Technical Report

UCAM-CL-TR-240
ISSN 1476-2986

Number 240

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Symbolic compilation and execution of programs by proof: a case study in HOL

Juanito Camilleri

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Symbolic Compilation and Execution of Programs by Proof: a case study in HOL

Juanito Camilleri[1] *jac1@cl.cam.ac.uk*
Computer Laboratory, University of Cambridge,
New Museums Site, Cambridge CB2 3QG, England

Abstract

This paper illustrates the symbolic 'compilation' and 'execution' of programs by proof using the proof assistant HOL. We formalise the operational semantics of an occam-like programming language OC and show how synchronous communication in OC compiles to an intermediate programming language SAFE whose compilation yields instructions intended to drive machines that communicate via shared memory. We show how the symbolic formal manipulation of terms of a programming language, subject to the definition of its semantics, can *animate* a desired effect—be it compilation or execution. Needless to say, such compilation and execution by proof is rather slow, but it is fast enough to give vital feedback about the compilation algorithm being used. Without such animation it is hard to anticipate whether the compilation algorithm is reasonable before attempting to verify it. This is particularly true when attempting to find a plausible handshaking protocol that implements synchronous communication.

1 Introduction

This paper illustrates the symbolic 'compilation' and 'execution' of programs by proof. The mechanised proof assistant HOL [Gor85], is used to conduct this study. The HOL system supports a version of higher-order logic based on Church's formulation of simple type theory.

We formalise the operational semantics of an occam-like programming language OC and show how synchronous communication in OC compiles to an intermediate programming language SAFE whose compilation yields instructions intended to drive machines that communicate via shared memory. The ultimate goal of this work is to verify the OC compiler by proving a theorem asserting that the state changes resulting from the execution of the compiled code correspond to the state changes stipulated by the operational semantics. This paper, however, does not address this issue in detail. Instead, we show how the symbolic formal manipulation of terms of a programming language, subject to the definition of its semantics, can *animate* a desired effect—be it compilation or execution.

The approach to animation adopted in this paper is based on *conversions* [Pau]. These special class of inference rules in HOL, map a term of the logic to a theorem expressing the equality of that term to some other term. Because of the way the logic of the HOL system is represented in the strongly-typed, general-purpose programming language ML, the user can write conversions tailored for a specific application. For example β-conversion

[1]Contact address as from February 1992: Department of Computer Studies, University of Malta, Msida, Malta. Fax: +356 336450

is represented in HOL as a conversion that maps a term of the form $(\lambda x.t_1)t_2$ to a theorem:

$$\vdash (\lambda x.t_1)t_2 = t_1[t_2/x]$$

Therefore the conversion can generate a class of equations of the above form. Futhermore, conversions can be composed in an arbitrarily complex way. They can be constructed to yield a specific rewriting strategy on classes of terms to yield theorems of a required form.

There are several reasons for adopting the conversion approach to animation. First, it is relatively straightforward to write a conversion that animates behavioural definitions in the way described. These conversions can also be used for symbolic simulation or *partial simulation* when variables are used to represent components of a system while the behaviour of some other component is simulated. Second, because the same definitions are used in both animation and verification, we avoid errors resulting from discrepancies between the model of the definitions which is animated and the definitions whose properties are verified. Furthermore, the results of the animation are theorems backed up by formal proof. Hence if the animation of a definition conveys unexpected behaviour then this must be due to oversights in the definition. Third, the conversions used for animation can be used later to generate theorems that facilitate the verification process. Finally, when developing a specification for a large system, one can take an incremental and compositional approach to animation. In other words, one can write conversions on-the-fly to animate the definitions of the subsystems and then compose these conversions into a conversion that animates the whole system. For example, when defining a compiler, one may choose to write conversions that animate the compilation of expressions and declarations separately and then use those conversions when constructing a conversion that animates the compilation of commands.

In this paper we will present a conversion that maps an arbitrary term t of OC to a theorem expressing that the result of compiling t in an environment e is equivalent to some term t' of SAFE:

$$\vdash \texttt{Oc\_compile}\ t\ e = t'$$

The term t' is then manipulated by another conversion that yields a theorem asserting that the compilation of t' in some environment e' yields a list of machine instructions i:

$$\vdash \texttt{Safe\_Compile}\ t'\ e' = i$$

Another conversion then fetches the appropriate instruction of the instruction list, as determined by a program counter, and returns a theorem asserting the state resulting from the execution of the instruction within the framework of the SAFE-machine. Informally, suppose

$$\vdash \texttt{Machine\_Step}\ i\ s = s'$$

then we know that one step in the execution of i in state s yields the state s'. Of course one can apply the conversion again in state s' to yield

$$\vdash \texttt{Machine\_Step}\ i\ s' = s''$$

which of course can be rewritten as:

$$\vdash \texttt{Machine\_Step}\ i\ (\texttt{Machine\_Step}\ i\ s) = s''$$

This argument can be applied repetitively until the program terminates. Based on these arguments one can build a general-purpose conversion that animates the execution of the code resulting from the compilation of OC programs. Needless to say, such compilation and execution by proof is rather slow, but it is fast enough to give vital feedback about the compilation algorithm being used. Without such animation it is hard to anticipate whether the compilation algorithm is reasonable before attempting to verify it. Conversions similar to the ones described above played a significant role in determining a plausible handshaking protocol.

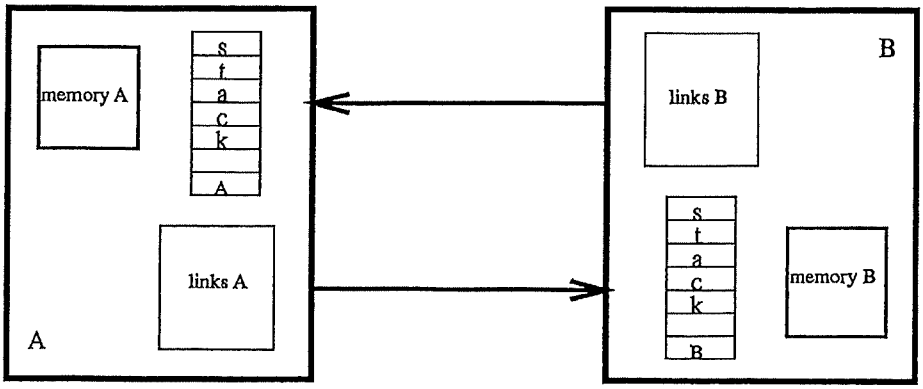
2 The SAFE machine and instruction set

Consider a machine whose state is a tuple $\langle(pc, stk), (mry, lnk)\rangle$ comprising a program counter pc, a stack stk, local memory mry, and link[2] memory lnk that can be read by other machines. The local memory and link memory of a machine are represented as functions of type `address->val`, a stack has type `(val)list`, and a program counter is a positive integer. At each step of computation, a machine takes input[3] from external links and executes the instruction specified by the program counter. The instruction may read to or write from local memory, or it may output to the machine's links.

The SAFE instruction set is described in Fig.1 and is defined in HOL using the type definition package [Mel88]. The result of this definition is the following theorem of higher-order logic, which is a complete and abstract characterisation of the data-type `instruction` and asserts the admissibility of defining functions over instructions by primitive recursion. Primitive recursion over instructions is used when defining a function `Step` that determines the effect on the state when the instructions are executed (see Fig.2).[4]

```
instruction =
⊢ ∀e0 f0 f1 f2 e1 e2 f3 f4 f5 f6 f7 f8 f9.
    ∃! fn.
     (fn SKP = e0) ∧
     (fn STP = e1) ∧
     (fn POP = e2) ∧
     (∀n. fn(JMP n) = f0 n) ∧
     (∀n. fn(JMZ n) = f1 n) ∧
     (∀n. fn(JMN n) = f2 n) ∧
     (∀n. fn(OP0 n) = f3 n) ∧
     (∀f. fn(OP1 f) = f4 f) ∧
     (∀f. fn(OP2 f) = f5 f) ∧
     (∀n. fn(GET n) = f6 n) ∧
     (∀n. fn(PUT n) = f7 n) ∧
     (∀n. fn(OUT n) = f8 n) ∧
     (∀n. fn(INP n) = f9 n)
```

[2]The links discussed in this paper should not be mistaken as links used in a transputer.
[3]Strictly speaking a machine inputs a function that maps addresses of external links to their values.
[4]Note that the HOL term `(b => x | y)` means: 'if b then x else y'.



`SKP`	A skip instruction
`STP`	Stops the machine by setting the program counter to 0
`POP`	Pop the top of the stack
`JMP`	Unconditional jump to instruction n
`JMZ` n	Pop stack then jump to instruction n if the result is zero
`JMN` n	Pop stack then jump to instruction n if the result is non-zero
`OPO` v	Push v onto stack
`OP1` *op*	Pop one value from stack, perform *op*, push result
`OP2` *op*	Pop two values from stack, perform *op*, push result
`GET` x	Push the contents of memory location x onto the stack
`PUT` x	Pop the top of the stack and store the result in memory location x
`OUT` x	Pop the top of the stack and store the result in link x
`INP` x	Input the value of x and push it onto the stack

Figure 1: The SAFE machine and instruction set.

```
rec_define
  instruction
  "(Step (SKP) =
     λinp ((pc,stk),(mry,lnk)).
       ((pc+1,stk),(mry,lnk)))                                          ∧
   (Step (JMP n)  =
     λinp ((pc,stk),(mry,lnk)).
       ((n,stk),(mry,lnk)))                                             ∧
   (Step (JMZ n)  =
     λinp ((pc,stk),(mry,lnk)).
       ((HD stk = 0) => ((n,TL stk),(mry,lnk))
                      | ((pc+1,TL stk),(mry,lnk))))                     ∧
   (Step (JMN n)  =
     λinp ((pc,stk),(mry,lnk)).
       ((HD stk = 0) => ((pc+1,TL stk),(mry,lnk))
                      | ((n,TL stk),(mry,lnk))))                        ∧
   (Step STP =
     λinp ((pc,stk),(mry,lnk)).
       ((0,stk),(mry,lnk)))                                             ∧
   (Step POP =
     λinp ((pc,stk),(mry,lnk)).
       ((pc+1,TL stk),(mry,lnk)))                                       ∧
   (Step (OP0 v) =
     λinp ((pc,stk),(mry,lnk)).
       ((pc+1,CONS v stk),(mry,lnk)))                                   ∧
   (Step (OP1 op1) =
     λinp ((pc,stk),(mry,lnk)).
       ((pc+1,CONS(op1(HD stk))(TL stk)),(mry,lnk)))                    ∧
   (Step (OP2 op2) =
     λinp ((pc,stk),(mry,lnk)).
       ((pc+1,CONS(op2(HD stk)(HD(TL stk)))(TL(TL stk))),(mry,lnk)))    ∧
   (Step (GET x)  =
     λinp ((pc,stk),(mry,lnk)).
       ((pc+1,CONS (mry x) stk),(mry,lnk)))                             ∧
   (Step (PUT x)  =
     λinp ((pc,stk),(mry,lnk)).
       ((pc+1,stk),((Store (HD stk) x mry),lnk)))                       ∧
   (Step (OUT x)  =
     λinp ((pc,stk),(mry,lnk)).
       ((pc+1,stk),(mry,(Store (HD stk) x lnk))))                       ∧
   (Step (INP x)  =
     λinp ((pc,stk),(mry,lnk)).
       ((pc+1,CONS (inp x) stk),(mry,lnk)))"
```

Figure 2: The function Step.

Instructions are fetched according to the following definition, which ensures that the computation stops once the program counter goes beyond the length of the instruction list.

```
Fetch =
⊢ ∀inst n.
    Fetch inst n =
    (((n = 0) ∨ n > (LENGTH inst)) => STP | EL(n - 1)inst)
```

Note that EL i $[x_0; x_1; \ldots; x_n] = x_i$ and LENGTH is a function that determines the length of a list.

In this paper we assume a configuration in which there are two SAFE machines that execute instructions in lock-step (see Fig.1). This assumption allows us to animate a handshake between the two machines in a simple, if not entirely realistic, framework. Given the definition of the function Step shown in Fig.2, and given that PC and LNK are functions that extract the program counter and link memory of a state, the overall behaviour resulting from the simultaneous execution of one step in each machine is defined as follows:

```
Machines_Step =
⊢ ∀((A,B):(instruction)list#(instruction)list) ((s0,s1):state#state).
    Machines_Step (A,B) (s0,s1) =
    Step(Fetch A (PC s0))(LNK s1)s0,
    Step(Fetch B (PC s1))(LNK s0)s1
```

In a step of execution Machine A inputs from the links of Machine B and vice-versa. Each machine then proceeds to execute an instruction. Therefore, if a machine writes to one of its links at some stage of computation, then the value can only be read by the other machine in the next stage of computation.

The definition of Machines_Step can be used to define the symbolic execution of a series of steps by primitive recursion on the number of steps taken.

```
Machines_Steps =
⊢ (∀i s. Machines_Steps 0 i s = s) ∧
  (∀n i s.
    Machines_Steps(SUC n)i s = Machines_Steps n i(Machines_Step i s))
```

Execution carried out according to this definition can be animated in HOL. One can write a conversion that given a pair of instruction lists and a pair of initial machine states, executes the instructions according to the definition of the SAFE machine starting from the initial states.

Each step of animation is done by a systematic specialization, unfolding and simplification of Machines_Step with the instructions i and the current state s of the two machines. The result is a theorem of the form:

$$\vdash \texttt{Machines\_Step}\ i\ s = s'$$

where s' is the state after one step of execution. Of course one can apply the same procedure to the instructions in state s' to yield

$$\vdash \texttt{Machines\_Step}\ i\ s' = s''$$

where s'' is the state after two steps of execution. This can be repeated until the machines halt—i.e. either both machines encounter a STP instruction or both execute all their instructions. Note that if the program of one of the machines terminates while the other can still execute instructions, then the machine with the terminated program will execute dummy SKP instructions until such time as the other terminates. The trail of theorems generated as described above and the definition of Machines_Steps can be used to automatically prove a final theorem asserting the result of executing the program on the machine being discussed.

The conversion that achieves this is called Animate_Machines_Steps. For example, consider the pair of instruction lists inst_pair

```
[OP0 1;PUT 2;JMZ 7;GET 2;OP1 PRE;JMP 2;INP 1;PUT 1], [OP0 6;OUT 1]
```

and assume that the initial state init_states of the machines is:

```
(((1:pc),([]:stk)),(((λx.0):mry),((λx. 0):lnk))),
(((1:pc),([]:stk)),(((λx.0):mry),((λx. 0):lnk)))
```

The result of executing Animate_Machines_Steps inst_pair init_states is the theorem shown in Fig.3. The table shows a trace of the state changes resulting from each step of the computation done to prove this theorem. The theorem itself asserts that the machines halt in eleven steps to yield the following final state:

```
(0:pc,[6]:stk),((λx'.((1=x') => 6 | 0)):mry,λx.0:lnk),
(0:pc,[6]:stk),((λx.0):mry,λx'.((1=x') => 6 | 0):lnk)
```

3 The language SAFE

The language SAFE discussed in this paper is a variant of SAFE0 [Hal89] and one of a family of languages being studied on the safemos project. SAFE allows one to program the SAFE machine using primitive constructs that admit communication through shared memory. Arithmetic expressions in SAFE may access the values of an external link. For example, the expression (3 + y) - (INPUT x) is valid and if evaluated on Machine A yields the result of subtracting the value of link x on Machine B from the sum of 3 and the value of memory location y of Machine A.

The syntactic classes of SAFE expressions, declarations, commands and programs are represented by the recursive types exp, dec, cmd and prog respectively. The syntax is defined as follows:

```
exp   ::= VAR string                          (local variable)
        | INPUT string                        (input from external link)
        | CONST num                           (constant)
        | UNOP (num->num) exp                 (unary operator)
        | BINOP (num->(num->num)) exp exp     (binary operator)
dec   ::= LVAR string                         (declare a local variable)
        | LINK string                         (declare a link)
```

Machine A	Machine B
((1,□),(λx.0),(λx.0))	(1,□),(λx.0),(λx.0)
((2,[1]),(λx.0),(λx.0))	(2,[6]),(λx.0),(λx.0)
((3,[1]),(λx'.((2 = x') => 1 | 0)),(λx.0))	(3,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
((4,□),(λx'.((2 = x') => 1 | 0)),(λx.0))	(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
((5,[1]),(λx'.((2 = x') => 1 | 0)),(λx.0))	(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
((6,[0]),(λx'.((2 = x') => 1 | 0)),(λx.0))	(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
((2,[0]),(λx'.((2 = x') => 1 | 0)),(λx.0))	(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
((3,[0]),(λx'.0),(λx.0))	(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
((7,□),(λx'.0),(λx.0))	(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
((8,[6]),(λx'.0),(λx.0))	(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
((9,[6]),(λx'.((1 = x') => 6 | 0))),(λx.0))	(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
((0,[6]),(λx'.((1 = x') => 6 | 0))),(λx.0))	(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))

```
⊢ Machines_Steps
11
([OPO 1;PUT 2;JMZ 7;GET 2;OP1 PRE;JMP 2;INP 1;PUT 1],
[OPO 6;OUT 1])
(((1,□),(λx.0),(λx.0)),(1,□),(λx.0),(λx.0)) =
((0,[6]),(λx'.((1 = x') => 6 | 0))),(λx.0)),
(0,[6]),(λx.0),(λx'.((1 = x') => 6 | 0))
```

Figure 3: The symbolic execution of an example program.

```
cmd   ::= SKIP                (instantaneous skip)
        | TSKIP               (time-consuming skip)
        | STOP                (stop)
        | ASSIGN string exp   (assign)
        | OUTPUT string exp   (output to a local link)
        | IF exp cmd cmd      (two-armed conditional)
        | SEQ cmd cmd         (sequence)
        | WHILE exp cmd       (while loop)
        | BLK dec cmd         (block)

prog  ::= PAR cmd cmd         (parallel composition)
```

The results of defining these recursive types in HOL are theorems that state the admissibility of defining functions over the syntactic structure of SAFE programs by primitive recursion. These theorems are of direct utility, since that is how the compiler is defined.

3.1 The SAFE compiler

In this paper we will refer to the components of a parallel composition as *processes*. The compilation of the commands in a process proceeds with respect to the *variable* and *link* environments generated by declarations made in the process, as well as with respect to the link environment resulting from the declarations of links in the other process. For example, consider the following SAFE program:

```
PAR (
  BLK (LINK 'x') (
  BLK (LINK 'y') (
  BLK (LVAR 'a') (                    Process A: compiles to
  SEQ (                               instructions that are
    OUTPUT 'y' (VAR 'a')) (           executed on Machine A.
    ASSIGN 'a' (INPUT 'x'))))) (
  BLK (LINK 'x') (
  BLK (LINK 'y') (
  BLK (LVAR 'c') (                    Process B: compiles to
  SEQ (                               instructions that are
    OUTPUT 'x' (CONST 6)) (           executed on Machine B.
    ASSIGN 'c' (INPUT 'y'))))))
```

This will be compiled to a pair of instruction lists whose components are executed on Machine A and B respectively. Both processes declare links called `'x'` and `'y'`; the link `'x'` in Process A is distinct from that in Process B and similarly for the links called `'y'`. The assignment in Process A should be understood as follows: input a value from link `'x'` in Machine B and assign it to the location `'a'` in Machine A. The output of this process states: output the value at location `'a'` of Machine A to link `'y'` of Machine A. Similarly, `'x'` in the output construct of Process B refers to link `'x'` of Machine B, while `'y'` in the input expression in that process refers to link `'y'` of Machine A. Therefore, when compiling Process A, one needs to know the addresses of links declared by Process B and vice-versa.

Let `env:mry#address` represent a variable environment and `link:lnk#address` represent a link environment where `address` represents the next free memory location and the next free link location respectively. The function

```
Com_cmd: cmd->env->link->link->num->(instruction)list
```

is defined by primitive recursion on the type `cmd`, where `env:mry#address` represents the local variable environment, the two arguments of type `link` represent the local and external link environments respectively, and `num` represents the address in memory from which the compiled code is to be stored.[5]

The definition of `Com_cmd` is the following theorem of higher-order logic:

```
Com_cmd =
⊢ (∀e l el. Com_cmd SKIP e l el = Com_SKIP) ∧
  (∀e l el. Com_cmd TSKIP e l el = Com_TSKIP) ∧
  (∀e l el. Com_cmd STOP e l el = Com_STOP) ∧
  (∀x ex e l el.
    Com_cmd(ASSIGN x ex)e l el = Com_ASSIGN(FST e x)ex e l el) ∧
  (∀x ex e l el.
    Com_cmd(OUTPUT x ex)e l el = Com_OUTPUT(FST l x)ex e l el) ∧
  (∀ex c1 c2 e l el.
    Com_cmd(IF ex c1 c2)e l el =
    Com_IF ex(Com_cmd c1 e l el)(Com_cmd c2 e l el)e l el) ∧
  (∀c1 c2 e l el.
    Com_cmd(SEQ c1 c2)e l el =
    Com_SEQ(Com_cmd c1 e l el)(Com_cmd c2 e l el)) ∧
  (∀ex c e l el.
    Com_cmd(WHILE ex c)e l el = Com_WHILE ex(Com_cmd c e l el)e l el) ∧
  (∀d c e l el.
    Com_cmd(BLK d c)e l el =
    Com_cmd c(FST(Com_dec d e l))(SND(Com_dec d e l))el)
```

Details of the compilation algorithm for each of the constructs of SAFE are not included here. As an example, however, consider the following definition of the compilation of `WHILE` constructs:

```
Com_WHILE =
⊢ ∀ex c_fn e l i n.
    Com_WHILE ex c_fn e l i n =
    (     let ex_prog = Com_exp ex e l i
     in   let n1 = n + (LENGTH ex_prog)
     in   let c_prog = c_fn(n1 + 1)
     in   let n2 = (n1 + 1) + (LENGTH c_prog)
     in
          ((c_prog = []) =>
            ex_prog ++ [JMN n] |
            ex_prog ++ ([JMZ(n2 + 1)] ++ (c_prog ++ [JMP n]))))
```

[5] We assume that a block of memory in each machine is reserved exclusively for the program code.

where Com_exp is the function that compiles expressions, LENGTH computes the length of a list of instructions, and n is the address of the first instruction of the resulting code. Notice that if the body of the loop is empty (i.e. represents SKIP) then optimised code is generated.

The compilation of SAFE programs is defined by the following theorem:

```
Safe_Compile =
⊢ ∀c0 c1 (e0,e1) (l0,l1).
    Safe_Compile(PAR c0 c1) (e0,e1) (l0,l1) =
    (let le0 = link_analysis c0 l0
     and le1 = link_analysis c1 l1
     in
      Com_cmd c0 e0 l0 le1 1,Com_cmd c1 e1 l1 le0 1)
```

where link_analysis[6] specifies the static analysis of link declarations made in a process. As seen later, this analysis allows one to compile a process of a program with information about the link declarations made in the other process as required. This is illustrated in the next section.

3.1.1 A compilation conversion for SAFE programs

Given a program of the form PAR c_0 c_1 one can systematically unfold it according to the definition of the compiler to yield the pair of those instruction lists that are the compiled code of c_0 and c_1 respectively. For example, consider the compilation of the program:

```
PAR (
  BLK (LVAR 'a') (                          Process A
  SEQ (
    TSKIP) (
    ASSIGN 'a' (INPUT 'x'))) (
  BLK (LINK 'x') (                          Process B
  OUTPUT 'x' (CONST 6)))
```

Rewriting Safe_Compile (PAR A B) using the definition of Safe_Compile yields

```
⊢ Safe_Compile (PAR A B) (e0,e1) (l0,l1) =
      (let le0 = link_analysis A l0
       and le1 = link_analysis B l1
       in
        Com_cmd A e0 l0 le1 1,Com_cmd B e1 l1 le0 1)
```

where (e0,e1) and (l0,l1) represent the initial variable and link environments for the machines—let us say that both are equal to:

```
((λy.0,1),(λy.0,1))
```

[6] link_analysis is defined by primitive recursion on SAFE commands. The definition is trivial and is not included here.

and the last argument to Com_cmd indicates that the code resulting from the compilation of A is to commence at address 1 in the block of memory reserved for the program code in Machine A (similarly for the compiled code of B on Machine B).

Rewriting with the definition of link_analysis yields

```
⊢ Safe_Compile (PAR A B) (e0,e1) (l0,l1) =
      (let le0 = (λy.0,1)
       and le1 = ((λi'.(('x' = i') => 1 | (λy.0)i')),2)
       in
        Com_cmd A e0 l0 le1 1,Com_cmd B e1 l1 le0 1)
```

In other words, link_analysis detects no declarations of links in Process A, and the declaration of link 'x' in Process B is detected and the link environment le1 is such that 'x' identifies the next free link (i.e. link 1) on machine B. Thus when Process A inputs from link 'x' of machine B, the input is in fact from link 1 of that machine. Unfolding the definition further yields:

```
⊢ Safe_Compile (PAR A B) (e0,e1) (l0,l1) =
  Com_cmd A(λy.0,1)(λy.0,1)((λi'.(('x' = i') => 1 | (λy.0)i')),2) 1,
  Com_cmd B(λy.0,1)(λy.0,1)(λy.0,1) 1
```

Rewriting with the definitions of Com_cmd, Com_decl, Com_LVAR, Com_ASSIGN, Com_INPUT, Com_TSKIP, Com_OUTPUT, and Com_CONST and simplifying further yields

```
⊢ Safe_Compile
   (PAR
    (BLK(LVAR 'a')(SEQ TSKIP (ASSIGN 'a'(INPUT 'x'))))
    (BLK(LINK 'x')(OUTPUT 'x'(CONST 6))))
   (((λx. 0),1),(λx. 0),1)
   (((λx. 0),1),(λx. 0),1) =
   [SKP;INP 1;PUT 1],[OPO 6;OUT 1]
```

It is straightforward to automate the reasoning described informally above as a conversion that symbolically compiles arbitrary SAFE programs into machine instructions intended to be executed on the configuration shown in Fig.1. The conversion that has this effect is called Animate_Safe_Compile.

3.2 The OC language

The language OC imposes a discipline, known as *handshaking*, on the communication between two processes. The handshaking protocol of OC is implemented in the SAFE language which in turn is compiled into machine instructions which are expected to yield the desired effect, namely *synchronous communication.*

The syntactic classes of OC expressions, variable declarations, commands, channels and programs are represented by recursive types exps, decl, acmds, chans and progs respectively. The syntax is defined as follows:

```
exps   ::= Var string                          (local variable)
         | Const num                           (constant)
         | Unop (num->num) exps                (unary operator)
         | Binop (num->(num->num)) exps exps   (binary operator)
```

```
decl    ::= Dec string              (declare a local variable)

acmds   ::= Skip                    (instantaneous skip)
          | Stop                    (stop)
          | Assign string exps      (assign)
          | Inpt string exps        (input from a channel)
          | Outpt string exps       (output to a channel)
          | If exps acmds acmds     (two-armed conditional)
          | Seq acmds acmds         (sequence)
          | While exps acmds        (while loop)
          | Blk decl acmds          (block)

chans   ::= AB string               (channel from Machine A to B)
          | BA string               (channel from Machine B to A)

progs   ::= Par acmds acmds         (parallel composition)
          | Chan chans progs        (program block)
```

Once again note that OC programs are intended to be executed on the machine configuration in Fig.1. A typical program comprises a series of channel declarations followed by a parallel construction of two commands, the first to be executed on Machine A and the second on Machine B. Channels are directed, and they are implemented using SAFE links. For example, a channel `'ch'` that conveys data from Machine A to B is implemented by three links: the *ready link* `'Rch'`, the *data link* `'Dch'` and the *acknowledge link* `'Ach'`. The ready and data links are declared on Machine A and the acknowledge link is declared on Machine B. The converse holds for channels that convey data from B to A.

The role of links in the implementation of channels is illustrated by the SAFE code that implements the communication primitives of OC. The compilation of the output construct of OC is defined by the following theorem:[7]

```
Com_Outpt =
⊢ ∀ch e.
    Com_Outpt ch e =
    (let (chr,chd,cha) = links_of_chan ch
     in
     SEQ (
       OUTPUT chd(Com_exps e)) (
       SEQ (
         OUTPUT chr(CONST tt)) (
         SEQ (
           WHILE(BINOP $==(INPUT cha)(CONST ff))
             SKIP) (
           OUTPUT chr(CONST ff)))))
```

According to this definition, an output command `Outpt` *ch e*:

- evaluates the expression *e* and writes the resulting value on link `D`*ch*, then

[7]The term `links_of_chan` specifies a function that, when given a string `'ch'`, returns a triple of strings (`'Rch'`,`'Dch'`,`'Ach'`).

- enables the ready link R*ch*, then
- waits until the neighbouring process acknowledges (i.e. until A*ch* is true).

An input construct compiles as follows:

```
Com_Inpt =
⊢ ∀ch x.
    Com_Inpt ch x =
    (let (chr,chd,cha) = links_of_chan ch
     in
     SEQ (
       WHILE(BINOP $==(INPUT chr)(CONST ff))
         SKIP) (
       SEQ (
         ASSIGN x(INPUT chd)) (
         SEQ (
           OUTPUT cha(CONST tt)) (
           SEQ (
             delay 3)(
             SEQ (
               OUTPUT cha(CONST ff)) (
               delay 4))))))
```

In this case, an input command Inpt *ch x*:

- waits until the neighbouring process enables the ready link R*ch* of the channel, then
- reads the value of the data link D*ch* and assigns it to location x, then
- acknowledges the input by enabling the acknowledge link A*ch*, then
- waits for the duration of 3 instructions—i.e. enough time to ensure that a neighbouring output process never fails to detect the acknowledgement, then
- disables the acknowledge link of the channel, and finally
- waits for the duration of another four instructions to ensure that the neighbouring output process has disabled the ready link of the channel.

The mapping of the other constructs of OC into SAFE is straightforward.

The compilation of OC programs is defined by primitive recursion on the types declared for the syntactic classes of the language. As is the case for the SAFE compiler, one can implement a conversion Animate_Oc_Compile that maps an arbitrary OC program to a theorem asserting the result of compiling the program.

4 Animating the handshaking protocol

Suppose that two processes A and B are executing in parallel, that `Inpt` *ch x* is one of the commands in A, and that `Outpt` *ch e* is one of the commands in B. When these are compiled into machine code (via the SAFE compiler), the code generated for the input command is the following part of the code generated for A:

```
1.    [INP Rch;          enter loop and wait until
2.     OPO ff;
3.     OP2 $==;          ready link is true
4.     JMN 1;
5.     INP Dch;          input from data link, and
6.     PUT x;            store the value in x
7.     OPO tt;
8.     OUT Ach;          set the acknowledge link to true
9.     SKP;SKP;SKP;      delay for three instruction steps
10.    OPO ff;
11.    OUT Ach;          reset acknowledge link to false
12.    SKP;              delay for four instruction steps
13.    SKP;
14.    SKP;
15.    SKP]
```

If the code generated for *e* is `e`, then the output command compiles to:

```
       e ++              execute code generated for e
1.    [OUT Dch;          output result to data link
2.     OPO tt;
3.     OUT Rch;          set ready link to true
4.     INP Ach;          enter loop and wait until
5.     OPO ff;
6.     OP2 $==;          acknowledge link is false
7.     JMN 5;
8.     OPO ff;
9.     OUT Rch]          reset ready link to false
```

which is part of the code generated for B. Recall that the resulting code for A and B is executed in lock-step on machines A and B respectively.

The code for the input command cannot be executed beyond its fourth instruction until the output code has evaluated the expression `e`, has output the value on link `Dch`, and has enabled the ready link `Rch`. The input command detects the willingness of the output process at most four steps after the ready link has been enabled. It then inputs from the data link `Dch`. In the meantime the output process waits for an acknowledgement. Consider the worst case scenario after the input process acknowledges. This occurs when the 8th instruction of the input command coincides with the 5th instruction of the output command—i.e. when the loop in the output process that is meant to detect

acknowledgement just misses the acknowledgement.[8] Consequently, the output process needs to go round the loop once more, thus the input process should wait for at least four steps before disabling the acknowledgement. The disabling of the acknowledgement link by the input (i.e. the execution of its 13th instruction) coincides with the execution of the 6th instruction of the output. This means that the output command has three instructions to execute before it can disable the ready link. The input process therefore must wait for a further four instructions before it can terminate safely. Otherwise, one can envisage, and indeed animate, a scenario in which two input commands are consumed by a single output!

From the description above, it is evident that the compilation of a handshaking protocol is by nature extremely machine dependent. For this reason it is rather hard to convince one's self that the protocol behaves as required. This is where the animation of compilation and execution were particularly reassuring. Consider the OC program `Handshake` defined as follows:

```
Chan (BA 'ch') (
Par (
  Blk (Dec 'X') (
  Inpt 'ch' 'X'))
  (
  Outpt 'ch' (Const 5)))
```

Executing the following conversion `Animate_Oc_Compile Handshake` yields the following theorem of higher-order logic asserting the result of compiling `Handshake` to SAFE:

```
⊢ Oc_Compile
    (Chan (BA 'ch') (
     Par (
       Blk(Dec 'X') (
       Inpt 'ch' 'X'))
       (
       Outpt 'ch' (Const 5)))) =

PAR (
  BLK (LINK(STRING A 'ch')) (
  BLK (LVAR 'X') (
  SEQ (
    WHILE(BINOP $==(INPUT(STRING R 'ch'))(CONST ff))
      SKIP) (
    SEQ (
      ASSIGN 'X'(INPUT(STRING D 'ch'))) (
      SEQ (
        OUTPUT(STRING A 'ch')(CONST tt)) (
        SEQ (
          SEQ TSKIP(SEQ TSKIP(SEQ TSKIP SKIP)))(
          OUTPUT(STRING A 'ch')(CONST ff)))))))) (
```

[8] Recall that when a value is output to a link, the value isn't available for input until the next step of computation.

```
BLK (LINK(STRING R 'ch')) (
BLK (LINK(STRING D 'ch')) (
SEQ (
  OUTPUT(STRING D 'ch')(CONST 5)) (
  SEQ (
    OUTPUT(STRING R 'ch')(CONST tt)) (
    SEQ (
      SEQ TSKIP(SEQ TSKIP(SEQ TSKIP SKIP))) (
      SEQ (
        OUTPUT(STRING R 'ch')(CONST ff)) (
        WHILE(BINOP $==(INPUT(STRING A 'ch'))(CONST ff))SKIP)))))))
```

If the SAFE program on the right-hand side of the equality is called `Safe_Handshake`, then executing `Animate_Safe_Compile Safe_Handshake` when the initial memory and link environment is $(((\lambda x.0),1),(\lambda x.0),1)$ on both machines, yields the following theorem asserting the result of compiling the program into machine code:

```
⊢ Safe_Compile Safe_Handshake
   (((λx. 0),1),(λx. 0),1)
   (((λx. 0),1),(λx. 0),1) =
   [INP 1;OP0 ff;OP2 $==;JMN 1;INP 2;PUT 1;OP0 tt;OUT 1;SKP;SKP;SKP;
    OP0 ff;OUT 1;SKP;SKP;SKP;SKP],
   [OP0 5;OUT 2;OP0 tt;OUT 1;OP0 ff;OUT 1;INP 1;OP0 ff;OP2 $==;JMN 5]
```

The execution of the resulting pair of instruction lists can be animated using the conversion `Animate_Machines_Steps`. The trace of computation and the theorem asserting the state change of the machines resulting from the execution of the code are shown in Fig.4. Notice that according to the theorem shown in Fig.4, the state resulting from the computation is:

```
((0:pc,[0;1;5]:stk),(λx'.((1=x')=>5|0)):mry,(λx'.0):lnk),
((0:pc,[0;1;5]:stk),(λx.0):mry,(λx'.((2=x')=>5|0)):lnk)
```

The stacks of both machines contain three values after the handshake. This highlights that the compilation algorithm presented above, generates code for a handshake that has side-effects on the contents of the stack. Hence the compilation algorithm in question does not manage stack space efficiently. It is trivial to alter the compilation algorithm to generate code that is more efficient in this sense. The algorithm presented here was chosen to illustrate the role played by animation in detecting such features.

Consider now the OC program `Check_Handshake`:

```
Chan (BA 'ch') (
Par (
  Blk (Dec 'X') (
  Seq (
    delay 5 (
    Seq (
      Inpt 'ch' 'X')(
      Inpt 'ch' 'X')))) (
  Outpt 'ch' (Const 5)))
```

Machine A	Machine B
((1,[]),(λx.0),(λx.0))	(1,[]),(λx.0),(λx.0)
((2,[0]),(λx.0),(λx.0))	(2,[5]),(λx.0),(λx.0)
((3,[0;0]),(λx.0),(λx.0))	(3,[5]),(λx.0),(λx'.((2=x')=>5|0))
((4,[1]),(λx.0),(λx.0))	(4,[1;5]),(λx.0),(λx'.((2=x')=>5|0))
((1,[]),(λx.0),(λx.0))	(5,[1;5]),(λx.0), (λx'.((1=x')=>1|((2=x')=>5|0)))
((2,[1]),(λx.0),(λx.0))	(6,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((3,[0;1]),(λx.0),(λx.0))	(7,[0;0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((4,[0]),(λx.0),(λx.0))	(8,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((5,[]),(λx.0),(λx.0))	(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((6,[5]),(λx.0),(λx.0))	(6,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((7,[5]),(λx'.((1=x')=>5|0)),(λx.0))	(7,[0;0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((8,[1;5]),(λx'.((1=x')=>5|0)),(λx.0))	(8,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((9,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))	(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((10,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0))),	(6,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((11,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))	(7,[0;1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((12,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))	(8,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((13,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))	(9,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((14,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))	(10,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))
((15,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))	(11,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0)))
((16,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0)))	(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0)))
((17,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))	(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0)))
((18,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))	(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0)))
((0,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))	(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0)))

Figure 4: Animating a handshake.

```
⊢ Machines_Steps
22
([INP 1;OP0 ff;OP2 $==;JMN 1;INP 2;PUT 1;OP0 tt;OUT 1;SKP;SKP;SKP;OP0 ff;OUT 1;SKP;SKP;SKP;SKP],
[OP0 5;OUT 2;OP0 tt;OUT 1;INP 1;OP0 ff;OP2 $==;JMN 5;OP0 ff;OUT 1])
(((1,[]),(λx.0),(λx.0)),(1,[]),(λx.0),(λx.0)) =

((0,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0)),  (0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))
```

Using the compilation conversions this program compiles to:

```
[SKP;SKP;SKP;SKP;SKP;INP 1;OPO ff;OP2 $==;JMN 6;INP 2;PUT 1;OPO tt;OUT 1;
 SKP;SKP;SKP;OPO ff;OUT 1;SKP;SKP;SKP;SKP;INP 1;OPO ff;OP2 $==;JMN 23;
 INP 2;PUT 1;OPO tt;OUT 1;SKP;SKP;SKP;OPO ff;OUT 1;SKP;SKP;SKP;SKP],
[OPO 5;OUT 2;OPO tt;OUT 1;INP 1;OPO ff;OP2 $==;JMN 5;OPO ff;OUT 1]
```

The program was chosen specifically because, according to the machine defined earlier, the execution of the 8th instruction of the first input command coincides with the 5th instruction of the output command—recall that when this occurs, the loop in the output process that is meant to detect the acknowledgement from the input just misses the acknowledgement. This example tests whether the number of delay steps used at the end of the first input command suffices to ensure that the output command has enough time, in the worst case scenario, to disable the ready link of channel 'ch' before the second input on that channel is attempted. Otherwise both inputs can be consumed by the single output. In other words, we ask whether the compiled code of `Check_Handshake` first handshakes and then enters an infinite loop where the second input waits for an output (which of course doesn't occur). The animation of the code generated by the compilation algorithm presented earlier is shown in Fig.5; the last five steps of the animation shown in the figure illustrate that the code generated for `Check_Handshake` behaves as expected.

Now suppose that an oversight in the definition of the compiler causes the compiled code of the first input to delay for three, rather than four, instruction steps after the acknowledgement link is disabled (i.e. suppose that one of the SKP instructions underlined above is missing). In this case, the execution of the code of the second input command begins before the output command disables the ready link, so that the second input command erroneously detects that there is a second output willing to handshake with it. Consequently, the two input commands are consumed by the single output. This scenario is confirmed by the animated trace of execution shown in Fig.6. Notice that contrary to what is expected from the OC program `Check_Handshake`, the animated execution of the code generated by this faulty compilation would terminate and the following theorem would result:

```
⊢ Machines_Steps 38
  ([SKP;SKP;SKP;SKP;SKP;INP 1;OPO ff;OP2 $==;JMN 6;INP 2;PUT 1;OPO tt;
    OUT 1;SKP;SKP;SKP;OPO ff;OUT 1;SKP;SKP;SKP;INP 1;OPO ff;OP2 $==;
    JMN 22;INP 2;PUT 1;OPO tt;OUT 1;SKP;SKP;SKP;OPO ff;OUT 1;SKP;SKP;SKP],
   [OPO 5;OUT 2;OPO tt;OUT 1;INP 1;OPO ff;OP2 $==;JMN 5;OPO ff;OUT 1])
  (((1,[]),(λx.0),(λx.0)),(1,[]),(λx.0),(λx.0)) =
  ((0,[0;1;5;0;1;5]),(λx'.((1=x')=>5|0),(λx'.0),
   (0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))))
```

This example shows how the animation of compilation and execution gives vital feedback about the consequences of one's definitions. Moreover, since the animation results from the formal manipulation of logical terms using the HOL proof assistant, we have the added assurance that the outcome of the animation is backed up by formal proof.

<table>
<tr><th>Machine A</th><th>Machine B</th></tr>
<tr><td>((1,[]),(λx.0),(λx.0))</td><td>(1,[]),(λx.0),(λx.0)</td></tr>
<tr><td>((2,[]),(λx.0),(λx.0))</td><td>(2,[5]),(λx.0),(λx.0)</td></tr>
<tr><td>((3,[]),(λx.0),(λx.0))</td><td>(3,[5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((4,[]),(λx.0),(λx.0))</td><td>(4,[1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((5,[]),(λx.0),(λx.0))</td><td>(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((6,[]),(λx.0),(λx.0))</td><td>(6,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((7,[1]),(λx.0),(λx.0))</td><td>(7,[0;0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((8,[0;1]),(λx.0),(λx.0))</td><td>(8,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((9,[0]),(λx.0),(λx.0))</td><td>(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((10,[]),(λx.0),(λx.0))</td><td>(6,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((11,[5]),(λx.0),(λx.0))</td><td>(7,[0;0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((12,[5]),(λx'.((1=x')=>5|0)),(λx.0))</td><td>(8,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((13,[1;5]),(λx'.((1=x')=>5|0)),(λx.0))</td><td>(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((14,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(6,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((15,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(7,[0;0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((16,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(8,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((17,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((18,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(6,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((19,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(7,[0;1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((20,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(8,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((21,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(9,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((22,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(10,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((23,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(11,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((24,[0;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((25,[0;0;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((26,[1;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((23,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((24,[0;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((25,[0;0;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((26,[1;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((23,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>etc....etc.</td><td>etc....etc.</td></tr>
</table>

Figure 5: Two inputs are not consumed by a single output.

<table>
<tr><th>Machine A</th><th>Machine B</th></tr>
<tr><td>((1,[]),(λx.0),(λx.0))</td><td>(1,[]),(λx.0),(λx.0)</td></tr>
<tr><td>((2,[]),(λx.0),(λx.0))</td><td>(2,[5]),(λx.0),(λx.0)</td></tr>
<tr><td>((3,[]),(λx.0),(λx.0))</td><td>(3,[5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((4,[]),(λx.0),(λx.0))</td><td>(4,[1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((5,[]),(λx.0),(λx.0))</td><td>(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((6,[]),(λx.0),(λx.0))</td><td>(6,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((7,[1]),(λx.0),(λx.0))</td><td>(7,[0;0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((8,[0;1]),(λx.0),(λx.0))</td><td>(8,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((9,[0]),(λx.0),(λx.0))</td><td>(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((10,[]),(λx.0),(λx.0))</td><td>(6,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((11,[5]),(λx.0),(λx.0))</td><td>(7,[0;0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((12,[5]),(λx'.((1=x')=>5|0)),(λx.0))</td><td>(8,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((13,[1;5]),(λx'.((1=x')=>5|0)),(λx.0))</td><td>(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((14,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(6,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((15,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(7,[0;0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((16,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(8,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((17,[1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(5,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((18,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(6,[1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((19,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(7,[0;1;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((20,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(8,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((21,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(9,[1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((22,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(10,[0;1;5]),(λx.0),(λx'.((1=x')=>1|((2=x')=>5|0)))</td></tr>
<tr><td>((23,[1;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(11,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((24,[0;1;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((25,[0;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((26,[0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((27,[5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((28,[5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((29,[1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((30,[1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((31,[1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((32,[1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((33,[1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((34,[0;1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.((1=x')=>1|0)))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((35,[0;1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((36,[0;1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((37,[0;1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((38,[0;1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
<tr><td>((0,[0;1;5;0;1;5]),(λx'.((1=x')=>5|0)),(λx'.0))</td><td>(0,[0;1;5]),(λx.0),(λx'.((2=x')=>5|0))</td></tr>
</table>

Figure 6: Two inputs are erroneously consumed by one output.

5 An Operational Semantics for OC

One can argue that the formal definition of the behaviour of a machine can act as an operational semantics for a high-level programming language such as OC as long as there is a formal mapping between the constructs of the language and the instructions that drive the machine. In this sense, we have already presented an operational semantics of OC programs. In this section, however, we define an alternative operational semantics as a transition system in the style of Plotkin [Plo81]. We define a transition relation inductively as the least relation closed under a set of production rules that state how the behaviour of a construct of OC can be deduced from the behaviour of its components.

5.1 Inductively defined relations in HOL

Inductive definitions are based on the concept of a relation being closed under a set of rules. Since rules are essentially implications—*if* the premisses and side conditions hold, *then* the conclusion holds—it is straightforward to express this concept in logic [Mel91]. For example, let Even be the least relation for which the following deduction rules hold.

$$\textbf{E1}\ \frac{}{\text{Even } 0} \qquad\qquad \textbf{E2}\ \frac{\text{Even } n}{\text{Even } (n+2)}$$

These rules state precisely the properties required of the relation Even. Rule **E1** states that it must contain 0; and rule **E2** states that if n is in Even then so is $n+2$. The relation Even may therefore simply be *defined* to be the least relation that satisfies these conditions. It then follows simply by definition that the rules **E1** and **E2** are satisfied by Even. Moreover, it follows immediately that Even is a subset of any other relation that satisfies these rules, since Even is defined to be the *least* such relation. This means that Even contains only those values that it must contain by virtue of satisfying the rules.

The following formula asserts that a relation P:num is closed under the rules for Even:

$$(P\ 0)\ \wedge\ (\forall n.\ P\ n \supset P\ (n+2))$$

Therefore, the assertion that Even is the least such relation is expressed as the term:

$$\begin{aligned}\vdash\ &\forall n.\ \text{Even}\ n\ =\\ &\quad \forall P.\ ((P\ 0)\ \wedge\ (\forall n.\ P\ n \supset P\ (n+2)) \supset P\ n\end{aligned}$$

This definition gives rise to an induction principle for reasoning about the relation Even. This principle of *rule induction* is essential for many proofs involving such relations. (The term 'rule induction' was coined by Glynn Winskel in [Win85]). For example, to prove that a property P holds of the relation Even it suffices to show that the set $S = \{n \mid P\ n\}$ is closed under the rules defining Even. Since Even is defined to be the least such set then Even is a subset or equal to S and therefore satisfies the property P.

The automation of a definitional mechanism for inductively defined relations in HOL and the use of the principle of rule induction in proofs of properties of such relations is discussed extensively in [Mel91] and [CM].

5.2 The transition semantics of OC

5.2.1 The Semantic Domains

Let `store` be an alias for the type `mry: address->val` presented in section 2 and let `envr` and `lnks` be domains of type `(name->address)#address`. For `envr`, the right operand of the type constructor `#` denotes the next free address in local memory, and for `lnks` it denotes the next free link. As mentioned earlier, channels are directed and hence an enumerated type `chan_type` with objects `ab` and `ba` is defined to help distinguish between channels that allow data-flow from Machine A to B and those that allow data to flow in the opposite direction. A channel, moreover, is implemented as a triple of links—the ready, data and acknowledge links—which are declared on Machine A or Machine B depending on the direction of data-flow.

The domain of type `address#address#address` is named `channel` and denotes the addresses of the links that implement a channel. Let `cenv` be an alias for the type of channel environments `lnks#lnks#(name->chan_type)`; the two operands of type `lnks` denote the link environments of Machine A and B respectively, while the mapping from names to channel types is used to establish the whereabouts of the links that implement a channel. For example, if a channel *ch* is accessed, and according to the third operand of `cenv` the channel has type `ab`, then one ought to look for the value on the ready link `R`*ch* and that of the data link `D`*ch* in Machine A (i.e. using the first operand of `cenv`) while the value on the acknowledge link `A`*ch* should be sought in Machine B (i.e. using the second operand of `cenv`). The converse holds if *ch* has type `ba`. The term `Links`, intended to specify the behaviour described above, is defined as follows:

```
⊢ ∀ce ch.
    Links ce ch =
    (let (ce0,ce1,ne) = ce in
     let (chr,chd,cha) = links_of_chan ch
     in
      ((ne ch = ab) =>
       (FST ce0 chr,FST ce0 chd,FST ce1 cha)  |
       (FST ce1 chr,FST ce1 chd,FST ce0 cha)))
```

5.2.2 The transition semantics

The operational semantics of `OC` commands is defined as a labelled transition system that determines the execution path between *command configurations*. A command configuration `config` is either *terminal* (`TT`) in which case it denotes the final store, or else it is *non-terminal* (`NT`) in which case it comprises a command and a store:

```
config ::=    TT store   |   NT acmds store
```

A label is either empty, or it denotes input or output.

```
label ::=    E   |   In channel val   |   Out channel val
```

The evaluation of `OC` expressions is a function `Exec` defined, as follows, by primitive recursion on the type `exps`:

$$\textbf{T1}\ \frac{}{\texttt{Trans}\ ce\ e\ (\texttt{NT Skip}\ s)\ \texttt{E}\ (\texttt{TT}\ s)}$$

$$\textbf{T2}\ \frac{}{\texttt{Trans}\ ce\ e\ (\texttt{NT Stop}\ s)\ \texttt{E}\ (\texttt{NT Stop}\ s)}$$

$$\textbf{T3}\ \frac{}{\texttt{Trans}\ ce\ e\ (\texttt{NT(Assign}\ X\ ex)\ s)\ \texttt{E}\ (\texttt{TT(assign}\ X\ s\ e\ v))}\ \texttt{Exec}\ ex\ e\ s = v$$

$$\textbf{T4}\ \frac{\texttt{Trans}\ ce\ e\ (\texttt{NT}\ c_0\ s)\ l\ conf}{\texttt{Trans}\ ce\ e\ (\texttt{NT(If}\ ex\ c_0\ c_1)\ s)\ l\ conf}\ \texttt{Exec}\ ex\ e\ s = \texttt{tt}$$

$$\textbf{T5}\ \frac{\texttt{Trans}\ ce\ e\ (\texttt{NT}\ c_1\ s)\ l\ conf}{\texttt{Trans}\ ce\ e\ (\texttt{NT(If}\ ex\ c_0\ c_1)\ s)\ l\ conf}\ \texttt{Exec}\ ex\ e\ s = \texttt{ff}$$

$$\textbf{T6}\ \frac{\texttt{Trans}\ ce\ e\ (\texttt{NT}\ c_0\ s)\ l\ (\texttt{NT}\ c_0'\ s')}{\texttt{Trans}\ ce\ e\ (\texttt{NT(Seq}\ c_0\ c_1)\ s)\ l\ (\texttt{NT(Seq}\ c_0'\ c_1)s')}$$

$$\textbf{T7}\ \frac{\texttt{Trans}\ ce\ e\ (\texttt{NT}\ c_0\ s)\ l\ (\texttt{TT}\ s')}{\texttt{Trans}\ ce\ e\ (\texttt{NT(Seq}\ c_0\ c_1)\ s)\ l\ (\texttt{NT}\ c_1\ s')}$$

$$\textbf{T8}\ \frac{\texttt{Trans}\ ce\ e\ (\texttt{NT}\ c\ s)\ l\ (\texttt{NT}\ c'\ s')}{\texttt{Trans}\ ce\ e\ (\texttt{NT(While}\ ex\ c)\ s)\ l\ (\texttt{NT(Seq}\ c'\texttt{(While}\ ex\ c))\ s')}\ \texttt{Exec}\ ex\ e\ s = \texttt{tt}$$

$$\textbf{T9}\ \frac{\texttt{Trans}\ ce\ e\ (\texttt{NT}\ c\ s)\ l\ (\texttt{TT}\ s')}{\texttt{Trans}\ ce\ e\ (\texttt{NT(While}\ ex\ c)\ s)\ l\ (\texttt{NT(While}\ ex\ c)\ s')}\ \texttt{Exec}\ ex\ e\ s = \texttt{tt}$$

$$\textbf{T10}\ \frac{}{\texttt{Trans}\ ce\ e\ (\texttt{NT(While}\ ex\ c)\ s)\ \texttt{E}\ (\texttt{TT}s)}\ \texttt{Exec}\ ex\ e\ s = \texttt{ff}$$

$$\textbf{T11}\ \frac{}{\texttt{Trans}\ ce\ e\ (\texttt{NT(Inpt}\ ch\ X)\ s)\ (\texttt{In(Links}\ ce\ ch)v)\ (\texttt{TT(assign}\ X\ s\ e\ v))}$$

$$\textbf{T12}\ \frac{}{\texttt{Trans}\ ce\ e\ (\texttt{NT(Outpt}\ ch\ ex)\ s)\ (\texttt{Out(Links}\ ce\ ch)v)\ (\texttt{TT}\ s)}\ \texttt{Exec}\ ex\ e\ s = v$$

$$\textbf{T13}\ \frac{\texttt{Trans}\ ce\ (\texttt{update}\ X\ e)\ (\texttt{NT}\ c\ s)\ l\ (\texttt{NT}\ c'\ s')}{\texttt{Trans}\ ce\ e\ (\texttt{NT(Blk(Dec}\ X)c)s)\ l\ (\texttt{NT(Blk(Dec}\ X)c')s')}$$

$$\textbf{T14}\ \frac{\texttt{Trans}\ ce\ (\texttt{update}\ X\ e)\ (\texttt{NT}\ c\ s)\ l\ (\texttt{TT}\ s')}{\texttt{Trans}\ ce\ e\ (\texttt{NT(Blk(Dec}\ X)c)s)\ l\ (\texttt{TT}\ s')}$$

Figure 7: The definition of the transition relation Trans

```
⊢ (∀X en s. Exec(Var X)en s = s(FST en X)) ∧
   (∀n en s. Exec(Const n)en s = n) ∧
   (∀op1 e en s. Exec(Unop op1 e)en s = op1(Exec e en s)) ∧
   (∀op2 e1 e2 en s.
     Exec(Binop op2 e1 e2)en s = op2(Exec e1 en s)(Exec e2 en s))
```

The transition relation `Trans: cenv->envr->config->label->config->bool` for commands is defined by the rules in Fig.7. Informally, a `Skip` operation does not affect the store, while a `Stop` construct regenerates itself in any store. An assignment statement $X := e$ augments the value of X in the current store with the result of evaluating e; the function `assign` in the assignment rule does precisely this. The rules for the `If` construct show which alternative command is executed depending on whether the expression evaluates to `0` or `1`; note that `ff` and `tt` are abbreviations for `0` and `1` respectively. The rules for sequential composition state that the execution of `Seq` c_0 c_1 in store s proceeds by executing constructs in c_0 until a terminal store s' is reached, then c_1 is executed starting from s'. The rules for `While` systematically unfold a while loop into a series of sequential statements until the boolean condition is `ff`. The rule for an input construct of the form `Inpt` ch X states that if a value v (indicated by the input label) is input on channel ch then the value of X in the current store is augmented to v. On the other hand, the rule for an output construct of the form `Outpt` ch e states that if e evaluates to v then the value v is output on channel ch (this is indicated by the output label). Finally, the rules for block constructs show how the environment is augmented by declarations.

The relation `Trans` is defined using the HOL package that automates the derived principle of inductively defined relations. The details of the package are omitted, but see [Mel91] and [CM] for more information. The result of the definition are the following theorems asserting that `Trans` is closed under the rules:

```
Trans_rules =
[⊢ ∀ce e s. Trans ce e(NT Skip s)E(TT s);
 ⊢ ∀ce e s. Trans ce e(NT Stop s)E(NT Stop s);
 ⊢ ∀ex e s v.
     (Exec ex e s = v) ⊃
     (∀ce X. Trans ce e(NT(Assign X ex)s)E(TT(assign X s e v)));
 ⊢ ∀ce e c0 s 1 conf ex.
     Trans ce e(NT c0 s)1 conf ∧ (Exec ex e s = tt) ⊃
     (∀c1. Trans ce e(NT(If ex c0 c1)s)1 conf);
 ⊢ ∀ce e c1 s 1 conf ex.
     Trans ce e(NT c1 s)1 conf ∧ (Exec ex e s = ff) ⊃
     (∀c0. Trans ce e(NT(If ex c0 c1)s)1 conf);
 ⊢ ∀ce e c0 s 1 c0' s'.
     Trans ce e(NT c0 s)1(NT c0' s') ⊃
     (∀c1. Trans ce e(NT(Seq c0 c1)s)1(NT(Seq c0' c1)s'));
 ⊢ ∀ce e c0 s 1 s'.
     Trans ce e(NT c0 s)1(TT s') ⊃
     (∀c1. Trans ce e(NT(Seq c0 c1)s)1(NT c1 s'));
 ⊢ ∀ce e c s 1 c' s' ex.
     Trans ce e(NT c s)1(NT c' s') ∧ (Exec ex e s = tt) ⊃
     Trans ce e(NT(While ex c)s)1(NT(Seq c'(While ex c))s');
 ⊢ ∀ce e c s 1 s' ex.
     Trans ce e(NT c s)1(TT s') ∧ (Exec ex e s = tt) ⊃
     Trans ce e(NT(While ex c)s)1(NT(While ex c)s');
```

```
⊢ ∀ex e s.
    (Exec ex e s = ff) ⊃ (∀ce c. Trans ce e(NT(While ex c)s)E(TT s));
⊢ ∀ce e ch X s v.
    Trans ce e(NT(Inpt ch X)s)(In(Links ce ch)v)(TT(assign X s e v));
⊢ ∀ex e s v.
    (Exec ex e s = v) ⊃
    (∀ce ch. Trans ce e(NT(Outpt ch ex)s)(Out(Links ce ch)v)(TT s));
⊢ ∀ce X e c s l c' s'.
    Trans ce(update X e)(NT c s)l(NT c' s') ⊃
    Trans ce e(NT(Blk(Dec X)c)s)l(NT(Blk(Dec X)c')s');
⊢ ∀ce X e c s l s'.
    Trans ce(update X e)(NT c s)l(TT s') ⊃
    Trans ce e(NT(Blk(Dec X)c)s)l(TT s')]
```

together with a theorem asserting that it is the least such relation:

```
Trans_ind =
⊢ ∀P.
    (∀ce e s. P ce e(NT Skip s)E(TT s)) ∧
    (∀ce e s. P ce e(NT Stop s)E(NT Stop s)) ∧
    (∀ex e s v.
      (Exec ex e s = v) ⊃
      (∀ce X. P ce e(NT(Assign X ex)s)E(TT(assign X s e v)))) ∧
    (∀ce e c0 s l conf ex.
      P ce e(NT c0 s)l conf ∧ (Exec ex e s = tt) ⊃
      (∀c1. P ce e(NT(If ex c0 c1)s)l conf)) ∧
    (∀ce e c1 s l conf ex.
      P ce e(NT c1 s)l conf ∧ (Exec ex e s = ff) ⊃
      (∀c0. P ce e(NT(If ex c0 c1)s)l conf)) ∧
    (∀ce e c0 s l c0' s'.
      P ce e(NT c0 s)l(NT c0' s') ⊃
      (∀c1. P ce e(NT(Seq c0 c1)s)l(NT(Seq c0' c1)s'))) ∧
    (∀ce e c0 s l s'.
      P ce e(NT c0 s)l(TT s') ⊃
      (∀c1. P ce e(NT(Seq c0 c1)s)l(NT c1 s'))) ∧
    (∀ce e c s l c' s' ex.
      P ce e(NT c s)l(NT c' s') ∧ (Exec ex e s = tt) ⊃
      P ce e(NT(While ex c)s)l(NT(Seq c'(While ex c))s')) ∧
    (∀ce e c s l s' ex.
      P ce e(NT c s)l(TT s') ∧ (Exec ex e s = tt) ⊃
      P ce e(NT(While ex c)s)l(NT(While ex c)s')) ∧
    (∀ex e s.
      (Exec ex e s = ff) ⊃ (∀ce c. P ce e(NT(While ex c)s)E(TT s))) ∧
    (∀ce e ch X s v.
      P ce e(NT(Inpt ch X)s)(In(Links ce ch)v)(TT(assign X s e v))) ∧
    (∀ex e s v.
      (Exec ex e s = v) ⊃
      (∀ce ch. P ce e(NT(Outpt ch ex)s)(Out(Links ce ch)v)(TT s))) ∧
    (∀ce X e c s l c' s'.
      P ce(update X e)(NT c s)l(NT c' s') ⊃
      P ce e(NT(Blk(Dec X)c)s)l(NT(Blk(Dec X)c')s')) ∧
    (∀ce X e c s l s'.
      P ce(update X e)(NT c s)l(TT s') ⊃
      P ce e(NT(Blk(Dec X)c)s)l(TT s')) ⊃
    (∀ce e c0 l c1. Trans ce e c0 l c1 ⊃ P ce e c0 l c1)
```

Hitherto, we have presented a transition system that defines the behaviour of OC commands, but, we still have to define the interaction of OC commands in programs of the form `Par` c_0 c_1. As is the case for commands, a program configuration `pconfig` can be either *terminal* (`PT`) or *non-terminal* (`PN`):

```
pconfig ::=   PT store store   |   PN progs store store
```

Let `PTrans` be a relation of type:

```
PTrans : cenv->envr->envr->pconfig->pconfig->bool
```

defined inductively by the rules shown in Fig.8. The definition of `Final` (used in Fig.8) is given by the following theorem[9] of higher-order logic, which allows us to express a class of rules as a single one:

```
⊢ ∀a b.
    Final a b =
    (Is_TT a =>
     (Is_TT b =>
      PT(Str_of a)(Str_of b) |
      PN(Par Skip(Com_of b))(Str_of a)(Str_of b)) |
     (Is_TT b =>
      PN(Par(Com_of a)Skip)(Str_of a)(Str_of b) |
      PN(Par(Com_of a)(Com_of b))(Str_of a)(Str_of b)))
```

The first rule of Fig.8 states that if c_0 inputs on channel *ch* and yields the configuration $conf_0$ and c_1 outputs on the same channel to become $conf_1$, then the parallel composition of c_0 and c_1 yields `Final` $conf_0$ $conf_1$. The second rule deals with the dual case—i.e. when c_0 outputs and c_1 inputs. The third rule states that if both c_0 and c_1 perform a transition that does not involve inputs or outputs to yield $conf_0$ and $conf_1$ respectively, then `Par` c_0 c_1 yields `Final` $conf_0$ $conf_1$ without any exchange of data. The last two rules deal with the declaration of channels. For example, the outcome of executing a step of `Chan(AB` X`)` P is the same as the outcome of a step of P after the channel environment has been augmented (using `new_abschan`) with the declaration of channel X of type `ab`. Similarly for channels of type `BA`. The definition of `new_abschan` is the following theorem of higher-order logic:

```
⊢ ∀s ce ch.
    new_abschan s ce ch =
    (let (ce0,ce1,ne) = ce
     in
      let (chr,chd,cha) = links_of_chan ch
      in
       ((s = ab) =>
        (update chd(update chr ce0),update cha ce1,record ne ch ab) |
        (update cha ce0,update chd(update chr ce1),record ne ch ba)))
```

[9]Note that `Is_TT` specifies a function that returns true when the configuration passed as argument is terminal and false otherwise.

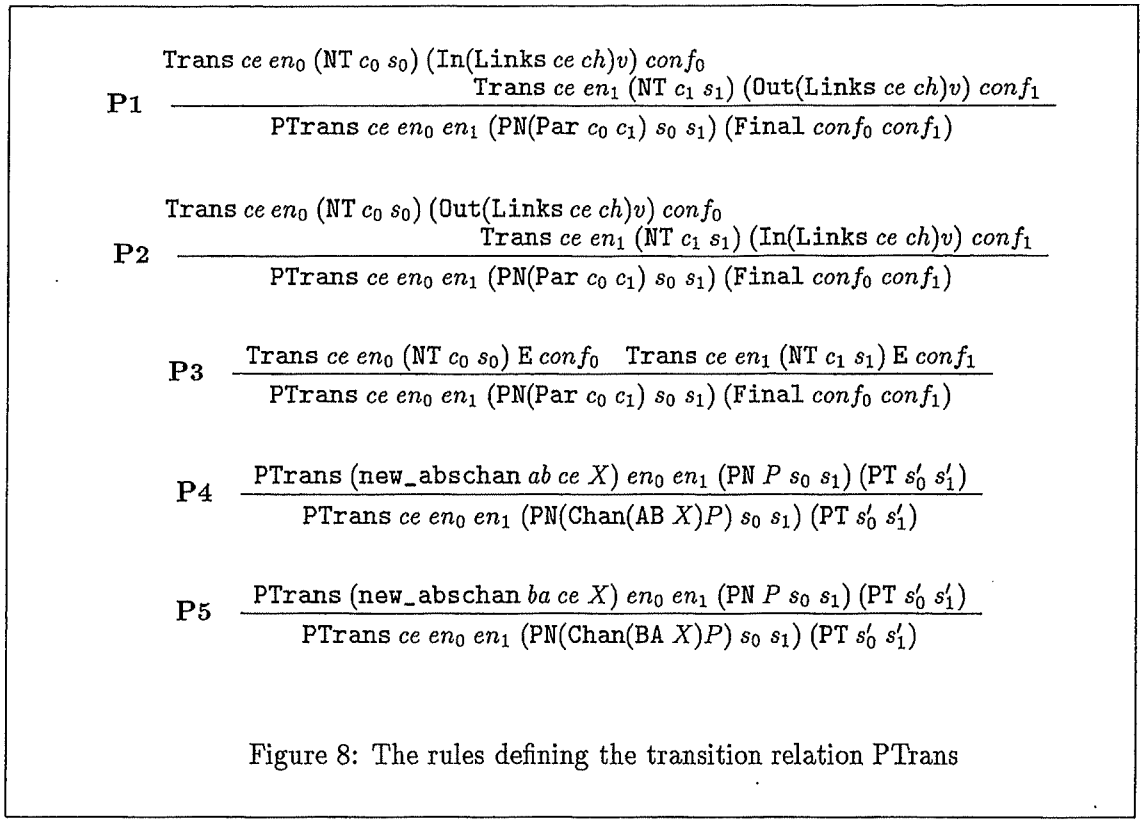


Figure 8: The rules defining the transition relation PTrans

Defining PTrans in HOL by the rules in Fig.8 results in the following automatically generated theorems asserting that the relation is closed under the rules:

```
PTrans_rules =
[⊢ ∀ce en0 c0 s0 conf0 en1 c1 s1 conf1.
     (∃ch v.
       Trans ce en0(NT c0 s0)(In(Links ce ch)v)conf0 ∧
       Trans ce en1(NT c1 s1)(Out(Links ce ch)v)conf1) ⊃
     PTrans ce en0 en1(PN(Par c0 c1)s0 s1)(Final conf0 conf1);
 ⊢ ∀ce en0 c0 s0 conf0 en1 c1 s1 conf1.
     (∃ch v.
       Trans ce en0(NT c0 s0)(Out(Links ce ch)v)conf0 ∧
       Trans ce en1(NT c1 s1)(In(Links ce ch)v)conf1) ⊃
     PTrans ce en0 en1(PN(Par c0 c1)s0 s1)(Final conf0 conf1);
 ⊢ ∀ce en0 c0 s0 conf0 en1 c1 s1 conf1.
     Trans ce en0(NT c0 s0)E conf0 ∧ Trans ce en1(NT c1 s1)E conf1 ⊃
     PTrans ce en0 en1(PN(Par c0 c1)s0 s1)(Final conf0 conf1);
 ⊢ ∀ce X en0 en1 P s0 s1 s0' s1'.
     PTrans(new_abschan ab ce X)en0 en1(PN P s0 s1)(PT s0' s1') ⊃
     PTrans ce en0 en1(PN(Chan(AB X)P)s0 s1)(PT s0' s1');
 ⊢ ∀ce X en0 en1 P s0 s1 s0' s1'.
     PTrans(new_abschan ba ce X)en0 en1(PN P s0 s1)(PT s0' s1') ⊃
     PTrans ce en0 en1(PN(Chan(BA X)P)s0 s1)(PT s0' s1')]
```

together with a theorem stating that it is the least such relation:

```
PTrans_ind =
⊢ ∀P'.
    (∀ce en0 c0 s0 conf0 en1 c1 s1 conf1.
      (∃ch v.
        Trans ce en0(NT c0 s0)(In(Links ce ch)v)conf0 ∧
        Trans ce en1(NT c1 s1)(Out(Links ce ch)v)conf1) ⊃
      P' ce en0 en1(PN(Par c0 c1)s0 s1)(Final conf0 conf1)) ∧
    (∀ce en0 c0 s0 conf0 en1 c1 s1 conf1.
      (∃ch v.
        Trans ce en0(NT c0 s0)(Out(Links ce ch)v)conf0 ∧
        Trans ce en1(NT c1 s1)(In(Links ce ch)v)conf1) ⊃
      P' ce en0 en1(PN(Par c0 c1)s0 s1)(Final conf0 conf1)) ∧
    (∀ce en0 c0 s0 conf0 en1 c1 s1 conf1.
      Trans ce en0(NT c0 s0)E conf0 ∧ Trans ce en1(NT c1 s1)E conf1 ⊃
      P' ce en0 en1(PN(Par c0 c1)s0 s1)(Final conf0 conf1)) ∧
    (∀ce X en0 en1 P s0 s1 s0' s1'.
      P'(new_abschan ab ce X)en0 en1(PN P s0 s1)(PT s0' s1') ⊃
      P' ce en0 en1(PN(Chan(AB X)P)s0 s1)(PT s0' s1')) ∧
    (∀ce X en0 en1 P s0 s1 s0' s1'.
      P'(new_abschan ba ce X)en0 en1(PN P s0 s1)(PT s0' s1') ⊃
      P' ce en0 en1(PN(Chan(BA X)P)s0 s1)(PT s0' s1')) ⊃
    (∀ce en0 en1 p p'. PTrans ce en0 en1 p p' ⊃ P' ce en0 en1 p p')
```

This completes the operational semantics of OC. We will, however, define a final transition relation

```
Execute : cenv->envr->envr->pconfig->pconfig->bool
```

which represents the semantics to completion of OC programs. The relation is defined by the rules shown in Fig.9.

6 Conclusion and Future Work

As stated in the introduction, the ultimate goal of this work is to verify the OC compiler. The goal is expressed as the following term of higher-order logic:

```
"∀ce en0 en1 p0 p1.
  Execute ce en0 en1 p0 p1 ⊃
  (∀c s0 s1.
    (PN c s0 s1 = p0) ⊃
    (∀s0' s1'.
      (PT s0' s1' = p1) ⊃
      (∀i s l0 l1.
        (i =
         Safe_Compile
         (Oc_Compile c)
         (((λx. 0),0),(λx. 0),0)
         (((λx. 0),0),(λx. 0),0)) ∧
        (s = ((1,[]),s0,l0),(1,[]),s1,l1) ⊃
        (∃n st0' st1' l0' l1'.
          Machines_Steps n i s = ((0,st0'),s0',l0'),(0,st1'),s1',l1'))))"
```

$$\textbf{E1}\quad \frac{\texttt{PTrans}\ ce\ en_0\ en_1\ p_0\ (\texttt{PT}\ s_0'\ s_1')}{\texttt{Execute}\ ce\ en_0\ en_1\ p_0\ (\texttt{PT}\ s_0'\ s_1')}$$

$$\textbf{E2}\quad \frac{\texttt{Execute}\ ce\ en_0\ en_1\ p_0\ p_0''\quad \texttt{PTrans}\ ce\ en_0\ en_1\ p_0''\ p_0'}{\texttt{Execute}\ ce\ en_0\ en_1\ p_0\ p_0'}$$

Figure 9: The rules defining the transition relation Execute

The proof should proceed by rule induction on the relation Execute; the subgoals generated will require nested rule inductions on the relations PTrans and Trans. Work on the compiler proof is in progress.

Formal animation, as described and used in this paper, yields theorems asserting how programs compile and execute. These theorems can be used to simplify and even prove subgoals generated in the verification of the compiler. Therefore we expect the same conversions used for animation and preliminary debugging of behavioural definitions to play a further role in the verification process.

7 Acknowledgements

I gratefully acknowledge the helpful comments and suggestions of Richard Boulton, Paul Curzon, Brian Graham, Tom Melham and John Van Tassel. Special thanks to Mike Gordon for his advice and support.

References

[Bar88] G. Barrett. The semantics and implementation of **occam**. DPhil thesis, Oxford University Computer Laboratory, 1988.

[BGHT90] R. Boulton, M. Gordon, J. Herbert, and J. Van Tassel. The HOL verification of ELLA designs. Technical Report 199, University of Cambridge Computer Laboratory, August 1990. Revised version in the Proceedings of the International Workshop on Formal Methods in VLSI Design, Miami, 1991.

[Cam88] Albert John Camilleri. *Executing Behavioural Definitions in Higher Order Logic.* PhD thesis, Computer Laboratory, University of Cambridge, July 1988.

[Cam89] Juanito Camilleri. An operational semantics for **occam**. *International Journal of Parallel Programming*, 18(5), October 1989.

[CM] Juanito Camilleri and T.F. Melham. Inductively defined relations in HOL. (to appear).

[Cur91] Paul Curzon. A verified compiler for a structured assembly language. In *Proceedings of the 1991 Internatiional Workshop on the HOL Theorem Proving System and its Applications.* IEEE Computer Society Press, 1991. To be published.

[Gor85] M.J.C. Gordon. HOL - a machine oriented formulation of higher order logic. Technical Report 68, Computer Laboratory, University of Cambridge, July 1985.

[Hal89] Roger Hale. Safe (version 0), November 1989. (unpublished note).

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* Wiley, 1990.

[inm84] inmos. **occam** *Programming Manual.* International Series in Computer Science. Prentice Hall, 1984.

[Joy89] Jeffrey J. Joyce. *Multi-Level Verification of Microprocessor-Based Systems.* PhD thesis, Computer Laboratory, University of Cambridge, December 1989. Report No. 195, Computer Laboratory, University of Cambridge, May 1990.

[Mel88] T.F. Melham. Automating Recursive type Definitions in Higher order logic. In *Current Trends in Hardware Verification and Automated Deduction.* Springer Verlag, 1988.

[Mel91] T.F. Melham. A package for inductive relation definitions in HOL. In *Proceedings of the 1991 Internatiional Workshop on the HOL Theorem Proving System and its Applications.* IEEE Computer Society Press, 1991. To be published.

[Mos85] Ben Moszkowski. Executing Temporal Logic Programs. Technical Report 71, University of Cambridge Computer Laboratory, August 1985.

[Pau] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(1983):119–149.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Department of Computer Science, Aarhus University Denmark, September 1981.

[Win85] G. Winskel. Introduction to the formal semantics of programming languages, October 1985. (unpublished lecture notes).