\bar{c})$
    $com\_mode(agent(\bar{c})) := alt\_run$

Alternation

$compile(\mathsf{ALT}\ G_1\,S_1 \ldots G_r\,S_r, e, m, x) =$
  $\mathsf{alt\_a}(\vec{G}, e, x)$,
  $\mathsf{alt\_s}(\vec{G}, \vec{l}, e, x)$,
  $\ldots$
  l_i,
  $\mathsf{inp}(c_i, v_i, e, x)$,
  $compile(S_i, e, m, x)$,
  $\mathsf{goto}(l_{Out})$,
  $\ldots$
  l_j,
  $compile(S_j, e, m, x)$,
  $\mathsf{goto}(l_{Out})$,
  $\ldots$
  l_{Out}
where $\vec{l} = l_1, \ldots, l_r$
  $\vec{G} = G_1, \ldots, G_r$
  $G_i = b_i\ :\ c_i\,?\,v_i$
  $G_j = b_j\ :\ (\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j$ or $\mathsf{SKIP}\ )$
  $1 \leq i \leq p < j \leq r$

$\boxed{\mathsf{alt\_a}(\vec{G}, e, x)}$
if *cmd is* $\mathsf{alt\_a}(\vec{G}, e, x)$
then
  $enable(b_1, \bar{c}_1)$
  $\ldots$
  $enable(b_p, \bar{c}_p)$
  if $\exists\, i\ (eval(b_i, e, x) \wedge ready\ \bar{c}_i)$
    $\vee\ t\_wait < timer$
    $\vee\ \exists\, k\ (eval(b_k, e, x))$
  then $com\_mode(a) := alt\_run$
    $loc(a) := next(loc(a))$
  else $com\_mode(a) := alt\_sleep$
    sleep at next$(loc(a))$
    if $\exists\, j\ (eval(b_j, e, x))$
    then seq $t_{min}(a) := t\_wait$
        time-insert
      endseq
where
  $G_i = b_i\ :\ c_i\,?\,v_i$
  $G_j = b_j\ :\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j$
  $G_k = b_k\ :\ \mathsf{SKIP}$
  $t\_wait = \begin{cases} \infty \text{ if } \forall\, j\ (\neg eval(b_j, e, x)), \text{ otherwise} \\ \min_j \{ eval(t_j, e, x) \mid eval(b_j, e, x) \} \end{cases}$
  $1 \leq i \leq p < j \leq q < k \leq r$

$\boxed{\mathsf{alt\_s\_com}} \mid \boxed{\mathsf{alt\_s\_time}} \mid \boxed{\mathsf{alt\_s\_skip}}\ (i, \vec{G}, \vec{l}, e, x)$
if *cmd is* $\mathsf{alt\_s}(\vec{G}, \vec{l}, e, x)$
  $\wedge\ eval(b_i, e, x)$
  $\wedge\ agent(\bar{c}_i) \notin \{ nil, a \} \mid timer > eval(t_i, e, x) \mid$
then
  $disable(b_1, \bar{c}_1)$
  $\ldots$
  $disable(b_p, \bar{c}_p)$
  $loc(a) := labeled\_loc(l_i)$
where
  $\vec{l} = l_1, \ldots, l_r$
  $\vec{G} = G_1, \ldots, G_r$
  $G_i = b_i\ :\ (\ c_i\,?\,v_i \mid \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_i \mid \mathsf{SKIP}\ )$

$\boxed{\mathsf{time\_wakeup}}$
if $time\_q$ not empty
  $\wedge\ timer > t_{min}(first(time\_q))$
then $com\_mode(first(time\_q)) := alt\_run$
  wakeup $first(time\_q)$
  time-delete $first(time\_q)$

Parallelism

$env\_size(\mathsf{SKIP}) = 0$

$env\_size(\mathsf{STOP}) = 0$

$env\_size(\mathsf{TIME}\,?\,v) = 0$

$env\_size(c\,?\,v) = 0$

$$env\_size(c\,!\,t) \;=\; eval\_size(t)$$
$$env\_size(v := t) \;=\; eval\_size(t)$$

$$env\_size(\mathsf{VAR}\; id_1,\ldots,id_r : S) \;=\; r + env\_size(S)$$
$$env\_size(\mathsf{CHAN}\; id_1,\ldots,id_r : S) \;=\; r + env\_size(S).$$

$$env\_size(\mathsf{WHILE}\; B\, S) = \\ max(\, eval\_size(B), env\_size(S)\,)$$

$$env\_size(\mathsf{IF}\; B_1\, S_1,\ldots B_r\, S_r) = \\ max_{i=1}^{r}\,(eval\_size(B_i), env\_size(S_i))$$

$$env\_size(\mathsf{SEQ}\; S_1 \ldots S_r) = \\ max_{i=1}^{r}\; env\_size(S_i)$$

$$env\_size(\mathsf{ALT}\; G_1\, S_1 \ldots G_r\, S_r) = \\ max_{i=1}^{r}\,(eval\_size(b_i), env\_size(S_i))$$

$$env\_size(\mathsf{PAR}\; S_1 \ldots S_r) = \\ \textstyle\sum_{i=1}^{r} env\_size(S_i)$$

$compile(\mathsf{PAR}\; S_1 \ldots S_r, e, m, x) =$
$\mathsf{par}(x, \vec{m}, \vec{l}, l_{Out})$,
l_1,
$compile(S_1, e, m, x)$,
end,
l_2,
$compile(S_2, e, m_2, m_2)$,
end,
...
l_r,
$compile(S_r, e, m_r, m_r)$,
end,
l_{Out}
where $\vec{l} = l_2, \ldots, l_r$
$\vec{m} = m_2, \ldots, m_r$
$m_1 = m$
$m_i = m_{i-1} + env\_size(S_{i-1})$
$2 \le i \le r$

$\boxed{\mathsf{par}(x, \vec{x}, \vec{l}, l_{Out})}$
if *cmd is* $\mathsf{par}(x, \vec{x}, \vec{l}, l_{Out})$
then $q := q.x'_1.\ldots.x'_r$
...
$father(x'_i) := a$
$loc(x'_i) := labeled\_loc(l_i)$
...
$count(a) := r$
$end\_par(a) := l_{Out}$
proceed
where $\vec{l} = l_1, \ldots, l_r$
$\vec{x} = x_1, \ldots, x_r$
$x'_i = a - x + x_i$
$1 \le i \le r$

$\boxed{\mathsf{end}}$
if *cmd is* end
then if $r > 1$
then *dequeue*
$r := r - 1$
else $a := x_f$
$loc(x_f) := a_{Out}$
where $r = count(x_f)$
$x_f = father(a)$
$l_{Out} = end\_par(x_f)$
$a_{Out} = labeled\_loc(l_{Out})$

7.3. Summary of $TRANSPUTER_{rel-code}$

Processor

$$content : LOC \rightarrow VAL$$
$$+1 : LOC \rightarrow LOC$$

$$FPtrReg, BPtrReg : DAEMON \cup \{nil\}$$
$$next : DAEMON \rightarrow DAEMON$$

$enqueue\; x \stackrel{\text{def}}{=}$ $BPtrReg := x$
if $FPtrReg = nil$
then $FPtrReg := x$
else $next(BPtrReg) := x$

$dequeue \stackrel{\text{def}}{=}$ **if** $FPtrReg = nil$
then $a := nil$
else $a := FPtrReg$
$loc(a) := loc(FPtrReg)$
$start := timer$
if $FPtrReg = BPtrReg$
then $FPtrReg := nil$
else $FPtrReg := next(FPtrReg)$

$$TPtrLoc : DAEMON \cup \{nil\}$$
$$TNextReg : LOC$$

$$Areg, Breg, Creg : VAL$$
$$random : VAL$$

$push(v) \stackrel{\text{def}}{=}$
$Areg := v,\; Breg := Areg,\; Creg := Breg$
$pop \stackrel{\text{def}}{=}$
$Areg := Breg, Breg := Creg,\; Creg := random$
$pop(v) \stackrel{\text{def}}{=}$
$v := Areg,\; pop$

$$Wreg, Ireg : VAL$$
$$proceed \stackrel{\text{def}}{=} Ireg := Ireg + 1$$

$$cmd \;:\; CODE$$
$$cmd\ is\ c \overset{\text{def}}{=} cmd = c \wedge mode = execute \wedge Wreg \neq nil$$
$$mode = \{load\text{-}label, compute\text{-}distance, load, fetch, execute\}$$

time–slice

if $mode = fetch \wedge Wreg \neq nil$
then if *elapsed*
$\quad\quad \wedge\ safe(content(Ireg))$
$\quad$ **then seq** *enqueue Wreg*
$\quad\quad\quad$ *dequeue*
$\quad\quad$ **endseq**
$\quad$ **else** $cmd := content(Ireg)$
$\quad\quad$ *proceed*
$\quad\quad$ $mode := execute$

Compilation, Resolving, Loading, ...

$$compile \;:\; STATEMENT \times ENV \times LOC \times DAEMON \to (CODE \cup LABEL)^*$$

$$loading\_loc \;:\; LOC$$
$$tmp\_prg, load\_prg \;:\; (CODE \cup LABEL)^*$$

if $mode = load\text{-}label$
then if $load\_prg = [\,]$
$\quad$ **then** $mode := compute\text{-}distances$
$\quad$ **else let** $load\_prg = [\,H\,|\,T\,]$
$\quad\quad$ $load\_prg := T$
$\quad\quad$ **if** $H \in LABEL$
$\quad\quad$ **then** $labeled\_loc(l) := loading\_loc$
$\quad\quad$ **if** $H \in CODE$
$\quad\quad$ **then** $tmp\_prg := append(tmp\_prg, c)$
$\quad\quad\quad$ $loading\_loc := loading\_loc+$

if $mode = compute\text{-}distances$
then if $load\_prg = [\,]$
$\quad$ **then** $mode := load$
$\quad$ **else let** $tmp\_prg = [\,H\,|\,T\,]$
$\quad\quad$ $loading\_loc := loading\_loc+$
$\quad\quad$ $tmp\_prg := T$
$\quad\quad$ **if** $H = c\ l_1 - l_2$
$\quad\quad$ **then** $load\_prg := append(load\_prg, H')$
$\quad\quad$ **else** $load\_prg := append(load\_prg, H)$
where $H' = c\ labeled\_loc(l_1) - labeled\_loc(l_2)$

load_cmd(c,t)

if $mode = load$
then if $load\_prg = [\,]$
$\quad$ **then** $mode := fetch$
$\quad$ **else let** $load\_prg = [\,H\,|\,T\,]$
$\quad\quad$ $load\_prg := T$
$\quad\quad$ $content(loading\_loc) := H$
$\quad\quad$ $loading\_loc := loading\_loc+$

Declarations

$compile(\mathsf{VAR}\ id_1, \ldots, id_r : S, e, m, x) =$
$\quad compile(S, e', m + r, x)$
where $\vec{id} = id_1, \ldots, id_r$
$\quad e' = (id_r, m + r - 1), \ldots, (id_1, m)\,|\,e$

$compile(\mathsf{CHAN}\ id_1, \ldots, id_r : S, e, m, x) =$
$\quad compile(\mathsf{init\_chan}(\vec{id}), e', m, x),$
$\quad compile(S, e', m + r, x)$
where $\vec{id} = id_1, \ldots, id_r$
$\quad e' = (id_r, m + r - 1), \ldots, (id_1, m)\,|\,e$

$compile(\mathsf{init\_chan}(\vec{c}), e, m, x) =$
$\quad \ldots$
$\quad$ `MINT`,
$\quad$ `STL` $bind(c_i, e, x)$,
$\quad \ldots$
where $\vec{c} = c_1, \ldots, c_r$
$\quad 1 \leq i \leq r$

$$ENV = (ID \times LOC)^*$$
$$bind \;:\; ID \times ENV \times DAEMON \to LOC$$
$$bind(id, [\,H\,|\,T\,], x) = \begin{cases} o - x & \text{if H = (id,o)} \\ bind(id, T, x) & \text{otherwise} \end{cases}$$

if *cmd is* `MINT` **then** $push(nil)$

if *cmd is* `STL` n **then** $pop(content(Wreg + n)$

Expressions

$$eval \;:\; EXP \times ENV \times DAEMON \to VAL$$

$compile(v := t, e, m, x) =$
$\quad \mathsf{eval}(t, e, x)$
$\quad$ `STL` $bind(v, e, x)$

if *cmd is* $\mathsf{eval}(t, e, x)$ **then** $push(eval(t, e, x))$

$compile(\mathsf{TIME}\,?\,v, e, m, x) =$
$\quad$ `LDTIMER`
$\quad$ `STL` $bind(v, e, x)$

if *cmd is* `LDTIMER`
then $push(timer)$

$compile(\mathsf{SKIP}) =$

$compile(\mathsf{STOP}, e, m, x) =$ `STOPP`

if *cmd is* `STOPP`
then $Areg, Breg, Creg := random$
$\quad$ *dequeue*

Conditionals

compile(WHILE $B\,S, e, m, x$) =
  l_{In},
  compile(if($\neg B, l_{Out}$), e, m, x),
  compile(S, e, m, x),
  J $l_{In} - l_{Out}$,
  l_{Out}

compile(if(B, L), e, m, x) =
  eval($\neg B, e, x$),
  CJ $L - l_{here}$,
  l_{here}

if *cmd is* J n **then** *Ireg* := *Ireg* + n

if *cmd is* CJ n
then **if** *Areg* = *false*
  then *Ireg* := *Ireg* + n
  else *Areg* := *Breg*
    Breg := *Creg*
    Creg := *random*

compile(IF $B_1\,S_1\ \ldots\ B_r\,S_r, e, m, x$) =
  ...
  compile(if(B_i, l_i), e, m, x),
  ...
  compile(STOP, e, m, x),
  ...
  l_i,
  compile(S_i, e, m, x),
  J $l_{Out} - l_{i+1}$,
  ...
  l_{Out}
where $l_{r+1} = l_{Out}$
  $1 \leq i \leq r$

Input

compile($c\,?\,v, e, m, x$) =
  LDLP *bind*(c, e, x),
  LDLP *bind*(v, e, x),
  IN

if *cmd is* LDLP n
then *push*(*Wreg* + n)

IN_idle
if *cmd is* IN $\wedge$ *content*(*Breg*) = *nil*
then *content*(*Breg*) := *Wreg*
  content(*Wreg* − 3) := *Areg*
  dequeue

IN_ready
if *cmd is* IN $\wedge$ *content*(*Breg*) $\neq$ *nil*
then *content*(*Areg*) := *content*(*agent*)
  agent := *nil*
  enqueue agent
where *agent* = *content*(*Breg*)

Output

compile($c\,!\,t, e, m, x$) =
  LDLP *bind*(c, e, x),
  eval(t, e, x)
  AJW −1
  OUT
  AJW +1

if *cmd is* AJW n **then** *Wreg* := *Wreg* + n

OUT_idle
if *cmd is* OUT $\wedge$ *content*(*Breg*) = *nil*
then *content*(*Breg*) := *Wreg*
  content(*Wreg*) := *Areg*
  dequeue

OUT_ready
if *cmd is* OUT
  $\wedge$ *agent* $\neq$ *nil*
  $\wedge$ *place* $\notin$ { *alt_sleep*, *alt_run*, *alt_ready* }
then *content*(*place*) := *Areg*
  agent := *nil*
  enqueue agent
where *agent* = *content*(*Breg*)
  place = *content*(*agent* − 3)

OUT_alt
if *cmd is* OUT
  $\wedge$ *agent* $\neq$ *nil*
  $\wedge$ *place* $\in$ { *alt_sleep*, *alt_run*, *alt_ready* }
then *dequeue*
  agent := *Wreg*
  content(*Wreg*) := *Areg*
  if *place* = *alt_sleep*
  then *time-delete agent*
    place := *alt_run*
    enqueue agent
where *agent* = *content*(*Breg*)
  place = *content*(*agent* − 3)

Alternation

$compile(\mathsf{ALT}\ G_1\, S_1 \ldots G_r\, S_r, e, m, x)$ =
 `AJW` -1,
 $compile(\mathsf{alt\_a}(\vec{G}), e, m, x-1)$,
 $compile(\mathsf{alt\_s}(\vec{G}, \vec{l}), e, m, x-1)$,
 ...
 l_i,
 `AJW` $+1$,
 $compile(c_i\,?\,v_i, e, m, x)$,
 $compile(S_i, e, m, x)$,
 `J` l_{Out},
 ...
 l_j,
 `AJW` $+1$,
 $compile(S_j, e, m, x)$,
 `J` l_{Out},
 ...
 l_{Out}
where $\vec{l} = l_1, \ldots, l_r$
 $\vec{G} = G_1, \ldots, G_r$
 $G_i = b_i\ :\ c_i\,?\,v_i$
 $G_j = b_j\ :\ (\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j$ or $\mathsf{SKIP}\ )$
 $1 \leq i \leq p < j \leq r$

if *cmd* is `ENBC`
then if $Areg = true$
 then if $content(Breg) = nil$
  then $content(Breg)$:= $Wreg$
  elsif $content(Breg) \neq Wreg$
  then $content(Wreg - 3)$:= alt_ready
 $Breg$:= $Creg$
 $Creg$:= $random$

if *cmd* is `ENBT`
then if $Areg = true$
 then if $content(Wreg - 4) = false$
  then $content(Wreg - 4)$:= $true$
   $content(Wreg - 5)$:= $Breg$
  elsif $Breg < content(Wreg - 5)$
  then $content(Wreg - 5)$:= $Breg$
 $Breg$:= $Creg$
 $Creg$:= $random$

if *cmd* is `ENBS`
then if $Areg = true$
 then $content(Wreg - 3)$:= alt_ready

if *cmd* is `TALTWT`|`ALTWT`
then if $content(Wreg - 5) \geq timer \wedge$ |
  $content(Wreg - 3) = alt\_run$
 then if $content(Wreg - 4)$
  then *time-insert*
  $content(Wreg - 3)$:= alt_sleep
  dequeue
 $Areg, Breg, Creg$:= $random$
 $content(Wreg)$:= nil

$compile(\mathsf{alt\_a}(\vec{G}), e, m, x)$ =
 `ALT`,
 ...
 `LDLP` $bind(c_i, e, x)$,
 $\mathsf{eval}(b_i, e, x)$,
 `ENBC`,
 ...
 $\mathsf{eval}(t_j, e, x)$,
 $\mathsf{eval}(b_j, e, x)$,
 `ENBT`,
 ...
 $\mathsf{eval}(b_r, e, x)$,
 `ENBS`,
 ...
 `ALTWT`
where
 $\vec{G} = G_1, \ldots, G_r$
 $G_i = b_i\ :\ c_i\,?\,v_i$
 $G_j = b_j\ :\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j$
 $G_k = b_k\ :\ b_k\ :\ \mathsf{SKIP}$
 $1 \leq i \leq l < j \leq p < k \leq r$

if *cmd* is `TALT` | `ALT`
then $content(Wreg - 4)$:= $false$ |
 $content(Wreg - 3)$:= alt_run

$compile(\mathsf{alt\_s}(\vec{G}, \vec{l}), e, m, x)$ =
 ...
 `LDLP` $bind(c_i, e, x)$,
 $\mathsf{eval}(b_i, e, x)$,
 `LDC` $l_i - l_{here}$,
 `DISC`,
 ...
 $\mathsf{eval}(t_j, e, x)$,
 $\mathsf{eval}(b_j, e, x)$,
 `LDC` $l_j - l_{here}$,
 `DIST`,
 ...
 $\mathsf{eval}(b_k, e, x)$,
 `LDC` $l_k - l_{here}$,
 `DISS`,
 ...
 `ALTEND`,
 l_{here}
where $\vec{l} = l_1, \ldots, l_r$
 $\vec{G} = G_1, \ldots, G_r$
 $G_i = b_i\ :\ c_i\,?\,v_i$
 $G_j = b_j\ :\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j$
 $G_k = b_k\ :\ b_k\ :\ \mathsf{SKIP}$
 $1 \leq i \leq p < j \leq q < k \leq r$

if *cmd is* `DISC`
then if *Breg*
  then if $content(Creg) = Wreg$
    then $content(Creg) := nil$
      $Areg := false$
    elsif $content(Creg) \neq nil$
      $\wedge\ content(Wreg) = nil$
    then $content(Wreg) := Areg$
      $Areg := true$
    else $Areg := false$
  else $Areg := false$
  $Breg, Creg := random$

if *cmd is* `DIST`
then if *Breg*
  then if $Creg < timer \wedge content(Wreg) = nil$
    then $content(Wreg) := Areg$
      $Areg := true$
    else $Areg := false$
  else $Areg := false$
  $Breg, Creg := random$

if *cmd is* `DISS`
then if *Breg*
  then if $content(Wreg) = nil$
    then $content(Wreg) := Areg$
      $Areg := true$
    else $Areg := false$
  else $Areg := false$
  $Breg, Creg := random$

if *cmd is* `ALTEND`
then $Ireg := Ireg + content(Wreg)$

$\boxed{\text{time\_wakeup}}$
if $TPtrLoc \neq nil \wedge timer > TNextReg$
then $com\_mode(TPtrLoc) := alt\_run$
  enqueue $TPtrLoc$
  time-delete $TPtrLoc$

Parallelism

$$\begin{aligned} env\_size &: STATEMENT \rightarrow \mathbf{N} \\ spec\_size &: STATEMENT \rightarrow \mathbf{N} \\ work\_size &: STATEMENT \rightarrow \mathbf{N} \\ work\_size(S) &= env\_size(S) + spec\_size(S) \end{aligned}$$

$$\begin{aligned} env\_size(\mathsf{SKIP}) &= 0 \\ env\_size(\mathsf{STOP}) &= 0 \\ env\_size(\mathsf{TIME}\,?\,v) &= 0 \\ env\_size(c\,?\,v) &= 0 \end{aligned}$$

$$\begin{aligned} env\_size(c\,!\,t) &= eval\_size(t) \\ env\_size(v := t) &= eval\_size(t) \end{aligned}$$

$$\begin{aligned} env\_size(\mathsf{VAR}\ id_1, \ldots, id_r : S) &= r + env\_size(S) \\ env\_size(\mathsf{CHAN}\ id_1, \ldots, id_r : S) &= r + env\_size(S). \end{aligned}$$

$env\_size(\mathsf{WHILE}\ B\ S) =$
  $max(\ eval\_size(B), env\_size(S)\ )$

$env\_size(\mathsf{IF}\ B_1\ S_1, \ldots B_r\ S_r) =$
  $max_{i=1}^{r}\ (eval\_size(B_i), env\_size(S_i))$

$env\_size(\mathsf{SEQ}\ S_1 \ldots S_r) =$
  $max_{i=1}^{r}\ env\_size(S_i)$

$env\_size(\mathsf{ALT}\ G_1\ S_1 \ldots G_r\ S_r) =$
  $max_{i=1}^{r}\ (eval\_size(b_i), env\_size(S_i))$

$env\_size(\mathsf{PAR}\ S_1 \ldots S_r) =$
  $env\_size(S_1) + \sum_{i=2}^{r} work\_size(S_i)$

$$\begin{aligned} spec\_size(\mathsf{SKIP}) &= 2 \\ spec\_size(\mathsf{STOP}) &= 2 \\ spec\_size(v := t) &= 2 \\ spec\_size(\mathsf{TIME}\,?\,v) &= 2 \\ spec\_size(c\,?\,v) &= 3 \\ spec\_size(c\,!\,t) &= 4 \end{aligned}$$

$$\begin{aligned} spec\_size(\mathsf{VAR}\ id_1, \ldots, id_r : S) &= spec\_size(S) \\ spec\_size(\mathsf{CHAN}\ id_1, \ldots, id_r : S) &= spec\_size(S) \end{aligned}$$

$spec\_size(\mathsf{WHILE}\ B\ S) = spec\_size(S)$

$spec\_size(\mathsf{IF}\ B_1\ S_1, \ldots B_r\ S_r) =$
  $max_{i=1}^{r}\ (spec\_size(S_i)$

$spec\_size(\mathsf{SEQ}\ S_1 \ldots S_r) =$
  $max_{i=1}^{r}\ spec\_size(S_i)$

$spec\_size(\mathsf{ALT}\ G_1\ S_1 \ldots G_r\ S_r) =$
$$\begin{cases} max_{i=1}^{r}\ (4, spec\_size(S_i)) & \text{case (a)} \\ max_{i=1}^{r}\ (6, spec\_size(S_i)) & \text{case (b)} \end{cases}$$

$spec\_size(\mathsf{PAR}\ S_1 \ldots S_r) =$
$max(4, spec\_size(S_1))$

$compile((\mathsf{PAR}\ S_1 \ldots S_r), e, m, x) =$
$\quad$ `AJW` -2
$\quad compile(\mathsf{par}(x, \vec{m}, \vec{l}, l_{Out}), e, m, x-2),$
$\quad compile(S_1, e, m_1, x-2),$
$\quad compile(\mathsf{end}(x-2), e, m, x-2),$
$\quad l_2,$
$\quad compile(S_2, e, m_2, m_2),$
$\quad compile(\mathsf{end}(m_2), e, m_2, x-2),$
$\quad \ldots$
$\quad l_r,$
$\quad compile(S_r, e, m_r, m_r),$
$\quad compile(\mathsf{end}(m_r), e, m_r, x-2),$
$\quad l_{Out}$, `AJW` $+2$
where $\vec{l} = l_2, \ldots, l_r$
$\quad \vec{m} = m_2, \ldots, m_r$
$\quad m_1 = m$
$\quad m_i = m_{i-1} + env\_size(S_{i-1}) + spec\_size(S_i)$
$\quad 2 \leq i \leq r$

$compile(\mathsf{par}(x, \vec{x}, \vec{l}, l_{Out}), e, m, x) =$
$\quad$ `LDC` r,
$\quad$ `STL` 1,
$\quad$ `LDC` $l_{Out} - l_{here}$
$\quad$ `LDPI`,
$\quad l_{here}$,
$\quad$ `STL` 0,
$\quad \ldots$
$\quad$ `LDC` $l_i - l'_i$,
$\quad$ `LDLP` $x_i - x$
$\quad$ `STARTP`,
$\quad l'_i$,
$\quad \ldots$
where $\vec{l} = l_1, \ldots, l_r$
$\quad \vec{x} = x_1, \ldots, x_r$
$\quad 1 \leq i \leq r$

$compile(\mathsf{end}(x_i), e, m, x) =$ `LDLP` $x - x_i$, `ENDP`

if *cmd is* `STARTP`
then $loc(Wreg + Areg) := Breg$
$\quad$ *enqueue* $Wreg + Areg$

if *cmd is* `ENDP`
then if $content(x_f) > 1$
$\quad$ **then** *dequeue*
$\quad\quad content(x_f + 1) := content(x_f + 1) - 1$
$\quad$ **else** $Ireg := content(x_f)$
$\quad\quad Wreg := x_f$
where $x_f = Wreg + Areg$

REFERENCES

[BowHePand:90]: J.P. Bowen, He Jifeng, P.K. Pandaya, 1990, *An Approach to Verifiable Compiling Specification and Prototyping*, Springer Verlag, LNCS 456, pp. 45-59.

[BowHe:93]: J.P. Bowen, He Jifeng, 1993, *Specification, Verification and Prototyping of an Optimized Compiler*, in: *Formal Aspects of Computing.*

[Bowen:93]: J.P. Bowen,1993, *From Programs to Object Code and back again using Logic Programming: Compilation and Decompilation*, in: *Journal of Software Maintenance: Research and Practice* 5(4):205-234.

[BĐR:94]: E.Börger, I.Đurđanović & D.Rosenzweig, 1994, *Occam: Specification and Compiler Correctness. Part I: The Primary Model*, E.-R. Olderog (Ed.), Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi), pages 489-508, North-Holland 1994

[BöRo:95a]: E.Börger & D.Rosenzweig, 1995, *A mathematical definition of Full Prolog*, in: *Science of Computer Programming* (to appear).

[BöRo:95b]: E.Börger & D.Rosenzweig, 1995, *The WAM—Definition and Compiler Correctness*, in: *Logic Programming: Formal Methods and Practical Applications*, C.Beierle, L.Plümer, eds., North-Holland, Series in Computer Science and Artificial Intelligence, pages 20-90 (chapter 2).

[Graham:90]: Ian Graham, 1990, *The Transputer Handbook*, Prentice Hall.

[Gur:95]: Y. Gurevich. Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, Ed. E. Börger, Oxford University Press, 1995.

[Hoare:91]: C.A.R. Hoare, *refinement algebra proves correctness of compiling specifications.* In C.C. Morgan and J.C.P. Woodcock, editors, 3^{rd} *Refinement Workshop*, Workshops in Computer Science, pages 33–48. Springer Verlag, 1991.

[Hoare et al.:87]: C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorenson, J.M. Spivey, and B.A. Sufrin. *Laws of programming.* Communications of the ACM, 30(8):672-687, August 1987.

[Hoare et al.:93]: C.A.R. Hoare, He Jifeng, and A. Sampaio. *Normal form approach to compiler design.* Acta Informatica, 30:701-739, 1993.

[Inmos]: Inmos, *Transputer Implementation of Occam.* In: *Communication Process Architecture*, Prentice Hall, note 21.

[Inmos:88]: Inmos, 1988, *Transputer Instruction Set – A compiler writer's guide*, INMOS document 72 TRN 119 05, Prentice Hall.

[MMO:95]: Markus Müller–Olm, *Structuring Code Generator Correctness Proofs by Stepwise Abstracting the Machine Language's Semantics*, ProCos II Esprit basic Research 7071, [Kiel MMO 12/3] (doctoral dissertation).

[PageLuk:91]: I. Page, W. Luk, 1991, *Compiling Occam into field-programmable gate arrays*, in *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, Abingdon EE&CS Books, pp. 271-283.

[Mitchell:90]: D.A.P. Mitchell et al, 1990, *Inside the Transputer*, Blackwell Scientific Publications.

A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study)*

Egon Börger
Dipartimento di Informatica
Università di Pisa
Corso Italia 40
56125 Pisa, Italy
E-Mail: *boerger@di.unipi.it*
Fax: ++39 50 887226

Giuseppe Del Castillo
Universität Paderborn
FB17 Informatik
Warburger Str. 100
33095 Paderborn, Germany
E-Mail: *giusp@uni-paderborn.de*
Fax: ++49 5251 603427

Abstract

We present a design approach which allows us to formally specify a real–life processor as composed out of its basic architectural (formally specified) components. The methodology provides means to rely upon hierarchical refinements and modular structuring of the specifications as a discipline to control the behaviour of complex units in terms of the behaviour of their components. In particular this enables us to prove interesting dynamic properties about the processor in terms of properties of its basic architectural components. The method makes use of Gurevich's concept of evolving algebra. We have developed the method to accomplish a reverse engineering project for the VLSI implemented microprocessor zCPU, the controller of the successful APE100 massively parallel machine.

Introduction

The APE100 massively parallel processor has been built as a dedicated machine for floating point intensive scientific applications and has proved to be rather successful for numerical simulations in Lattice Gauge Theory (see [3, 4]). As preparation for a possible upgrade to a new APE1000 machine (see [5]) we have accepted the challenging reverse engineering task to construct formal models for the architecture in such a way that the upgrading process can be guided by these models. The models are intended to provide precise descriptions between the existing block diagrams and verbal explanations on one side and the C-code for the APE100 simulator on the other side; they are required to be usable for producing executable prototypes and to offer the possibility to experiment with design decisions at various levels of abstraction.

We have developed a series of formal models, at different levels of abstraction, which correspond to views of the architecture as provided by different languages in the APE100 compilation chain (a characteristic part of the advanced software environment of APE100, see [1]). The ground model

*An abridged version of this paper will appear in the Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95).

APESE has been defined in [7]; it reflects the APE100 model of parallel execution as viewed by the user who approaches the machine as programmer in the high level parallel programming language Apese, a parallel Fortran like user expandable language especially designed for APE100. In this paper we show how this model can be transformed by stepwise refinement to a provably correct model LEX (loadable executable code) of APE100 at the hardware level, going through mainly two other intermediate models Assembler and ZIC (zCPU intermediate code) which correspond to languages of the APE100 compilation chain.

We concentrate our attention here on the VLSI superscalar integer processor zCPU which acts as controller for APE100 and represents the most original part of the project (including the pipelining and VLIW parallelism for the execution of compiled ZIC code; see [2] for technical details of the zCPU). In section 1 we provide a formal description of the standard architectural components of zCPU, namely the register file RF (which plays the role of a cache system), the (extended) ALU, the sequencing and addressing unit (which works as dedicated hardware independently and concurrently with the main ALU but can cooperate with the latter to calculate complex addresses for say multidimensional array references), the condition code unit, an instruction decoding unit and other special auxiliary devices and registers. In section 2 we define a method by which given units can be composed in a precise way to complex units. Using well known techniques from the literature (see [12]) the composition can be done in a modular way. This greatly simplifies the task to prove properties of complex units in terms of properties of their simpler constituents. Defining the components as evolving algebras (in the sense of Gurevich [14]) to which we add entries and exits allows us to adopt also the evolving algebra refinement techniques which have been used successfully to formally specify and prove properties of complex systems (see for ex. [11, 8, 9]).

In section 3 we explain how for the formal model LEX of the zCPU processor we can make the following informal statement into a precise mathematical assertion and give a mathematical proof for it.

Main Theorem. Under precisely stated assumptions on the compiler, the model LEX of the processor zCPU executes compiled Apese programs correctly.

In the appendices we provide the full abstract definition of the zCPU processor, obtained by putting together the definitions of its basic units.

The proof of the theorem, which will be given in a sequel to this paper, proceeds by a series of correctness theorems which establish that each level n of the refinement hierarchy leading from APESE to LEX is correctly implemented by the model at level n+1. Note that the "general strategy used throughout in APE100 of moving functions from hardware to software, as long as this does not entail performance penalties" ([6]) results in a sometimes very subtle interplay between the compiler and the hardware, in particular where it comes to deal with VLIW and pipelining parallelism. The freedom of abstraction which comes with the notion of evolving algebras allows us to reflect this strong role of the compiler within the APE100 project without going into its details; since we want to concentrate on the architecture, the compiler properties are explicitly formulated as abstract assumptions for the claims on the behaviour of the processor.

This paper is not the place to compare the new evolving algebra approach to the numerous other formal specification methodologies in the literature. The goal of this paper is to report on a challenging reverse engineering project which has been carried out successfully using the evolving algebra approach. We want to convince the practitioner by an example from real-life that:

- one can use the evolving algebra specification methodology to produce readable but nevertheless precise specifications without previous formal training and without formal overhead;

- the evolving algebra specification method scales to complex systems.

This explains also why we do not give here a formal definition of the underlying semantics of evolving algebras (which however has been rigorously defined in [14]); our specifications can be read and understood as abstract code which can serve as basis for the implementation of executable prototypes.

1 The Datapath Components of the zCPU

The zCPU processor is built out of several main units, namely:

- the register file RF;
- the ALU_MPY_DIV unit (ALU for brevity), an arithmetico-logical unit containing special independent devices for multiplication and division;
- the CC&STATUS unit, which processes the condition codes and the state information for handling jumps and exceptions;
- the Address Generation Unit AGU, for calculating program and data addresses;
- the Input/Output Subsystem IOS, providing the interface between zCPU and its data memory (DATAMEM);
- the INSTR unit, for instruction fetch;
- the units DATAMEM and PROGMEM, data and program memory of the zCPU.

In addition, other components are needed in order to coordinate the operations of the units listed above, namely some internal registers and a DECODE unit, for decoding instructions words read from program memory into the appropriate control codes required by the units.

Each unit is specified as an evolving algebra (in the sense of [14]) with entries and exits. The latter are vehicles for an explicit description of a desired input/output behaviour. This behaviour is defined by finitely many rules of the evolving algebra and possibly some conditions on the functions which appear in the rules. Each rule is of the form

$$\textbf{if } \mathit{Cond} \textbf{ then } \mathit{Updates}$$

where *Cond* is a first-order expression and *Updates* a finite set of function updates

$$f(t_1, \ldots, t_n) := t$$

which are executed *simultaneously* each time *Cond* is true[1]. For the description of the parallelism in APE100 it is convenient to rely upon the lock-step interpretation of evolving algebras under which

[1]Note that such a rule transforms a structure ("state") — i.e. a set of functions $\mathcal{S}$ over given domains — into another structure $\mathcal{S}'$ which differs from $\mathcal{S}$ by some of the functions being changed for some arguments. Functions which appear in an evolving algebra but never as outer function f of a function update $f(t_1, \ldots, t_n) := t$ are called *external*: they represent the environment for the evolving algebra.

in each step each rule which can be applied is applied. (For an exact definition of this lock-step semantics of evolving algebras see [14]).

The entries and exits can be viewed as terms, which include 0-ary functions. As an experiment we will use in this paper essentially only 0-ary functions for entries and exits. Each function can be constrained by conditions, which can serve various purposes. For example, exits are often defined by equations; in the special case of a combinational unit all of them are defined as functions of only entries. Another use of conditions on functions are (integrity) contraints, which are assumed (or guaranteed) for a correct behaviour of the unit under consideration.

We are going to define now three characteristic units, namely RF, ALU_MPY_DIV and a register: the reader will recognize (and can check through the details provided in the appendices) that the other basic units can be specified in a similar way.

1.1 The register file RF

The zCPU register file defines the interaction between 64 general registers and the rest of the processor. The content $reg(addr)$ of any register $addr \in \{0, \ldots, 63\}$ becomes accessible through one of the five RF-ports $OutPort_i$ $(i = 1, 2, 3, 5)$ and in_port_j $(j = 4, 5)$, where the fifth can be used as input (in_port_5) and as output $(OutPort_5)$ port[2].

The values of the RF-exits $OutPort_i$ are computed from the entries $addr_i$ using reg by the rules

$$OutPort_i := reg(addr_i) \quad \text{for } i = 1, 2, 3. \tag{1}$$

The entries $addr_4 \in \{0, \ldots, 63\}$, $in\_port_4 \in INTEGER$ and $write\_enable_4 \in \{0, 1\}$ are used to update reg on $addr_4$ to in_port_4 if the input port number 4 is enabled for writing: this is formalized by the rule

$$\textbf{if } write\_enable_4 \textbf{ then } reg(addr_4) := in\_port_4. \tag{2}$$

Port number 5 is special because it can be used for either reading or writing: in the latter case the value of $OutPort_5$ becomes undefined. Thus, the behaviour of the RF unit with entries $addr_i, i \in \{1, \ldots, 5\}$, $in\_port_j, write\_enable_j, j \in \{4, 5\}$ and exits $OutPort_k, k \in \{1, 2, 3, 5\}$ is defined by rules (1), (2) and by the following rule for RF-port number 5:

$$\begin{array}{ll} \textbf{if} & write\_enable_5 \\ \textbf{then} & reg(addr_5) := in\_port_5 \\ & OutPort_5 := undef \\ \textbf{else} & OutPort_5 := reg(addr_5) \end{array} \tag{3}$$

The RF unit works under the additional assumption that it is not allowed to read and to write a register at the same time, as well as to write to the same register through the two input ports 4 and 5. These conditions are formalized by the following integrity contraints on the RF entries:

$$\begin{array}{rcl} write\_enable_4 & \Rightarrow & addr_4 \notin \{addr_1, addr_2, addr_3, addr_5\} \\ write\_enable_5 & \Rightarrow & addr_5 \notin \{addr_1, addr_2, addr_3\} \\ write\_enable_4 \wedge write\_enable_5 & \Rightarrow & addr_4 \neq addr_5. \end{array}$$

[2] We denote internal registers by capital initial letters and try to adhere to the terminology of [6]. The reader should not confuse the general registers of the register file and the internal registers of the zCPU. The former are named by numbers $0, \ldots, 63$ and their contents accessed using the function reg, the latter are viewed by us as 0-ary functions which can be updated by transition rules (see below 1.3).

A peculiarity of the RF units consists in its exits: usually exits are defined by equations and possibly depend on the internal state of the unit, while updates are used to modify the internal state of the unit. In RF the exits are written through updates: this is just a notational shorthand similar to that used in 1.3. In fact, the functions in_port_i of RF are internal registers of RF, whose exits are also exits of RF itself.[3]

1.2 The arithmetical unit ALU_MPY_DIV

The arithmetical unit of zCPU consists of three parts which can work in parallel, one for the additive, logical and shift operations, one for (3 types of) multiplication and one for division. The entries are *math_code* (indicating the operation code), op_i (for the two operands), three condition code entries $carry_{\mathrm{in}}$, $extend_{\mathrm{in}}$, $zero_{\mathrm{in}}$ and four entries *md_ctrl* for multiplier and divider control. The exits are $math\_res_{\mathrm{out}}$ for the computed result, and one for each condition code (the above plus negative value, overflow, division by zero). These exits are characterized in a purely functional way. Technically speaking this means that we abstract from the time needed by the device to compute the values at the exits which correspond to the values appearing at the entries. In particular, $math\_res_{\mathrm{out}}$ is defined as a function

$$math\_res_{\mathrm{out}} = math\_res(math\_code, md\_ctrl, op_1, op_2, carry_{\mathrm{in}}, extend_{\mathrm{in}}).$$

(The exits for condition codes are defined in a similar way, see appendix A.2).

In case *math_code* indicates an additive, logical or shift operation, $math\_res_{\mathrm{out}}$ is the usual combinational function of op_1, op_2, $carry_{\mathrm{in}}$ and $extend_{\mathrm{in}}$ (and the functions corresponding to the condition codes are similarly defined).

In case *math_code* indicates an operation for MPY or DIV, the entry *md_mux* (in *md_ctrl*) distinguishes between multiplicative operations and division. In case of a multiplication the entry *md_mux* and an additional entry *add_mul* will determine which function will be used to compute the value of the operation in question on the arguments op_1, op_2.

This function is however not combinational, because more than one clock cycle is needed for its computation by the unit. The two dedicated hardware devices which execute multiplications and divisions interfere with the main ALU pipeline only when the multiplication or division instructions are issued or when the result is ready for write–back. Therefore the ALU can execute other operations while multiplications or divisions are in progress. As a consequence—we consider now the case of multiplications—at the beginning (when the entry *mul_in* in *md_ctrl* satisfies $mul\_in = 0$) the operands op_i must be stored in internal registers $MulOp_i$ of the multiplier and a counter (*MulStep*) must be set to determine when the multiplication result is ready, namely after 2 further clock cycles. The compiler is assumed to guarantee that the distance between two consecutive multiplicative instructions is at least 3, i.e. that *mul_in* changes from 0 to 1 and will not assume 0 again before 2 clock cycles.

Thus, the behaviour of the MPY part of the ALU_MPY_DIV unit is formalized by the rule

$$\textbf{if } mul\_in = 0 \textbf{ then } \mathrm{start\_mul}(op_1, op_2) \textbf{ else } \mathrm{mul\_busy}$$

[3] This notation is used in other units as well (e.g. DECODE): when a register name (denoted by upper case initial) appears in the list of exits of a unit, it should be interpreted as explained here.

where

$$\begin{aligned}
\text{start_mul}(op_1, op_2) \quad &\equiv \quad MulOp_1 := op_1 \\
& \quad MulOp_2 := op_2 \\
& \quad MulStep := 1 \\
\text{mul_busy} \quad &\equiv \quad MulStep := MulStep + 1.
\end{aligned}$$

The function *mul_ready*, indicating when the result of the multiplication is ready to be written back into the destination register, is defined by:

$$mul\_ready(mul\_in) \quad = \quad mul\_in \neq 0 \wedge MulOp_1 \neq undef \wedge MulOp_2 \neq undef \wedge MulStep \geq 2.$$

A similar formalization is done for the behaviour of the DIV–subdevice of ALU_MPY_DIV, making use of the fourth entry *start_div* in *md_ctrl* (see appendix A.2).

1.3 A register unit

A *register*[4] X can be viewed as a very simple unit, represented by an evolving algebra with one entry $X.in$ and one exit $X.out$, as well as a 0-ary dynamic function X holding its contents. The unit contains the transition rule

$$X := X.in$$

which formalizes writing the given value into the register, and a definition

$$X.out = X$$

which defines the output of the register unit simply as the content of the register.[5] Once this has been said, we clearly identify $X.out$ and X: in the following we shall write only X, without distinguishing it notationally from $X.out$.

2 Composition of the Datapath Components

In this section we show how to compose the basic units to the zCPU processor in such a way that the behavioural properties we are interested in can be controlled by the "modular" composition technique.

In doing this, we use the natural pictorial representation of evolving algebras with entries and exits as boxes with ingoing and outgoing arrows. Composing units then means to connect outgoing with ingoing arrows. In this paper we can only refer to the literature (see for example [12]), where it is shown how the resulting notion of computation of a "composed" unit can be defined rigorously in terms of the notion of computation of the components. It is also shown there that all the combinations we need can be obtained in a modular way from the basic units by applying parallel or sequential composition and feedback.

We proceed now to show as example how to connect the register file with the ALU_MPY_DIV unit, using some additional small units so as to obtain the kernel for the interpretation of arithmetic

[4] Note that we refer here to *internal registers*, not to the *general purpose registers* of the Register File, represented in our model by the function *reg*.

[5] Note that the crucial effect of a register is that the value of the entry is made available for the next step at the exit.

intructions. The connection of outgoing arrows with ingoing arrows is expressed by identifying exits with entries; as is well known a global specification of the composed unit can be obtained by substituting in the appropriate places of the components the entries with the exits, according to those identifications.

Here we identify the RF-exits (output ports) $OutPort_i$ ($i = 1, 2$) with the entries op_i of the ALU_MPY_DIV unit, the ALU_MPY_DIV exit $math\_res_{\mathrm{out}}$ with the Res register entry in and the Res register exit out with the RF-entry in_port_4.

The fields of the current (arithmetic) instruction are contained in 4 additional registers, namely MAC for the mathematical operation code, Ri for the address of the register which contains the i-th operand ($i = 1, 2$), RR for the address of the destination register. We connect Ri with the entry $addr_i$ of RF and RR with $addr_4$ (passing through two additional registers RR_2 and RR_3, which delay the value for two steps)[6]. MAC is connected to the $math\_code$ entry of ALU_MPY_DIV (again passing through a delay register MAC_2). Since the value of MAC is also needed for computing a certain portion of the control code, we connect it to the combinational unit DECODE (instruction decoding unit).

In particular, DECODE provides the information for enabling writing through the RF-port number 4. Since this value is needed after 2 steps — namely the time needed to compute the result of the arithmetic operation — it is passed from a DECODE-exit to the RF-entry $write\_enable_4$ going through two delay registers WER and WER_2.

In this way we have obtained the following unit (arithmetic subunit) which suffices to compute the result of simple arithmetic instructions, such as the addition instruction `ADD`, as we show in the next section. Note that this unit formalizes a portion of the zCPU block diagram in [6].

For use in the next section, we reassume the relevant rules and conditions of the thus modified arithmetic subunit in the black box form, where all substitutions shown by the figure are done (namely $[R1/addr_1, R2/addr_2, WER_2/write\_enable_4, RR_3/addr_4, Res/in\_port_4]$ for the RF unit and $[MAC_2/math\_code, OutPort_1/op_1, OutPort_2/op_2]$ for the ALU_MPY_DIV unit):

$$\begin{array}{l}
OutPort_1 := reg(R1) \\
OutPort_2 := reg(R2) \\
MAC_2 := MAC \\
WER := MAC \in \{\,\mathtt{ADD}, \ldots\} \\
RR_2 := RR
\end{array}$$

$$\begin{array}{l}
Res := math_res(MAC_2, md_ctrl, OutPort_1, OutPort_2, carry_{\mathrm{in}}, extend_{\mathrm{in}}) \\
WER_2 := WER \\
RR_3 := RR_2
\end{array}$$

$$\textbf{if } WER_2 \textbf{ then } \ reg(RR_3) := Res$$

(note the rules above are grouped according to the pipeline stages).

Similar constructions can be made to compose the units for the execution of input/output instructions, jumps etc. (see Appendix C).

[6] For reasons of exposition, instead of just one we introduce here two delay registers for RR which allows us to abstract from some peculiarity in the compilation scheme of APE100. See also the footnote at the end of the next section.

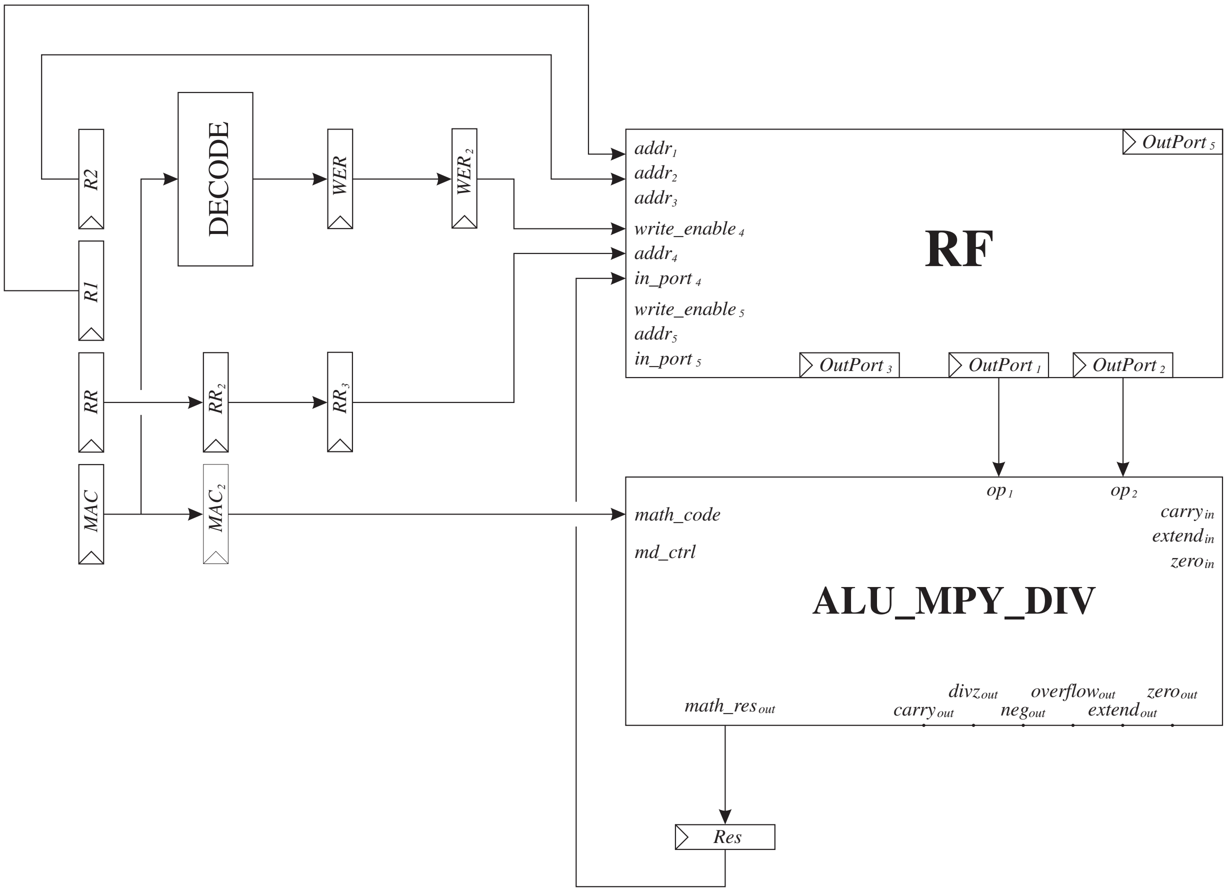


Figure 1: Arithmetic subunit of the zCPU

3 Correct Implementation of the ZIC Instruction Set

The two crucial aspects of the compilation strategy for the zCPU are the following: (i) Assembler instructions are macro-expanded into short sequences of ZIC instructions, whose execution is easily shown to implement the Assembler instructions correctly; (ii) the ZIC instructions are fed into an optimizer to find a schedule which minimizes the time of their execution through pipelined VLIWs and avoids structural and data hazards (i.e. no device overbooking occurs and no data are used before their computation is terminated). We prove in lemmas 1,2 below that this pipelined instruction execution is correct. The reader should keep in mind that we illustrate the technique here only for arithmetical instructions but that a similar construction can be made for input–output and jump instructions in such a way that the correctness of their simultaneous execution together with arithmetical instructions (VLIW scheme) can be proved rigorously. Due to size restrictions, we have to leave the details for a sequel to this paper.

Proposition. The (extended) arithmetic subunit executes additive arithmetic instructions correctly.[7]

[7]By "extended" we want to point to the fact that minor additional components have to be introduced to take care of condition code for zero, overflow etc. A similar proposition can be proved for the extension of the unit for

Proof. Due to space limitations we can only illustrate the proof technique by going through the major steps for an example; we choose the addition instruction ([`ADD` *RR R1 R2*]), whose execution is defined by the rule

$$\textbf{if } MAC = \texttt{ADD} \textbf{ then } reg(RR) := reg(R1) + reg(R2).$$

This rule is part of the evolving algebra ZIC defined in [13], which models APE100 as interpreter of ZCPU Intermediate Code (and which can easily be shown to implement correctly the APESE ground model for APE100 presented in [7]). Its execution presents the three usual phases (we skip the condition codes, which are computed in the second phase):

$$\begin{array}{rcl} \text{read\_operands} & \equiv & OutPort_1 := reg(R1) \\ & & OutPort_2 := reg(R2) \\ \text{calc\_result} & \equiv & \textbf{if } MAC = \texttt{ADD} \textbf{ then } Res := OutPort_1 + OutPort_2 \\ \text{write\_result} & \equiv & \textbf{if } MAC = \texttt{ADD} \textbf{ then } reg(RR) := Res \end{array}$$

In the general case, Res will take $math\_res(MAC, md\_ctrl, OutPort_1, OutPort_2, carry_{\text{in}}, extend_{\text{in}})$ (which for MAC = `ADD` takes the value $OutPort_1 + OutPort_2$, independently of the value of md_ctrl, $carry_{\text{in}}$ and $extend_{\text{in}}$).

This leads to an intermediate model ZIC_{PH} where the execution of ZIC-instructions is decomposed into three (or more, for some other instructions) sequential steps, such that the following intuitive statement can be made precise and proved by induction on the length of ZIC-computations.

Lemma 1. The evolving algebra ZIC_{PH} implements the model ZIC correctly.

The second step in the proof of the proposition introduces the parallelism due to pipelining. The rules of ZIC_{PH} are modified in such a way that no conflict arises when they are executed not any more sequentially, but in parallel (using the lock-step semantics of evolving algebras, see [14]), leading to the model LEX ("loadable executable code", the final level in the APE100 compilation chain). A rigorous formulation and a transparent proof can be given for the following statement.

Lemma 2. Under the assumption that the compiler guarantees data and control independence of instructions which enter the pipe, the model LEX implements the model ZIC_{PH} correctly.

Note that the optimizer mentioned above makes sure that the assumption of the lemma is guaranteed. The two preceding lemmas prove the proposition.

Proof. Due to the modular character of our formalization, the proof can be given instructionwise; essentially it consists in a local analysis of the effect of the LEX rules for each type of ZIC instruction. Again we illustrate the proof principle for the case of the `ADD` instruction.

The phases (the sequential steps) of the execution of instructions in ZIC_{PH} become pipe stages in LEX, where at each step all the rules are applied simultaneously. The main problem is to make sure that each pipe stage operates on the correct data.

When for the given instruction (`ADD`) it comes to execute calc_result (in the second pipe stage), the value of MAC in the preceding stage is needed; therefore MAC is delayed by $MAC_2 := MAC$ and replaced in *math_res* by MAC_2.

multiplications and divisions.

Similarly, in the third pipe stage, for writing the result to the destination register the three values for the RF-entries $write\_enable_4$, $addr_4$ and in_port_4 are needed. The value of $write\_enable_4$ must be 1 if MAC = `ADD`; as this value is computed by the DECODE unit in the first pipe stage, it must be delayed twice (through registers WER and WER_2). Similarly, the address of the destination register RR is read in the first pipe stage and needed in the third one (at the RF-entry $addr_4$); therefore, it is delayed twice through the registers RR_2 and RR_3.[8] Finally, the result to be written has been computed by calc_result (and placed in the Res register) in the previous pipe stage; therefore, no delay registers are needed for it, and the rule write_result of the LEX model is obtained by the write_result rule of the $\mathrm{ZIC_{PH}}$ model, by replacing MAC = `ADD` by WER_2 and RR by RR_3.

Conclusion

We have shown on an example how to prove that the hardware level model LEX of the APE100 control processor zCPU executes compiled programs correctly if the compiler satisfies the assumptions which are realized by the APE100 compilation chain. Along these lines [13] has developed the specification of the zCPU processor which is the basis for the complete correctness proof, which we have to leave for a sequel to this paper. In the appendix we list the complete formal definition of the zCPU, without further explanation.

Note that from the software point of view zCPU and the MAD chip—the floating point unit of the APE100 processor—can be seen as different elements of just one VLIW system. In terms of our specification method this means that zCPU and MAD can be seen as modules of APE100 whose architectural composition and dynamic interaction can be fully described by our methodology in a precise yet simple way. Note also that the case of zCPU and of APE100 reverse engineering is not restrictive. Our methodology can be applied to other processors as well. Se for ex. [10] where the evolving algebra methodology is used to prove the correctness of general pipelining principles for RISC architectures.

Acknowledgement

We are particularly grateful to Raffaele Tripiccione from the APE group in Pisa for numerous hours spent during the last two years for explaining to us the secrets of the APE100 project.

[8] As has been noted above, the zCPU has not two but only one delay register for RR. Therefore, the value of RR has to be taken during the second pipe stage — this is assured in APE100 by the compiler. In [13] the model is developed with all the details which are needed to describe exactly the behaviour of the zCPU on the basis of the APE100 compilation scheme.

Appendices

In the following four appendices we list with only a few more explanations the full abstract definition of the zCPU processor. Appendix A defines each single unit separately. Appendix B contains the block diagram of the zCPU, giving a rough representation of the units described in Appendix A and their connections. Appendix C defines how the units are composed, by producing formal identifications among exits and entries (corresponding to the connections shown in the block diagram). Appendix D lists the final definition of the zCPU after all the substitution are done.

A Definition of the zCPU units

Before listing the definitions of the units, we define the two main sets of operation codes for the instructions which can be executed by the zCPU, i.e. MAC-type instructions (arithmetic) and IOC-type instructions (input/output, jumps). The MAC-type operation codes are:

$$MATH\_OP \quad = \quad ALU\_OP \cup MUL\_OP \cup DIV\_OP \cup \{\mathtt{NOP}\}$$

where

$$\begin{aligned}
ALU_OP &= LOGIC \cup ADDER \cup SHIFTER \\
LOGIC &= \{\, \mathtt{OR}, \mathtt{AND}, \mathtt{NOR}, \mathtt{NAND}, \mathtt{XOR}, \mathtt{ZERO}, \mathtt{XNOR}, \mathtt{FF} \,\} \\
ADDER &= \{\, \mathtt{ADD}, \mathtt{ADDC}, \mathtt{SUB}, \mathtt{SUBC}, \mathtt{CMP} \,\} \\
SHIFTER &= \{\, \mathtt{LSH}, \mathtt{ASH}, \mathtt{XSH}, \mathtt{ROT} \,\} \\
MUL_OP &= \{\, \mathtt{MUL1}, \mathtt{MUL2}, \mathtt{MULA2}, \mathtt{MUL3} \,\} \\
DIV_OP &= \{\, \mathtt{DIV1}, \mathtt{DIV2} \,\}
\end{aligned}$$

The IOC-type operation codes are:

$$IOC\_OP \quad = \quad LOAD \cup STORE \cup BRANCH \cup SPECIAL \cup \{\mathtt{IONOP}\}$$

where

$$\begin{aligned}
LOAD &= \{\, \mathtt{LD}, \mathtt{LDA}, \mathtt{LDPA}, \mathtt{LDSR} \,\} \\
STORE &= \{\, \mathtt{ST}, \mathtt{STSR} \,\} \\
BRANCH &= \{\, \mathtt{JUMP}, \mathtt{SKIP} \,\} \\
SPECIAL &= \{\, \mathtt{FLD}, \mathtt{FST}, \mathtt{HALT}, \mathtt{BREAK}, \mathtt{SETF}, \mathtt{CLEARF} \,\}
\end{aligned}$$

A.1 Unit RF (Register File)

Entries: $addr_1$; $addr_2$; $addr_3$; $addr_4$, in_port_4, $write\_enable_4$; $addr_5$, in_port_5, $write\_enable_5$

Exits: $OutPort_1$, $OutPort_2$, $OutPort_3$, $OutPort_5$

Integrity constraints:

$$write\_enable_4 \Rightarrow addr_4 \notin \{\, addr_1, addr_2, addr_3, addr_5 \,\}$$
$$write\_enable_5 \Rightarrow addr_5 \notin \{\, addr_1, addr_2, addr_3 \,\}$$
$$write\_enable_4 \wedge write\_enable_5 \Rightarrow addr_4 \neq addr_5$$

Transition rules:

$\text{read\_reg}_{1,2} \equiv$
$\equiv$ $OutPort_1 := reg(addr_1)$
$OutPort_2 := reg(addr_2)$

$\text{read\_reg}_3 \equiv$
$\equiv$ $OutPort_3 := reg(addr_3)$

$\text{write\_reg}_4 \equiv$
$\equiv$ **if** $write\_enable_4$
then $reg(addr_4) := in\_port_4$

$\text{rw\_reg}_5 \equiv$
$\equiv$ **if** $write\_enable_5$
then $reg(addr_5) := in\_port_5$
$OutPort_5 := undef$
else $OutPort_5 := reg(addr_5)$

Note that port 5 of the RF can be used for reading and writing.

A.2 Unit ALU (ALU/Multiplier/Divider)

Entries: $math\_code$; op_1, op_2; $carry_{\text{in}}, extend_{\text{in}}, zero_{\text{in}}$; md_ctrl

Exits: $math\_res_{\text{out}}, carry_{\text{out}}, divz_{\text{out}}, neg_{\text{out}}, overflow_{\text{out}}, extend_{\text{out}}, zero_{\text{out}}$

Abbreviations:

$md\_ctrl \equiv mul\_in, start\_div, md\_mux, add\_mul$

Note that *md_ctrl* stands for "multiplier/divider control".

Definitions:

$$\begin{aligned}
math_res_{\text{out}} &= math_res(math_code, md_ctrl, op_1, op_2, carry_{\text{in}}, extend_{\text{in}}) \\
carry_{\text{out}} &= carry(math_code, op_1, op_2, carry_{\text{in}}) \\
divz_{\text{out}} &= divz(op_2) \\
neg_{\text{out}} &= neg(math_code, md_ctrl, op_1, op_2, carry_{\text{in}}, extend_{\text{in}}) \\
overflow_{\text{out}} &= overflow(math_code, op_1, op_2, carry_{\text{in}}) \\
extend_{\text{out}} &= extend(op_1, op_2) \\
zero_{\text{out}} &= zero(math_code, md_ctrl, op_1, op_2, carry_{\text{in}}, extend_{\text{in}}, zero_{\text{in}})
\end{aligned}$$

where

$math\_res(math\_code, md\_ctrl, op_1, op_2, carry_{\text{in}}, extend_{\text{in}}) =$
if $math\_code \notin MUL\_OP \cup DIV\_OP$
then $alu\_res(math\_code, op_1, op_2, carry_{\text{in}}, extend_{\text{in}})$
else $mul\_div\_res(md\_ctrl)$

$alu\_res(math\_code, op_1, op_2, carry_{\text{in}}, extend_{\text{in}}) =$
if $math\_code = \texttt{ADD}$ then $add(op_1, op_2)$
else if $math\_code = \texttt{ADDC}$ then $addc(op_1, op_2, carry_{\text{in}})$
else ...

$$
\begin{array}{l}
\mathit{mul\_div\_res}(\mathit{md\_ctrl}) \; = \\
\quad \text{if } \mathit{md\_mux} \in \{0, 1\} \\
\quad \text{then } \; \text{if } \mathit{mul\_ready}(\mathit{mul\_in}) \\
\qquad\quad \text{then } \mathit{mul\_res}(\mathit{md\_mux}, \mathit{add\_mul}, \mathit{MulOp1}, \mathit{MulOp2}) \\
\qquad\quad \text{else } \mathit{undef} \\
\quad \text{else } \; \text{if } \mathit{div\_ready}(\mathit{div\_start}) \vee \mathit{mod\_ready}(\mathit{div\_start}) \\
\qquad\quad \text{then } \mathit{div\_res}(\mathit{DivOp1}, \mathit{DivOp2}) \\
\qquad\quad \text{else } \mathit{undef}
\end{array}
$$

$$
\begin{array}{l}
\mathit{mul\_ready}(\mathit{mul\_in}) \; = \; (\mathit{mul\_in} \neq 0 \wedge \mathit{MulOp1} \neq \mathit{undef} \wedge \mathit{MulOp2} \neq \mathit{undef} \wedge \mathit{MulStep} \geq 2) \\
\mathit{div\_ready}(\mathit{div\_start}) \; = \; (\mathit{start\_div} \neq 1 \wedge \mathit{DivOp1} \neq \mathit{undef} \wedge \mathit{DivOp2} \neq \mathit{undef} \wedge \mathit{DivStep} = 34) \\
\mathit{mod\_ready}(\mathit{div\_start}) \; = \; (\mathit{start\_div} \neq 1 \wedge \mathit{DivOp1} \neq \mathit{undef} \wedge \mathit{DivOp2} \neq \mathit{undef} \wedge \mathit{DivStep} = 35)
\end{array}
$$

Note that, in the zCPU, computing multiplications needs two clock cycles; for the division, the ALU makes the quotient available after 34 clock cycles, the remainder of the division after 35 clock cycles. The definitions above are needed to express this behaviour of the ALU (see also the transition rules below and the related remark).

$$
\begin{array}{l}
\mathit{mul\_res}(\mathit{md\_mux}, \mathit{add\_mul}, x, y) \; = \\
\quad \text{if } \mathit{md\_mux} = 0 \wedge \mathit{add\_mul} = 0 \text{ then } \mathit{mull}(x, y) \\
\quad \text{else if } \mathit{md\_mux} = 0 \wedge \mathit{add\_mul} = 1 \text{ then } \mathit{mula}(x, y) \\
\quad \text{else } \mathit{mulm}(x, y)
\end{array}
$$

$$
\begin{array}{l}
\mathit{div\_res}(x, y) \; = \\
\quad \text{if } \mathit{div\_ready}(\mathit{div\_start}) \text{ then } \mathit{div}(x, y) \\
\quad \text{else if } \mathit{mod\_ready}(\mathit{div\_start}) \text{ then } \mathit{mod}(x, y) \\
\quad \text{else } \mathit{undef}
\end{array}
$$

$$\mathit{divz}(\mathit{op}_2) \; = \; (\mathit{op}_2 = 0)$$

$$\mathit{neg}(\mathit{math\_code}, \mathit{md\_ctrl}, \mathit{op}_1, \mathit{op}_2, \mathit{carry}_{\text{in}}, \mathit{extend}_{\text{in}}) \; = \; (\mathit{math\_res}_{\text{out}} < 0)$$

$$
\begin{array}{l}
\mathit{zero}(\mathit{math\_code}, \mathit{md\_ctrl}, \mathit{op}_1, \mathit{op}_2, \mathit{carry}_{\text{in}}, \mathit{extend}_{\text{in}}, \mathit{zero}_{\text{in}}) \; = \\
\quad \text{if } \mathit{math\_code} \; \notin \{ \mathtt{ADDC}, \mathtt{SUBC}, \mathtt{MUL3} \} \\
\quad \text{then } \mathit{math\_res}_{\text{out}} = 0 \\
\quad \text{else } (\mathit{math\_res}_{\text{out}} = 0) \wedge \mathit{zero}_{\text{in}}
\end{array}
$$

Transition rules (for multiplier and divider subunits):

$$
\begin{array}{l}
\text{mul\_rule} \; \equiv \\
\quad \equiv \; \textbf{if} \;\; \mathit{mul\_in} = 0 \\
\qquad \textbf{then} \;\; \mathit{MulOp1} := \mathit{op}_1 \\
\qquad\qquad\quad \mathit{MulOp2} := \mathit{op}_2 \\
\qquad\qquad\quad \mathit{MulStep} := 1 \\
\qquad \textbf{else} \;\; \mathit{MulStep} := \mathit{MulStep} + 1
\end{array}
$$

div_rule $\equiv$

$$
\begin{array}{l}
\equiv \ \textbf{if} \ \ \mathit{start\_div} = 1 \\
\quad \textbf{then} \ \ \mathit{DivOp1} := op_1 \\
\quad \qquad \quad \mathit{DivOp2} := op_2 \\
\quad \qquad \quad \mathit{DivStep} := 1 \\
\quad \textbf{else} \ \ \mathit{DivStep} := \mathit{DivStep} + 1
\end{array}
$$

Note that the way multiplications and divisions are computed requires two different actions: first, the multiplier/divider must be activated (by *mul_in/start_div*) in order to begin computing the operation; second, the result must be stored (when it is ready). In the meantime a new multiplication/division should not be started. As in the zCPU we have different operation codes for starting and ending multiplications/divisions, it is responsibility of the compiler to schedule instructions in such a way that the ALU is always provided with the correct values of *mul_in/start_div*. Note that, except for the rules which formalize the interaction between the main ALU and the multiplier and divider subunits, all the ALU functions are specified by pure equations without using any transition rules.

A.3 Unit CC&STATUS (Condition Codes & Status Register)

Entries: $\mathit{math\_cc}$, $\mathit{if\_status}_{\mathrm{in}}$, $\mathit{halt}_{\mathrm{in}}$, $\mathit{parity\_err}_{\mathrm{in}}$; $\mathit{enable\_cc}$; $\mathit{write\_mask}$, val

Exits: $\mathit{zcpu\_ex}$, $\mathit{status\_register}$

Abbreviations:

$$
\begin{array}{l}
\mathit{math\_cc} \ \equiv \ \mathit{carry}_{\mathrm{in}}, \mathit{divz}_{\mathrm{in}}, \mathit{neg}_{\mathrm{in}}, \mathit{overflow}_{\mathrm{in}}, \mathit{extend}_{\mathrm{in}}, \mathit{zero}_{\mathrm{in}} \\
\mathit{enable\_cc} \ \equiv \ \mathit{en\_carry}, \mathit{en\_divz}, \mathit{en\_neg}, \mathit{en\_overflow}, \mathit{en\_extend}, \mathit{en\_zero}
\end{array}
$$

Only certain instructions require that the condition code registers are set: therefore writing into those registers is governed by enabling bits ($\mathit{enable\_cc}$). See below the "write_cc_math" rule.

$$
\begin{array}{rl}
\mathrm{update\_mask}(\mathit{val}) \ \equiv & \mathit{Msk\_Carry} := \mathit{val}[3] \\
& \mathit{Msk\_Divz} := \mathit{val}[7] \\
& \mathit{Msk\_iFstatus} := \mathit{val}[11] \\
& \mathit{Msk\_Neg} := \mathit{val}[15] \\
& \mathit{Msk\_oVerflow} := \mathit{val}[19] \\
& \mathit{Msk\_eXtend} := \mathit{val}[23] \\
& \mathit{Msk\_Zero} := \mathit{val}[27]
\end{array}
$$

Definitions:

$$
\begin{array}{rl}
\mathit{status\_register} \ = & \mathit{Ex\_Parity} \,.\, \mathit{ParityErr} \,.\, \mathit{Ex\_Halt} \,.\, \mathit{Halt} \\
& .\, \mathit{Msk\_Zero} \,.\, \mathit{Ex\_Zero} \,.\, \mathit{Zero} \,.\, 0 \\
& .\, \mathit{Msk\_eXtend} \,.\, \mathit{Ex\_eXtend} \,.\, \mathit{eXtend} \,.\, 0 \\
& .\, \mathit{Msk\_oVerflow} \,.\, \mathit{Ex\_oVerflow} \,.\, \mathit{oVerflow} \,.\, 0 \\
& .\, \mathit{Msk\_Neg} \,.\, \mathit{Ex\_Neg} \,.\, \mathit{Neg} \,.\, 0 \\
& .\, \mathit{Msk\_iFstatus} \,.\, \mathit{Ex\_iFstatus} \,.\, \mathit{iFstatus} \,.\, 0 \\
& .\, \mathit{Msk\_Divz} \,.\, \mathit{Ex\_Divz} \,.\, \mathit{Divz} \,.\, 0 \\
& .\, \mathit{Msk\_Carry} \,.\, \mathit{Ex\_Carry} \,.\, \mathit{Carry} \,.\, \mathit{ExcpReg}
\end{array}
$$

The status register contains in one machine word the relevant information about the condition codes and exceptions ("status information"): it is accessible through the instructions `LDSR` and `STSR`.

$$
\begin{aligned}
zcpu\_ex \;=\; & Halt \vee ParityErr \vee (iFstatus \wedge Msk\_iFstatus) \vee \\
& \vee (Carry \wedge Msk\_Carry) \vee (Divz \wedge Msk\_Divz) \vee \\
& \vee (Neg \wedge Msk\_Neg) \vee (oVerflow \wedge Msk\_oVerflow) \vee \\
& \vee (eXtend \wedge Msk\_eXtend) \vee (Zero \wedge Msk\_Zero)
\end{aligned}
$$

Note that we summarize the status information in the definition of the exit *status_register*: its dynamic change is reflected by the local updates of its components through the transition rules given below. Similarly, the exit *zcpu_ex* — used for detecting conditions leading to exceptions — depends on the value of registers contained in the CC&Status unit.

Transition rules:

write_cc_math $\equiv$
$\equiv$ **if** *en_carry* **then** $Carry := carry_{\text{in}}$
if *en_divz* **then** $Divz := divz_{\text{in}}$
if *en_neg* **then** $Neg := neg_{\text{in}}$
if *en_overflow* **then** $oVerflow := overflow_{\text{in}}$
if *en_extend* **then** $eXtend := extend_{\text{in}}$
if *en_zero* **then** $Zero := zero_{\text{in}}$

write_cc_ifstatus $\equiv$
$\equiv$ $iFstatus := if\_status_{\text{in}}$

Note that, in APE100, the value of if_status_{in} bit depends on the state of the floating point units of APE100.

write_cc_halt $\equiv$
$\equiv$ $Halt := halt_{\text{in}}$

write_cc_parity $\equiv$
$\equiv$ $ParityErr := parity\_err_{\text{in}}$

write_exception_mask(*write_mask*, *val*) $\equiv$
$\equiv$ **if** *write_mask*
then update_mask(*val*)

exception_rule $\equiv$
$\equiv$ **if** $\neg ExcpReg$
then $ExcpReg := zcpu\_ex$
$Ex\_Halt := Halt$
$Ex\_Parity := ParityErr$
$Ex\_iFstatus := iFstatus$
$Ex\_Carry := Carry$
$Ex\_Divz := Divz$
$Ex\_Neg := Neg$
$Ex\_oVerflow := oVerflow$
$Ex\_eXtend := eXtend$
$Ex\_Zero := Zero$

A.4 Unit AGU (Address Generation Unit)

Entries: abs_{in}, pma_{in}, *base*, *disp*, pma_mux_{in}

Exits: $dma_{\text{out}}, pma_{\text{out}}$

Definitions:

$$
\begin{aligned}
dma_{\text{out}} &= gen\_addr(abs_{\text{in}}, pma_{\text{in}}, base, disp) \\
pma_{\text{out}} &= next\_pma(abs_{\text{in}}, pma_{\text{in}}, base, disp, pma\_mux_{\text{in}})
\end{aligned}
$$

where

$$
\begin{aligned}
& gen\_addr(abs_{\text{in}}, pma_{\text{in}}, base, disp) = \\
& \quad (1 - abs_{\text{in}})pma_{\text{in}} + base + disp \\[1ex]
& next\_pma(abs_{\text{in}}, pma_{\text{in}}, base, disp, pma\_mux_{\text{in}}) = \\
& \quad \text{if } pma\_mux_{\text{in}} = 0 \text{ then } 0 \\
& \quad \text{else if } pma\_mux_{\text{in}} = 1 \text{ then } pma_{\text{in}} + 1 \\
& \quad \text{else } gen\_addr(abs_{\text{in}}, pma_{\text{in}}, base, disp)
\end{aligned}
$$

Note that upon insertion of the AGU unit into the zCPU processor the exits dma_{out} and pma_{out} will be written into the registers *DMA* and *PMA* resp. (see A.11 and C.11 for further explanation.)

A.5 Unit PROGMEM (Program Memory)

Entries: *addr*

Exits: pmd_{out}

Definitions:

$$pmd_{\text{out}} = instr(addr)$$

Note that *addr* will be provided by the Address Generation Unit (AGU) and pmd_{out} will be read by the Instruction Fetch unit (INSTR). See C.5 and C.7 below.

A.6 Unit DATAMEM (Data Memory)

Entries: $addr, data_{\text{in}}, mem\_write\_enable$

Exits: $data_{\text{out}}$

Definitions:

$$data_{\text{out}} = mem(addr)$$

Transition rules:

$$
\begin{aligned}
& \text{write\_mem} \equiv \\
& \quad \equiv \textbf{if} \ \ mem\_write\_enable \\
& \qquad \textbf{then} \ \ mem(addr) := data_{\text{in}}
\end{aligned}
$$

Note that $data_{\text{in}}$ will be read from the Input/Output Subsystem (IOS), namely from its *val_to_store* exit (see C.6).

A.7 Unit INSTR (Instruction Fetch)

Entries: pmd_{in}

Exits: *MAC*, *RR*, *R1*, *R2*, *IOC*, *RD*, *RA*, *Disp*

Definitions:

$$
\begin{array}{lcl}
MAC & = & MAC(PMD) \\
RR & = & RR(PMD) \\
R1 & = & R1(PMD) \\
R2 & = & R2(PMD) \\
IOC & = & IOC(PMD) \\
RD & = & RD(PMD) \\
RA & = & RA(PMD) \\
Disp & = & Disp(PMD)
\end{array}
$$

Transition rules:

instruction_fetch $\equiv$
$\equiv \quad PMD := pmd_{\text{in}}$

A.8 Unit IOS (Input/Output Subsystem)

Entries: st_data, val_from_RF, $status\_register$;
ld_addr, dma_{in}, val_from_mem

Exits: Dat, val_to_load, val_to_store

Note that st_data, ld_addr, dma_{in} and val_from_mem take their values from the registers $StData$, $LdAddr_2$, DMA and from the exit $data_{\text{out}}$ of unit DATAMEM respectively (see C.8). Note also that for notational simplicity we declare the register Dat to be also an IOS exit.

Definitions:

$$
\begin{array}{lcl}
val\_to\_store & = & \text{if } st\_data \text{ then } val\_from\_RF \text{ else } status\_register \\
val\_to\_load & = & \text{if } ld\_addr \text{ then } DMA_2 \text{ else } Dat
\end{array}
$$

Transition rules:

delay_DMA $\equiv$
$\equiv \quad DMA_2 := dma_{\text{in}}$

Note that this update of DMA_2 is in accordance with the view of DMA_2 as delay register for DMA; namely in C.8 dma_{in} will be replaced by DMA.

read_mem $\equiv$
$\equiv \quad Dat := val\_from\_mem$

A.9 Unit EVALCOND (Evaluation of Branch Conditions)

Entries: $which\_cond$; $carry_{\text{in}}$, if_status_{in}, neg_{in}, $overflow_{\text{in}}$, $zero_{\text{in}}$

Exits: yes_no

Definitions:

$$yes\_no \;=\; eval\_cond(which\_cond, carry_{\text{in}}, if\_status_{\text{in}}, neg_{\text{in}}, overflow_{\text{in}}, zero_{\text{in}})$$

Note that yes_no will be connected to the corresponding DECODE entry (see C.10).

A.10 Unit DECODE (Instruction Decoding)

Entries: $math\_code$, io_code, yes_no

Exits: pma_mux_{out}, abs_{out}, $halt_{\text{out}}$; MAC_2, WER, $EnableCC$, $MDCtrl$;
$MemWE$, $StData$, WED, $LdAddr$, $WrMask$, $Cancel$

Abbreviations:

$$EnableCC \equiv En\_Carry, En\_Divz, En\_Neg, En\_oVerflow, En\_eXtend, En\_Zero$$
$$MDCtrl \equiv MulIn, StartDiv, MDMux, AddMul$$

Definitions:

$$abs_{\text{out}} = IOC \notin \{\,\mathtt{LDPA}, \mathtt{SKIP}\,\}$$
$$pma\_mux_{\text{out}} = \text{if } (io\_code \in \{\,\mathtt{JUMP}, \mathtt{SKIP}\,\} \wedge yes\_no \ \wedge \ \neg Cancel) \text{ then } 2 \text{ else } 1$$
$$halt_{\text{out}} = (io\_code = \mathtt{HALT}) \wedge \neg Cancel$$

Transition rules:

$$\begin{aligned} &\text{MAC\_decode} \equiv \\ &\equiv \ MAC_2 := math\_code \\ &\quad\ WER := (math\_code \in MATH\_OP \setminus \{\mathtt{CMP}, \mathtt{MUL1}, \mathtt{DIV1}\}) \wedge \neg Cancel \end{aligned}$$

Note that the update of MAC_2 is in accordance with the view of MAC_2 as delay of MAC; indeed $math\_code$ is a DECODE entry which in C.10 will be replaced by MAC.

$$\begin{aligned} &\text{decode\_EnableCC} \equiv \\ &\equiv \ En\_Carry := (math\_code \in LOGIC \cup ADDER) \wedge \neg Cancel \\ &\quad\ En\_Divz := (math\_code = \mathtt{DIV1}) \wedge \neg Cancel \\ &\quad\ En\_Neg := (math\_code \in MATH\_OP \setminus \{\mathtt{MUL1}, \mathtt{DIV1}\}) \wedge \neg Cancel \\ &\quad\ En\_oVerflow := (math\_code \in LOGIC \cup ADDER \cup \{\mathtt{ASH}\}) \wedge \neg Cancel \\ &\quad\ En\_eXtend := (math\_code \in SHIFTER) \wedge \neg Cancel \\ &\quad\ En\_Zero := (math\_code \in MATH\_OP \setminus \{\mathtt{MUL1}, \mathtt{DIV1}\}) \wedge \neg Cancel \end{aligned}$$

$$\begin{aligned} &\text{decode\_MDCtrl} \equiv \\ &\equiv \ MulIn := (math\_code \neq \mathtt{MUL1}) \vee Cancel \\ &\quad\ StartDiv := (math\_code = \mathtt{DIV1}) \wedge \neg Cancel \\ &\quad\ MDMux := (\text{ if } (math\_code = \mathtt{DIV2}) \text{ then } 2 \\ &\qquad\qquad\qquad \text{else if } (math\_code = \mathtt{MUL3}) \text{ then } 1 \\ &\qquad\qquad\qquad \text{else } 0\,)\,(1 - Cancel) \\ &\quad\ AddMul := (math\_code = \mathtt{MULA2}) \wedge \neg Cancel \end{aligned}$$

$$\begin{aligned} &\text{IOC\_decode} \equiv \\ &\equiv \ MemWE := (io\_code \in \{\,\mathtt{ST}, \mathtt{STSR}\,\}) \wedge \neg Cancel \\ &\quad\ StData := (io\_code = \mathtt{ST}) \wedge \neg Cancel \\ &\quad\ WED := (io\_code \in \{\,\mathtt{LD}, \mathtt{LDA}, \mathtt{LDPA}\,\}) \wedge \neg Cancel \\ &\quad\ LdAddr := (io\_code \in \{\,\mathtt{LDA}, \mathtt{LDPA}\,\}) \wedge \neg Cancel \\ &\quad\ WrMask := (io\_code = \mathtt{LDSR}) \wedge \neg Cancel \\ &\quad\ Cancel := (io\_code \in \{\,\mathtt{JUMP}, \mathtt{SKIP}\,\} \wedge yes\_no \ \wedge \ \neg Cancel) \end{aligned}$$

A.11 Registers

In order to implement pipelining, we need also to introduce some delay registers. As all the registers behave in the same way, we can define the behaviour of a "register unit" by means of the following schema (describing a generic register X)[9]:

Entries: $X.in$

Exits: $X.out$

Definitions:

$X.out = X$

Transition rules:

write_register_X $\equiv$
$\equiv \quad X := X.in$

(Recall that, for brevity, we write simply X in place of $X.out$ within terms and transition rules).

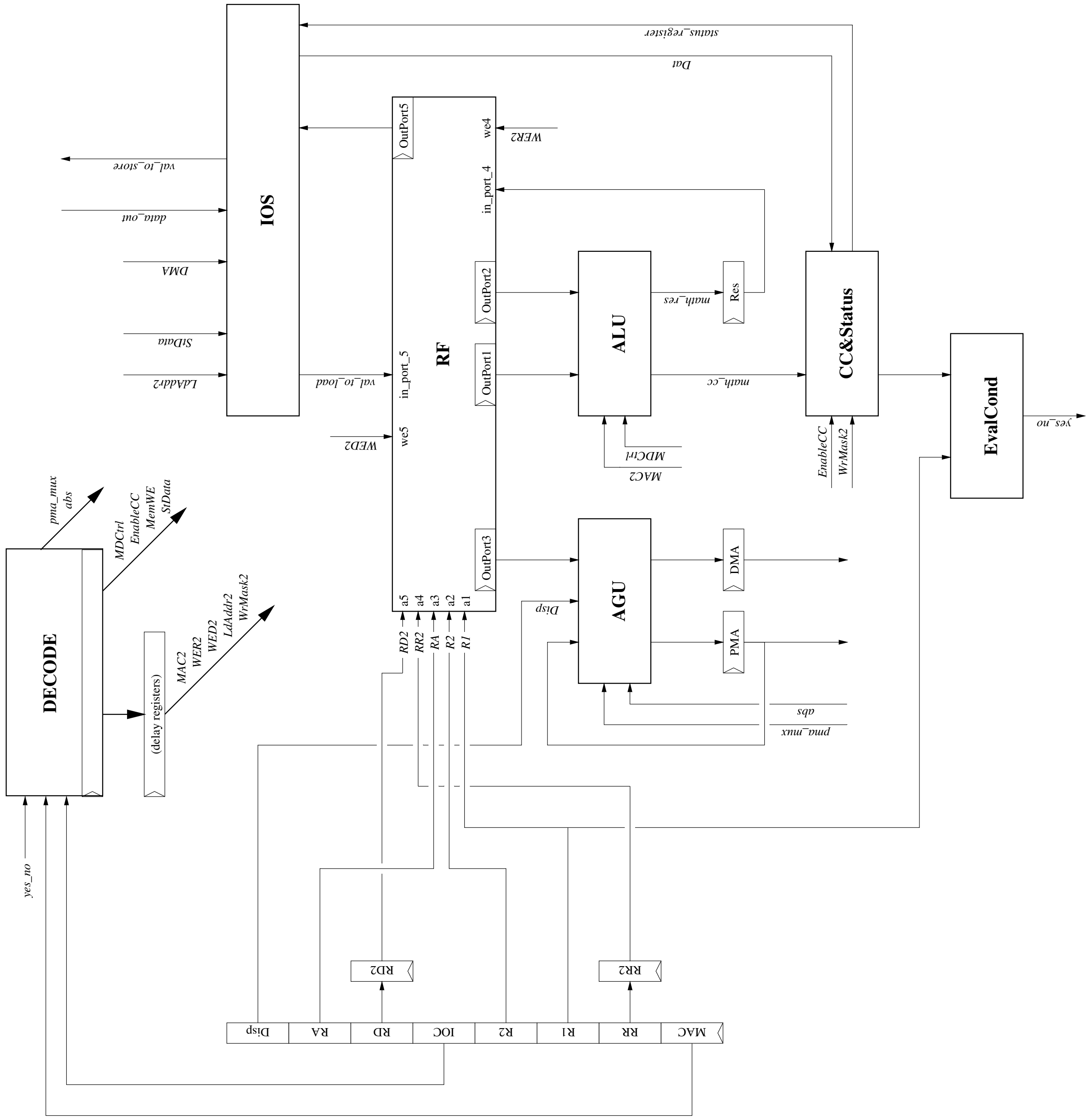
In the zCPU we have the following delay registers: $WER_2, RR_2, WED_2, RD_2, LdAddr_2, WrMask_2$. We have also three other registers in the zCPU:

- *Res*, used as temporary storage for the ALU results;
- *PMA* (*program memory address*), the program counter;
- *DMA* (*data memory address*), containing the computed address needed for load/store operations.

[9]See also 1.3 for more details on internal registers.

B ZCPU Block Diagram

The following block diagram represents pictorially the zCPU at the level of its main building blocks. It is intended as an help to get an intuitive idea of what is formally defined in Appendices A, C and D, rather than as a complete reference (for example, the units DATAMEM and PROGMEM are not represented, as well as some other items which are instead included in the textual description).



C Composition of the zCPU units

This appendix contains the identifications of exits and entries, which are needed to compose the zCPU units described above, according to the block diagram shown in Appendix B. We proceed in the same order as in Appendix A, defining a substitution for the entries of each unit presented there.

In order to resolve possible ambiguities between names used within definitions of different units, as well as to ease the task of following the connection paths between the units, we adopt the notational convention of prefixing names of exits by the unit name (for example, the exit $carry_{\mathrm{out}}$ of the ALU unit is globally identified as $\mathrm{ALU}.carry_{\mathrm{out}}$).

For more clarity and consistency of notation, we also use similar prefixes before the name of registers defined in A.11: the convention here is that the prefix of the register name is the name of the unit which provides the value to be written in the register (see C.11).

However, each register has a unique name, which allows us to remove all the prefixes from register names in the final description of the zCPU (Appendix D).

C.1 Unit RF (Register File)

$$
\begin{array}{lcl}
addr_1 & = & \mathrm{INSTR}.R1 \\
addr_2 & = & \mathrm{INSTR}.R2 \\
addr_3 & = & \mathrm{INSTR}.RA \\
addr_4 & = & \mathrm{INSTR}.RR_2 \\
in\_port_4 & = & \mathrm{ALU}.Res \\
write\_enable_4 & = & \mathrm{DECODE}.WER_2 \\
addr_5 & = & \mathrm{INSTR}.RD_2 \\
in\_port_5 & = & \mathrm{IOS}.val\_to\_load \\
write\_enable_5 & = & \mathrm{DECODE}.WED_2
\end{array}
$$

C.2 Unit ALU (ALU/Multiply/Divide)

$$
\begin{array}{lcl}
math\_code & = & \mathrm{DECODE}.MAC_2 \\
op_1 & = & \mathrm{RF}.OutPort1 \\
op_2 & = & \mathrm{RF}.OutPort2 \\
carry_{\mathrm{in}} & = & \mathrm{CC\&STATUS}.carry_{\mathrm{out}} \\
extend_{\mathrm{in}} & = & \mathrm{CC\&STATUS}.extend_{\mathrm{out}} \\
zero_{\mathrm{in}} & = & \mathrm{CC\&STATUS}.zero_{\mathrm{out}} \\
mul\_in & = & \mathrm{DECODE}.MulIn \\
start\_div & = & \mathrm{DECODE}.StartDiv \\
md\_mux & = & \mathrm{DECODE}.MDMux \\
add\_mul & = & \mathrm{DECODE}.AddMul
\end{array}
$$

where

$$
\begin{array}{lcl}
\mathrm{CC\&STATUS}.carry_{\mathrm{out}} & \equiv & \mathrm{CC\&STATUS}.status\_register[1] \\
\mathrm{CC\&STATUS}.extend_{\mathrm{out}} & \equiv & \mathrm{CC\&STATUS}.status\_register[21] \\
\mathrm{CC\&STATUS}.zero_{\mathrm{out}} & \equiv & \mathrm{CC\&STATUS}.status\_register[25]
\end{array}
$$

C.3 Unit CC&STATUS (Condition Codes & Status Register)

$$
\begin{aligned}
carry_{\mathrm{in}} &= \mathrm{ALU}.carry_{\mathrm{out}} \\
divz_{\mathrm{in}} &= \mathrm{ALU}.divz_{\mathrm{out}} \\
neg_{\mathrm{in}} &= \mathrm{ALU}.neg_{\mathrm{out}} \\
overflow_{\mathrm{in}} &= \mathrm{ALU}.overflow_{\mathrm{out}} \\
extend_{\mathrm{in}} &= \mathrm{ALU}.extend_{\mathrm{out}} \\
zero_{\mathrm{in}} &= \mathrm{ALU}.zero_{\mathrm{out}} \\
if\_status_{\mathrm{in}} & \quad \text{[external function]} \\
halt_{\mathrm{in}} &= \mathrm{DECODE}.halt_{\mathrm{out}} \\
parity\_err_{\mathrm{in}} &= parity\_check(PMD) \\
en\_carry &= \mathrm{DECODE}.En\_Carry \\
en\_divz &= \mathrm{DECODE}.En\_Divz \\
en\_neg &= \mathrm{DECODE}.En\_Neg \\
en\_overflow &= \mathrm{DECODE}.En\_oVerflow \\
en\_extend &= \mathrm{DECODE}.En\_eXtend \\
en\_zero &= \mathrm{DECODE}.En\_Zero \\
write\_mask &= \mathrm{DECODE}.WrMask_2 \\
val &= \mathrm{IOS}.Dat
\end{aligned}
$$

C.4 Unit AGU (Address Generation Unit)

$$
\begin{aligned}
abs_{\mathrm{in}} &= \mathrm{DECODE}.abs_{\mathrm{out}} \\
pma_{\mathrm{in}} &= \mathrm{AGU}.PMA \\
base &= \mathrm{RF}.OutPort3 \\
disp &= \mathrm{INSTR}.Disp \\
pma\_mux_{\mathrm{in}} &= \mathrm{DECODE}.pma\_mux_{\mathrm{out}}
\end{aligned}
$$

C.5 Unit PROGMEM (Program Memory)

$$
addr = \mathrm{AGU}.PMA
$$

C.6 Unit DATAMEM (Data Memory)

$$
\begin{aligned}
addr &= \mathrm{AGU}.DMA \\
data_{\mathrm{in}} &= \mathrm{IOS}.val\_to\_store \\
mem\_write\_enable &= \mathrm{DECODE}.MemWE
\end{aligned}
$$

C.7 Unit INSTR (Instruction Fetch)

$$
pmd_{\mathrm{in}} = \mathrm{PROGMEM}.pmd_{\mathrm{out}}
$$

C.8 Unit IOS (Input/Output Subsystem)

$$\begin{aligned}
\mathit{st_data} &= \text{DECODE}.\mathit{StData} \\
\mathit{val_from_RF} &= \text{RF}.\mathit{OutPort5} \\
\mathit{status_register} &= \text{CC\&STATUS}.\mathit{status_register} \\
\mathit{ld_addr} &= \text{DECODE}.\mathit{LdAddr}_2 \\
\mathit{dma}_{\text{in}} &= \text{AGU}.\mathit{DMA} \\
\mathit{val_from_mem} &= \text{DATAMEM}.\mathit{data}_{\text{out}}
\end{aligned}$$

C.9 Unit EVALCOND (Evaluation of Branch Conditions)

$$\begin{aligned}
\mathit{which_cond} &= \text{INSTR}.\mathit{R1} \\
\mathit{carry}_{\text{in}} &= \text{CC\&STATUS}.\mathit{carry}_{\text{out}} \\
\mathit{if_status}_{\text{in}} &= \text{CC\&STATUS}.\mathit{if_status}_{\text{out}} \\
\mathit{neg}_{\text{in}} &= \text{CC\&STATUS}.\mathit{neg}_{\text{out}} \\
\mathit{overflow}_{\text{in}} &= \text{CC\&STATUS}.\mathit{overflow}_{\text{out}} \\
\mathit{zero}_{\text{in}} &= \text{CC\&STATUS}.\mathit{zero}_{\text{out}}
\end{aligned}$$

where

$$\begin{aligned}
\text{CC\&STATUS}.\mathit{carry}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status_register}[1] \\
\text{CC\&STATUS}.\mathit{if_status}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status_register}[9] \\
\text{CC\&STATUS}.\mathit{neg}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status_register}[13] \\
\text{CC\&STATUS}.\mathit{overflow}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status_register}[17] \\
\text{CC\&STATUS}.\mathit{zero}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status_register}[25]
\end{aligned}$$

C.10 Unit DECODE (Instruction Decoding)

$$\begin{aligned}
\mathit{math_code} &= \text{INSTR}.\mathit{MAC} \\
\mathit{io_code} &= \text{INSTR}.\mathit{IOC} \\
\mathit{yes_no} &= \text{EVALCOND}.\mathit{yes_no}
\end{aligned}$$

C.11 Registers

$$\begin{aligned}
\text{DECODE}.\mathit{WER}_2.\mathit{in} &= \text{DECODE}.\mathit{WER} \\
\text{INSTR}.\mathit{RR}_2.\mathit{in} &= \text{INSTR}.\mathit{RR} \\
\text{DECODE}.\mathit{WED}_2.\mathit{in} &= \text{DECODE}.\mathit{WED} \\
\text{INSTR}.\mathit{RD}_2.\mathit{in} &= \text{INSTR}.\mathit{RD} \\
\text{DECODE}.\mathit{LdAddr}_2.\mathit{in} &= \text{DECODE}.\mathit{LdAddr} \\
\text{DECODE}.\mathit{WrMask}_2.\mathit{in} &= \text{DECODE}.\mathit{WrMask} \\
\text{ALU}.\mathit{Res}.\mathit{in} &= \text{ALU}.\mathit{math_res}_{\text{out}} \\
\text{AGU}.\mathit{PMA}.\mathit{in} &= \text{AGU}.\mathit{pma}_{\text{out}} \\
\text{AGU}.\mathit{DMA}.\mathit{in} &= \text{AGU}.\mathit{dma}_{\text{out}}
\end{aligned}$$

D Full definition of the zCPU

This appendix contains the full definition of the zCPU processor. It is obtained by combining the units of Appendix A, performing the substitutions corresponding to the equations listed in Appendix C and to the equations defining the exits of the different units. For further explanations see [13]. Note that the zCPU itself can be seen as a unit, with its own entries and exits.

Unit zCPU

Entries: $\mathit{if\_status}_{\mathrm{in}}$

Exits: *zcpu_ex*

Abbreviations:

$$\begin{array}{rcl}
\mathit{absolute} & \equiv & \mathit{IOC} \notin \{\, \mathtt{LDPA}, \mathtt{SKIP} \,\} \\
\mathit{yes_no} & \equiv & \mathit{eval_cond}(\mathit{R1}, \mathit{Carry}, \mathit{iFstatus}, \mathit{Neg}, \mathit{oVerflow}, \mathit{Zero}) \\
\mathit{pma_mux} & \equiv & \text{if } (\mathit{IOC} \in \{\, \mathtt{JUMP}, \mathtt{SKIP} \,\} \wedge \mathit{yes_no} \ \wedge \ \neg \mathit{Cancel}) \\
 & & \text{then } 2 \\
 & & \text{else } 1
\end{array}$$

$$\begin{array}{rcl}
\mathit{val_to_store} & \equiv & \text{if } \mathit{StData} \text{ then } \mathit{OutPort}_5 \text{ else } \mathit{status_register} \\
\mathit{val_to_load} & \equiv & \text{if } \mathit{LdAddr}_2 \text{ then } \mathit{DMA}_2 \text{ else } \mathit{Dat}
\end{array}$$

$$\begin{array}{rcl}
\mathit{status_register} & \equiv & \mathit{Ex_Parity} \,.\, \mathit{ParityErr} \,.\, \mathit{Ex_Halt} \,.\, \mathit{Halt} \\
 & & .\, \mathit{Msk_Zero} \,.\, \mathit{Ex_Zero} \,.\, \mathit{Zero} \,.\, 0 \\
 & & .\, \mathit{Msk_eXtend} \,.\, \mathit{Ex_eXtend} \,.\, \mathit{eXtend} \,.\, 0 \\
 & & .\, \mathit{Msk_oVerflow} \,.\, \mathit{Ex_oVerflow} \,.\, \mathit{oVerflow} \,.\, 0 \\
 & & .\, \mathit{Msk_Neg} \,.\, \mathit{Ex_Neg} \,.\, \mathit{Neg} \,.\, 0 \\
 & & .\, \mathit{Msk_iFstatus} \,.\, \mathit{Ex_iFstatus} \,.\, \mathit{iFstatus} \,.\, 0 \\
 & & .\, \mathit{Msk_Divz} \,.\, \mathit{Ex_Divz} \,.\, \mathit{Divz} \,.\, 0 \\
 & & .\, \mathit{Msk_Carry} \,.\, \mathit{Ex_Carry} \,.\, \mathit{Carry} \,.\, \mathit{ExcpReg}
\end{array}$$

Integrity constraints:

$$\mathit{WER}_2 \Rightarrow \mathit{RR}_2 \notin \{\, \mathit{R1}, \mathit{R2}, \mathit{RA}, \mathit{RD}_2 \,\}$$
$$\mathit{WED}_2 \Rightarrow \mathit{RD}_2 \notin \{\, \mathit{R1}, \mathit{R2}, \mathit{RA} \,\}$$
$$\mathit{WER}_2 \wedge \mathit{WED}_2 \Rightarrow \mathit{RR}_2 \neq \mathit{RD}_2$$

Note that it is possible to ensure that these contraints are satisfied by imposing appropriate contraints on the compiler.

Definitions:

$$\begin{array}{rcl}
\mathit{zcpu_ex} & = & \mathit{Halt} \vee \mathit{ParityErr} \vee (\mathit{iFstatus} \wedge \mathit{Msk_iFstatus}) \vee \\
 & & \vee\, (\mathit{Carry} \wedge \mathit{Msk_Carry}) \vee (\mathit{Divz} \wedge \mathit{Msk_Divz}) \vee \\
 & & \vee\, (\mathit{Neg} \wedge \mathit{Msk_Neg}) \vee (\mathit{oVerflow} \wedge \mathit{Msk_oVerflow}) \vee \\
 & & \vee\, (\mathit{eXtend} \wedge \mathit{Msk_eXtend}) \vee (\mathit{Zero} \wedge \mathit{Msk_Zero})
\end{array}$$

Transition rules:

instruction_fetch $\equiv$
$\equiv$ $PMD := instr(PMA)$

MAC_decode $\equiv$
$\equiv$ $MAC_2 := MAC$
$WER := (MAC \in MATH\_OP \setminus \{\texttt{CMP}, \texttt{MUL1}, \texttt{DIV1}\}) \wedge \neg Cancel$

decode_EnableCC $\equiv$
$\equiv$ $En\_Carry := (MAC \in LOGIC \cup ADDER) \wedge \neg Cancel$
$En\_Divz := (MAC = \texttt{DIV1}) \wedge \neg Cancel$
$En\_Neg := (MAC \in MATH\_OP \setminus \{\texttt{MUL1}, \texttt{DIV1}\}) \wedge \neg Cancel$
$En\_oVerflow := (MAC \in LOGIC \cup ADDER \cup \{\texttt{ASH}\}) \wedge \neg Cancel$
$En\_eXtend := (MAC \in SHIFTER) \wedge \neg Cancel$
$En\_Zero := (MAC \in MATH\_OP \setminus \{\texttt{MUL1}, \texttt{DIV1}\}) \wedge \neg Cancel$

decode_MDCtrl $\equiv$
$\equiv$ $MulIn := (MAC \neq \texttt{MUL1}) \vee Cancel$
$StartDiv := (MAC = \texttt{DIV1}) \wedge \neg Cancel$
$MDMux := (\text{if } (MAC = \texttt{DIV2}) \text{ then } 2 \text{ else if } (MAC = \texttt{MUL3}) \text{ then } 1 \text{ else } 0)(1 - Cancel)$
$AddMul := (MAC = \texttt{MULA2}) \wedge \neg Cancel$

IOC_decode $\equiv$
$\equiv$ $MemWE := (IOC \in \{\texttt{ST}, \texttt{STSR}\}) \wedge \neg Cancel$
$StData := (IOC = \texttt{ST}) \wedge \neg Cancel$
$WED := (IOC \in \{\texttt{LD}, \texttt{LDA}, \texttt{LDPA}\}) \wedge \neg Cancel$
$LdAddr := (IOC \in \{\texttt{LDA}, \texttt{LDPA}\}) \wedge \neg Cancel$
$WrMask := (IOC = \texttt{LDSR}) \wedge \neg Cancel$
$Cancel := (IOC \in \{\texttt{JUMP}, \texttt{SKIP}\} \wedge yes\_no \ \wedge \ \neg Cancel)$

Note the condition on the right side of the *Cancel* update expresses that a jump is going to take place, which implies that the instruction word following the jump, which has already been fetched, must be canceled (i.e. decoded as a "no operation"). This works under the compiler condition that there are never two immediately successive jumps.

delay_WER_RR $\equiv$
$\equiv$ $WER_2 := WER$
$RR_2 := RR$

IOC_delay $\equiv$
$\equiv$ $WED_2 := WED$
$LdAddr_2 := LdAddr$
$WrMask_2 := WrMask$

delay_RD $\equiv$
$\equiv$ $RD_2 := RD$

read_operands $\equiv$
$\equiv$ $OutPort_1 := reg(R1)$
$OutPort_2 := reg(R2)$

read_addr_reg $\equiv$
$\equiv$ $OutPort_3 := reg(RA)$

read_data_reg $\equiv$
$\equiv$ $OutPort_5 := reg(RD_2)$

write_result $\equiv$
$\equiv$ **if** WER_2
then $reg(RR_2) := Res$

write_dest_reg $\equiv$
$\equiv$ **if** WED_2
then $reg(RD_2) := val\_to\_load$

calc_result $\equiv$
$\equiv$ $Res := math\_res(\ MAC_2, MulIn, StartDiv, MDMux, AddMul,$
$OutPort_1, OutPort_2, Carry, eXtend\ )$

mul_rule $\equiv$
$\equiv$ **if** $MulIn = 0$
then $MulOp1 := OutPort_1$
$MulOp2 := OutPort_2$
$MulStep := 1$
else $MulStep := MulStep + 1$

div_rule $\equiv$
$\equiv$ **if** $StartDiv = 1$
then $DivOp1 := OutPort_1$
$DivOp2 := OutPort_2$
$DivStep := 1$
else $DivStep := DivStep + 1$

write_cc_math $\equiv$
$\equiv$ **if** En_Carry
then $Carry\ := carry(MAC_2, OutPort_1, OutPort_2, Carry)$

if En_Divz
then $Divz\ := divz(OutPort_2)$

if En_Neg
then $Neg\ := neg(\ MAC_2, MulIn, StartDiv, MDMux, AddMul,$
$OutPort_1, OutPort_2, Carry, eXtend\ )$

if $En\_oVerflow$
then $oVerflow\ := overflow(MAC_2, OutPort_1, OutPort_2, Carry)$

if En_eXtend
then $eXtend\ := extend(OutPort_1, OutPort_2)$

if En_Zero
then $Zero\ := zero(\ MAC_2, MulIn, StartDiv, MDMux, AddMul,$
$OutPort_1, OutPort_2, Carry, eXtend, Zero\ )$

write_cc_ifstatus $\equiv$

$\equiv$ $iFstatus := if\_status_{\text{in}}$

write_cc_halt $\equiv$

$\equiv$ $Halt := (IOC = \texttt{HALT}) \wedge \neg Cancel$

write_cc_parity $\equiv$

$\equiv$ $ParityErr := parity\_check(PMD)$

exception_rule $\equiv$

$\equiv$ **if** $\neg ExcpReg$

then $ExcpReg := zcpu\_ex$

$Ex\_Halt := Halt$

$Ex\_Parity := ParityErr$

$Ex\_iFstatus := iFstatus$

$Ex\_Carry := Carry$

$Ex\_Divz := Divz$

$Ex\_Neg := Neg$

$Ex\_oVerflow := oVerflow$

$Ex\_eXtend := eXtend$

$Ex\_Zero := Zero$

calc_PMA $\equiv$

$\equiv$ $PMA :=$ if $pma\_mux = 0$ then 0

else if $pma\_mux = 1$ then $PMA + 1$

else $(1 - absolute)PMA + OutPort_3 + Disp$

calc_DMA $\equiv$

$\equiv$ $DMA := (1 - absolute)PMA + OutPort_3 + Disp$

delay_DMA $\equiv$

$\equiv$ $DMA_2 := DMA$

write_mem $\equiv$

$\equiv$ **if** $MemWE$

then $mem(DMA) := val\_to\_store$

read_mem $\equiv$

$\equiv$ $Dat := mem(DMA)$

write_mask $\equiv$

$\equiv$ **if** $WrMask_2$

then update_mask(val_to_load)

where update_mask(val) $\equiv$

$\equiv$ $Msk\_Carry := val[3]$

$Msk\_Divz := val[7]$

$Msk\_iFstatus := val[11]$

$Msk\_Neg := val[15]$

$Msk\_oVerflow := val[19]$

$Msk\_eXtend := val[23]$

$Msk\_Zero := val[27]$

References

[1] The APE100 Collaboration, *APE100 Primer*, INFN, A100/PRIM/G02.

[2] The APE100 Collaboration, *zCPU User Guide*, INFN, A100/ZCPU/G01.

[3] A. Bartoloni et al., *A Hardware Implementation of the APE100 Architecture*, in: International Journal of Modern Physics, C 4 (1993), p. 969.

[4] A. Bartoloni et al., *The Software of the APE100 Processor*, in: International Journal of Modern Physics, C 4 (1993), p. 955.

[5] A. Bartoloni et al., *APEmille: A Parallel Processor in the Teraflops Range*, manuscript, 1995.

[6] G. Bastianello et al., *A high performance single chip processing unit for parallel processing and data acquisition systems*, in: Nuclear Instruments and Methods in Physics Research, A324 (1993), p. 543.

[7] E. Börger, G. Del Castillo, P. Glavan, D. Rosenzweig, *Towards a mathematical specification of the APE100 architecture: the APESE model*, in: B. Pehrson and I. Simon (Eds.), *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam, 396–401.

[8] E. Börger, I. Durdanovic, *Correctness of Compiling Occam to Transputer Code*, March 1995.

[9] E. Börger, U. Glässer, W. Müller, *Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines*, in: Carlos Delgado Kloos and Peter T. Breuer (Eds.), *Formal Semantics for VHDL*, pp. 107–139, Kluwer Academic Publishers, 1995.

[10] E. Börger, S. Mazzanti, *A correctness proof for pipelining in RISC architectures* (manuscript).

[11] E. Börger, D. Rosenzweig, *The WAM - Definition and Compiler Correctness*, in: *Logic Programming: Formal Methods and Practical Applications* (C.Beierle, L.Plümer, Eds.), Elsevier Science B.V./North-Holland, Series in Computer Science and Artificial Intelligence, 1995, pp. 20–90 (chapter 2).

[12] A. Brüggemann, L. Priese, D. Rödding, R. Schätz, *Modular decomposition of automata*, in: Springer LNCS 171, 1984, 198-236.

[13] G. Del Castillo, *Descrizione matematica dell'architettura parallela APE100*, Tesi di Laurea (in Italian), Università di Pisa, Pisa, 1995.

[14] Y. Gurevich, *Evolving Algebras 1993: Lipari Guide*, in: Specification and Validation Methods, Ed. E. Börger, Oxford University Press, 1995.

Recent Publications in the BRICS Notes Series

NS-95-4 Yuri Gurevich and Egon Börger. *Evolving Algebras. Mini-Course*. July 1995. iv+222 pp.

NS-95-3 Andrew D. Gordon. *Bisimilarity as a Theory of Functional Programming. Mini-Course*. July 1995. iv+59 pp.

NS-95-2 Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors. *Proceedings of the Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS* (Aarhus, Denmark, 19–20 May, 1995), May 1995. vi+334 pp.

NS-95-1 Igor Walukiewicz. *Notes on the Propositional μ-calculus: Completeness and Related Results*. February 1995. 54 pp.

NS-94-6 Uffe H. Engberg, Kim G. Larsen, and Peter D. Mosses, editors. *Proceedings of the 6th Nordic Workshop on Programming Theory* (Aarhus, Denmark, 17–19 October, 1994), December 1994. v+483pp.

NS-94-5 Andrew M. Pitts. *Some Notes on Inductive and Co-Inductive Techniques in the Semantics of Functional Programs,* DRAFT VERSION. December 1994. vi+135 pp.

NS-94-4 Peter D. Mosses, editor. *Abstracts of the 6th Nordic Workshop on PROGRAMMING THEORY* (Aarhus, Denmark, 17–19 October, 1994), October 1994. v+52 pp.

NS-94-3 Sven Skyum, editor. *Complexity Theory: Present and Future* (Aarhus, Denmark, 15–18 August, 1994), September 1994. v+213 pp.

NS-94-2 David Basin. *Induction Based on Rippling and Proof Planning. Mini-Course*. August 1994. 62 pp.

NS-94-1 Peter D. Mosses, editor. *Proc. 1st International Workshop on Action Semantics* (Edinburgh, 14 April, 1994), May 1994. 145 pp.