Formal Aspects of Computing (1999) 0: 000–000
© 1999 BCS

Formal Aspects of Computing

Formalization of Time and Space

Eric C.R. Hehner
Department of Computer Science, University of Toronto, Toronto, Canada

Keywords: Formal methods; Time bounds; Real-time programming; Space bounds; Average space

Abstract. Time and space limitations can be specified, and proven, in exactly the same way as functionality. Proofs of time bounds, both implementation-independent and real-time, and of space requirements, both worst-case and average-case, are given in complete detail.

1. Introduction

Time and space requirements of problems, and algorithms to solve them, have been studied under the name "computational complexity". Problems and algorithms are classified according to their time and space requirements expressed to within multiplicative and additive constants. See, for example, [Pap94]. Those studies do not look at individual programs, since their intent is to classify the complexity of an algorithm independent of any program to compute it, or to classify the complexity of a problem independent of any algorithm to solve it. In this paper we are concerned with the time and space requirements of programs, and with the multiplicative and additive constants that are of interest in real-time programming, though we also want to be able to abstract away from them when they are not critical. This paper proposes a formalism for the purpose.

Various formalisms have been proposed to account for the execution time of programs when time is critical; the most recent is [HaU98], and perhaps the earliest is [Sha79]; our formalism is very similar to those two. When space is critical, its analysis tends to be trivial: just look at the declarations; dynamic space allocation is avoided whenever possible. However, the formalization of time we use works, without change, for dynamic space requirements (but not for space requirements that depend on reference topology). The purpose of this paper is to demonstrate the formalism on a variety of spatial measures and program structures.

Correspondence and offprint requests to: Eric Hehner, Department of Computer Science, University of Toronto, Toronto ON M5S 3G4, Canada. e-mail: hehner@cs.utoronto.ca

In this formalism, we first decide what quantities are of interest, and introduce a variable for each such quantity. A specification is then a boolean expression whose variables represent the quantities of interest. The term “boolean expression” means an expression of type boolean, and is not meant to restrict the types of variables and subexpressions, nor the operators, within a specification. Quantifiers, functions, terms from the application domain, and terms invented for one particular specification are all welcome.

In a specification, some variables may represent inputs, and some may represent outputs. A specification is implemented on a computer when, for any values of the input variables, the computer generates (computes) values of the output variables to satisfy the specification. In other words, we have an implementation when the specification is true of every computation. (Note that we are specifying computations, not programs.) A program is a specification that has been implemented, so that a computer can execute it.

Suppose we are given specification S . If S is a program, a computer can execute it. If not, we have some programming to do. That means building a program P such that $S \Leftarrow P$ is a theorem; this is called refinement. Since S is implied by P , all computer behavior satisfying P also satisfies S . We might refine in steps, finding specifications R , Q , ... such that $S \Leftarrow R \Leftarrow Q \Leftarrow \ldots \Leftarrow P$.

If we are interested in time and space, we just introduce variables to stand for these quantities. These variables are “ghosts” in the sense that they are not part of the implementation; they do not occupy space, and assignments to them do not take time. Rather, they represent the space and time taken by other variables and assignments.

2. Notation

Here are all the notations used in this paper, arranged by precedence level.

0.	$0\ 1\ 2\ \infty\quad x\ y\quad (\ )$	numbers, variables, bracketed expressions
1.	$f(x)$	function application
2.	2^x	exponentiation
3.	$\times\ \ /\ \ \uparrow$	multiplication, division, maximum
4.	$+\ \ -$	addition, subtraction
5.	$=\ \ \neq\ \ <\ \ >\ \ \leq\ \ \geq$	comparisons
6.	$\neg$	negation
7.	$\wedge$	conjunction
8.	$\vee$	disjunction
9.	$\Rightarrow\ \ \Leftarrow$	implications
10.	$:=$ **if then else**	assignment, conditional
11.	$\forall\cdot\ \ \exists\cdot\ \ ;\ \ \|$	quantifiers, sequential composition, parallel composition
12.	$=\ \ \Longrightarrow\ \ \Longleftarrow$	equality, implications

Exponentiation serves to bracket all operations within the exponent. The multiplication sign $\times$ is sometimes omitted. The infix operators $/$ $-$ associate from left to right. The infix operators $\times\ \uparrow\ +\ \wedge\ \vee\ ;\ \|$ are associative (they associate in both directions). On levels 5, 9, and 12 the operators are continuing; for example, $a=b=c$ neither associates to the left nor associates to the right, but means $a=b \wedge b=c$. On any one of these levels, a mixture of continuing operators can be used. For example, $a \leq b < c$ means $a \leq b \wedge b < c$. The operators $=\ \Longrightarrow\ \Longleftarrow$ are identical to $=\ \Rightarrow\ \Leftarrow$ except for precedence.

3. Example: Exponentiation

To illustrate the formalism, we choose the simplest example we can find with nonconstant space requirements. The problem is specified as $y'=2^x$, where x and y are the initial values and x' and y' are the final values of two natural (nonnegative integer) variables. To make the space requirement nonconstant, we solve the problem using an internal (nontail) recursion. Here is the solution.

(0) $y'=2^x \;\Longleftarrow\; \mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (x:=x-1;\ y'=2^x;\ y:=2\times y)$

A compiler views the specification $y'=2^x$ as just an identifier, and (0) as a procedure definition. The procedure name is $y'=2^x$ and its body is $\mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (x:=x-1;\ y'=2^x;\ y:=2\times y)$. The second occurrence of the procedure name $y'=2^x$ is a recursive call. Execution of procedure $y'=2^x$ will result in a final value of y equal to 2 raised to the initial value of x .

A prover views the program parts as specifications, and (0) as a theorem to be proven. To see the program parts as specifications, we use the axioms

(1) $x:=e \;=\; x'=e \wedge y'=y \wedge \ldots$
(2) $\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q \;=\; b\wedge P \vee \neg b\wedge Q$
$\qquad\qquad\qquad\qquad\qquad\;=\; (b\Rightarrow P) \wedge (\neg b\Rightarrow Q)$
(3) $P;Q \;=\; \exists x'', y'', \ldots\cdot$ (for $x', y', \ldots$ substitute $x'', y'', \ldots$ in P)
$\qquad\qquad\qquad\qquad\qquad\wedge$ (for $x, y, \ldots$ substitute $x'', y'', \ldots$ in Q)

for assignment, conditional, and sequential composition. From these axioms, many useful laws can be proven, such as the substitution law:

(4) $x:=e;\ P \;=\;$ (for x substitute e in P)

Thus (0) can be proven as follows.

(5) $\mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (\underline{x:=x-1;\ y'=2^x};\ y:=2\times y)$ use substitution law (4)
$=\ \mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (y'=2^{x-1};\ \underline{y:=2\times y})$ use (1) to expand
$=\ \mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (\underline{y'=2^{x-1};\ x'=x \wedge y'=2\times y})$ use (3) to expand
$=\ \mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (\underline{\exists x'', y''\cdot y''=2^{x-1} \wedge x'=x'' \wedge y'=2\times y''})$ use one-point law
$=\ \mathbf{if}\ x=0\ \mathbf{then}\ \underline{y:=1}\ \mathbf{else}\ \underline{y'=2\times 2^{x-1}}$ use (1) to expand; simplify
$=\ \mathbf{if}\ x=0\ \mathbf{then}\ x'=x \wedge y'=1\ \mathbf{else}\ y'=2^x$ use (2) to expand
$=\ x=0 \wedge x'=x \wedge y'=1 \;\vee\; x\neq 0 \wedge y'=2^x$ logic and arithmetic
$\Longrightarrow\ y'=2^x$

Every step of this proof is trivial. With a small amount of practice, a human prover can take fewer, larger steps. An automated prover can perform the entire proof silently without help. For further explanation of this programming theory, please consult [Heh93]. So far, we have not considered any resource requirements — neither time nor space. (Termination is a time requirement; termination means that execution time is finite.)

4. Time

To talk about time, we just add a time variable t . We do not change the theory at all; the time variable is treated just like any other variable, as part of the state. The interpretation of t as time is justified by the way we use it. We use t for the initial time, the time at which execution starts, and t' for the final time, the time at which execution ends. To allow for the possibility of nontermination we take the domain of time to be a number system extended with an infinite number ∞ .

To obtain the real execution time, we take the domain of time to be the real numbers extended with ∞ , and we insert time increments $t := t+(\text{something})$ with all operations. Of course, this requires intimate knowledge of the implementation, both hardware and software; there is no way to avoid it if we want the real execution time. For many purposes, we can avoid the need to know implementation details by using a more abstract version of time. For example, we can consider that time is an extended integer, that a recursive call takes time 1 , and that all else takes time 0 . In that case, we place $t := t+1$ just before or after the recursive call. We can also say whatever we want about time in any specification. In our example, we can say

$$(6)\quad t'=t+x \;\Longleftarrow\; \mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (x:=x-1;\ t:=t+1;\ t'=t+x;\ y:=2\times y)$$

Here is the proof of (6).

$$\begin{array}{lll}
(7) & \mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (\underline{x:=x-1;\ t:=t+1;\ t'=t+x};\ y:=2\times y) & \text{use subst. law (4) twice} \\
= & \mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (\underline{t'=t+1+x-1;\ y:=2\times y}) & \text{simplify; use (1) to expand} \\
= & \mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (\underline{t'=t+x;\ x'=x \wedge y'=2\times y \wedge t'=t}) & \text{use (3) to expand} \\
= & \mathbf{if}\ x=0\ \mathbf{then}\ y:=1\ \mathbf{else}\ (\underline{\exists x'', y'', t''\cdot\ t''=t+x \wedge x'=x'' \wedge y'=2\times y'' \wedge t'=t''}) & \text{one-pt law} \\
= & \mathbf{if}\ x=0\ \mathbf{then}\ \underline{y:=1}\ \mathbf{else}\ t'=t+x & \text{use (1) to expand} \\
= & \mathbf{if}\ x=0\ \mathbf{then}\ x'=x \wedge y'=1 \wedge t'=t\ \mathbf{else}\ t'=t+x & \text{use (2) to expand} \\
= & x=0 \wedge x'=x \wedge y'=1 \wedge t'=t\ \vee\ x\neq 0 \wedge t'=t+x & \text{logic and arithmetic} \\
\Longrightarrow & t'=t+x &
\end{array}$$

By proving (6), we prove that the execution time is x in this abstract measure. Again here, and in all further proofs, the steps are completely trivial. Anyone experienced in this sort of proof can see the entire proof in one step. A proof about the execution time is no more difficult, and yields more information, than a proof about termination.

5. Space

To talk about space, we just add a space variable s . Like the time variable t , s is not part of the implementation, but only used in specifying and calculating space requirements. We use s for the space occupied initially at the start of execution, and s' for the space occupied finally at the end of execution. Once again, to allow for the possibility that execution endlessly consumes space, we take the domain of space to be the natural numbers extended with ∞ .

We cannot assume that the initial space occupied is 0 , just as we cannot assume that a computation begins at time 0 . Any program may be used as part of a larger program, and it may not be the first part. In our example, the program is called recursively within itself; the recursive invocation does not begin at the same time, nor with the same occupied space, as the nonrecursive invocation.

Wherever space is being increased, we insert $s:= s+$(the increase) to adjust s appropriately, and wherever space is being decreased, we insert $s:= s-$(the decrease). In our example, the only change to the occupied space occurs at the recursive call (which was the whole point of the example). At the beginning of the call, a return address is pushed onto a stack, increasing s by 1 (let us say). At the end of the call, the return addess is popped, decreasing s by 1 . Considering only space, ignoring time and results for a moment, we can prove

(8) $s'=s \;\Longleftarrow\; \textbf{if } x=0 \textbf{ then } y:= 1 \textbf{ else } (x:= x-1;\; s:= s+1;\; s'=s;\; s:= s-1;\; y:= 2\times y)$

which says that the space occupied is the same at the end as at the start. The proof:.

(9) $\textbf{if } x=0 \textbf{ then } y:= 1 \textbf{ else } (x:= x-1;\; s:= s+1;\; s'=s;\; s:= s-1;\; \underline{y:= 2\times y})$ use (1) to expand
$=$ $\textbf{if } x=0 \textbf{ then } y:= 1 \textbf{ else } (\underline{x:= x-1;\; s:= s+1;\; s'=s};\; \underline{s:= s-1;\; x'=x \wedge y'=2\times y \wedge s'=s})$
use substitution law (4) in two parts
$=$ $\textbf{if } x=0 \textbf{ then } y:= 1 \textbf{ else } (\underline{s'=s+1;\; s'=s-1})$ use (3) to expand
$=$ $\textbf{if } x=0 \textbf{ then } y:= 1 \textbf{ else } (\underline{\exists x'', y'', s''\cdot s''=s+1 \wedge s'=s''-1})$ use one-point law
$=$ $\textbf{if } x=0 \textbf{ then } \underline{y:= 1} \textbf{ else } s'=s$ use (1) to expand
$=$ $\textbf{if } x=0 \textbf{ then } x'=x \wedge y'=1 \wedge s'=s \textbf{ else } s'=s$ use (2) to expand
$=$ $x=0 \wedge x'=x \wedge y'=1 \wedge s'=s \;\vee\; x\neq 0 \wedge s'=s$ logic and arithmetic
$\Longrightarrow$ $s'=s$

Time monotonically increases during execution, and consequently the execution time of a program is the difference $t'-t$ between the time t when execution began and the time t' when it ends. Space, on the other hand, may increase and decrease during execution. The difference $s'-s$ does not tell us much about the space usage. There are two measures of interest: the maximum space occupied, and the average space occupied. We look first at maximum space.

6. Maximum Space

Let m be the maximum space occupied at the start of execution, and m' be the maximum space occupied by the end of execution. Wherever space is being increased, we insert $m:= m\uparrow s$ to keep m current, where $\uparrow$ is an infix maximum operator. There is no need to adjust m at a decrease in space. In our example, we can prove

(10) $m\ge s \;\Rightarrow\; m' = m\uparrow(s+x) \;\Longleftarrow$
$\textbf{if } x=0 \textbf{ then } y:= 1$
$\textbf{else } (x:= x-1;\; s:= s+1;\; m:= m\uparrow s;\; m\ge s \;\Rightarrow\; m' = m\uparrow(s+x);\; s:= s-1;\; y:= 2\times y)$

We want the specification to say that the maximum space is the starting space plus x more locations: $m' = s+x$. However, in a larger context, it may happen that m is already greater than $s+x$ at the start of this program fragment; so we must write $m' = m\uparrow(s+x)$. Furthermore, since m is supposed to be the maximum value of s , we assume $m\ge s$. In the body, we have placed $(s:= s+1;\; m:= m\uparrow s)$ just before the recursive call, and $s:= s-1$ just after.

Now here is the proof of (10). Using Axiom (2) and some boolean algebra, we can break the proof into two cases. First case:

(11) $x{=}0 \land (\underline{y{:=}\,1})$ use (1) to expand
$=$ $x{=}0 \land x'{=}x \land y'{=}1 \land s'{=}s \land m'{=}m$ logic and arithmetic
$\Longrightarrow$ $m{\ge}s \;\Rightarrow\; m' = m{\uparrow}(s{+}x)$

Second case:

(12) $x{\ne}0 \land (x{:=}\,x{-}1;\; s{:=}\,s{+}1;\; m{:=}\,m{\uparrow}s;\; m{\ge}s \;\Rightarrow\; m' = m{\uparrow}(s{+}x);\; s{:=}\,s{-}1;\; \underline{y{:=}\,2{\times}y})$
use (1) to expand
$=$ $x{\ne}0 \land (\,x{:=}\,x{-}1;\; s{:=}\,s{+}1;\; m{:=}\,m{\uparrow}s;\; m{\ge}s \;\Rightarrow\; m' = m{\uparrow}(s{+}x);$
$s{:=}\,s{-}1;\; \underline{x'{=}x} \land \underline{y'{=}2{\times}y} \land \underline{s'{=}s} \land m'{=}m\,)$ drop useless conjuncts
$\Longrightarrow$ $x{\ne}0 \land (\underline{x{:=}\,x{-}1;\; s{:=}\,s{+}1;\; m{:=}\,m{\uparrow}s;\; m{\ge}s \;\Rightarrow\; m' = m{\uparrow}(s{+}x)};\; \underline{s{:=}\,s{-}1;\; m'{=}m})$
use substitution law (4) in two sections
$=$ $x{\ne}0 \land (\underline{m{\uparrow}(s{+}1){\ge}s{+}1} \;\Rightarrow\; m' = m{\uparrow}(s{+}1){\uparrow}(\underline{s{+}1{+}x{-}1});\; m'{=}m)$ simplify
$=$ $x{\ne}0 \land (m' = m{\uparrow}(s{+}1){\uparrow}(s{+}x);\; m'{=}m)$ use (3) to expand
$=$ $x{\ne}0 \land (\exists m''\cdot\; m'' = m{\uparrow}(s{+}1){\uparrow}(s{+}x) \;\land\; m'{=}m'')$ use one-point law
$=$ $x{\ne}0 \land m' = m{\uparrow}(s{+}1){\uparrow}(s{+}x)$ logic and arithmetic
$\Longrightarrow$ $m{\ge}s \;\Rightarrow\; m' = m{\uparrow}(s{+}x)$

7. Average Space

To find the average space occupied, we find the cumulative space-time product, and then divide by the execution time. Let p be the cumulative space-time product at the start of execution, and p' be the cumulative space-time product at the end of execution. We still need variable s , which we adjust exactly as before. We no longer need variable t ; however, an increase in p occurs where there is an increase in t , and the increase is s times the increase in t . Here is the example.

```
(13)  $p' = p + sx + x(x{+}1)/2 \;\Longleftarrow$
          **if** $x{=}0$ **then** $y{:=}\,1$
          **else** $(x{:=}\,x{-}1;\; s{:=}\,s{+}1;\; p{:=}\,p{+}s;\; p' = p + sx + x(x{+}1)/2;\; s{:=}\,s{-}1;\; y{:=}\,2{\times}y)$
```

The specification $p' = p + sx + x(x{+}1)/2$ says that the space-time product is increased by two terms. The first one, sx , is the product of the initial space and the time taken by the computation, earlier proven to be x . The second one, $x(x{+}1)/2$, is due to the additional space required by this computation. Hence the average space occupied is $(x{+}1)/2$. Here is the proof, again in two cases.

(14) $x{=}0 \land (\underline{y{:=}\,1})$ use (1) to expand
$=$ $x{=}0 \land x'{=}x \land y'{=}1 \land s'{=}s \land p'{=}p$ logic and arithmetic
$\Longrightarrow$ $p' = p + sx + x(x{+}1)/2$

(15) $x{\ne}0 \land (x{:=}\,x{-}1;\; s{:=}\,s{+}1;\; p{:=}\,p{+}s;\; p' = p + sx + x(x{+}1)/2;\; s{:=}\,s{-}1;\; \underline{y{:=}\,2{\times}y})$
use (1) to expand
$=$ $\underline{x{\ne}0} \land (\,x{:=}\,x{-}1;\; s{:=}\,s{+}1;\; p{:=}\,p{+}s;\; p' = p + sx + x(x{+}1)/2;$
$s{:=}\,s{-}1;\; \underline{x'{=}x} \land \underline{y'{=}2{\times}y} \land \underline{s'{=}s} \land p'{=}p\,)$ drop useless conjuncts

$\Longrightarrow\ \ \underline{x{:=}\,x{-}1;\ \ s{:=}\,s{+}1;\ \ p{:=}\,p{+}s;\ \ p' = p + sx + x(x{+}1)/2};\ \ \underline{s{:=}\,s{-}1;\ \ p'{=}p}$
use substitution law (4) in two sections

$=\ \ p' = p + \underline{s + 1 + (s{+}1)(x{-}1) + (x{-}1)(x{-}1{+}1)/2};\ \ p'{=}p$ simplify

$=\ \ p' = p + sx + x(x{+}1)/2;\ \ p'{=}p$ use (3) to expand

$=\ \ \exists p''\cdot\ p'' = p + sx + x(x{+}1)/2\ \wedge\ p'{=}p''$ use one-point law

$=\ \ p' = p + sx + x(x{+}1)/2$

Having proven our refinement now for five separate specifications, no further proof is needed to put them all together. For free, we have

```
(16)  $y'{=}2^x \wedge t'{=}t{+}x \wedge s'{=}s \wedge (m{\ge}s \Rightarrow m' = m{\uparrow}(s{+}x)) \wedge p' = p + sx + x(x{+}1)/2 \ \Longleftarrow$
          if $x{=}0$ then $y{:=}\,1$
          else ( $x{:=}\,x{-}1;\ s{:=}\,s{+}1;\ t{:=}\,t{+}1;\ p{:=}\,p{+}s;$
                 $y'{=}2^x \wedge t'{=}t{+}x \wedge s'{=}s \wedge (m{\ge}s \Rightarrow m' = m{\uparrow}(s{+}x)) \wedge p' = p + sx + x(x{+}1)/2;$
                 $s{:=}\,s{-}1;\ y{:=}\,2{\times}y$ )
```

8. Example: Towers of Hanoi

The previous example used linear time and space. As a second example, we choose a computation with a different complexity: the well-known Towers of Hanoi. Let x be the number of disks. For performance purposes, the program is as follows:

```
(17)  MovePile ⟸  if $x{>}0$
                   then ($x{:=}\,x{-}1$;  MovePile;  $x{:=}\,x{+}1$;
                         MoveDisk;
                         $x{:=}\,x{-}1$;  MovePile;  $x{:=}\,x{+}1$)
                   else ok
```

To move the pile of disks, if there is at least one disk, first, ignore the bottom disk, move the remaining pile, then reconsider all disks; now move one disk (the one we were previously ignoring); then again ignore the bottom disk, move the remaining pile, then reconsider all disks. If there are no disks, do nothing. The “empty statement” *ok* says that all variables remain unchanged.

(18) $ok\ =\ x'{=}x \wedge y'{=}y \wedge \ldots$

We use a two-tailed **if** with an empty statement for the **else**-part in preference to a one-tailed **if** because the latter is syntactically ambiguous (which can be resolved by adding an explicit ending, but then it is no shorter than a two-tailed **if** with an empty **else**-part), and semantically misleading (one tends to forget that there are still two cases to consider).

Next we have to decide what to prove. We have not included enough details of the Towers of Hanoi to prove that the disks get moved to the right place; we have only a disk counter, and we can prove

(19) $x'{=}x$

which says that the number of disks ends as it began. To measure time, we add a time variable t . We suppose that *MoveDisk* takes time 1 , and that is all it does that we care about at the moment, so we replace it by $t:= t+1$. We replace *MovePile* with the specification

(20) $t:= t + 2^x - 1$

which says that x is unchanged and the execution time is $2^x - 1$. We need to prove

$$
\begin{aligned}
&(21)\quad t:= t + 2^x - 1 \;\Longleftarrow\; && \mathbf{if}\ x{>}0\\
& && \mathbf{then}\ (x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ x:= x{+}1;\\
& && \qquad t:= t{+}1;\\
& && \qquad x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ x:= x{+}1\ )\\
& && \mathbf{else}\ ok
\end{aligned}
$$

Using Axiom 2, we break the proof into two cases:

$$
\begin{aligned}
&(22)\quad t:= t + 2^x - 1 \;\Longleftarrow\; && x{>}0 \wedge (x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ x:= x{+}1;\\
& && \qquad t:= t{+}1;\\
& && \qquad x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ x:= x{+}1\ )
\end{aligned}
$$

(23) $t:= t + 2^x - 1 \;\Longleftarrow\; x{=}0 \wedge ok$

To prove (22) we start with its right side.

(24) $\underline{x{>}0} \wedge (x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ x:= x{+}1;\ \ t:= t{+}1;\ \ x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ \underline{x:= x{+}1})$

drop conjunct ; use (1) to expand

$\Longrightarrow\quad x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ x:= x{+}1;\ \ t:= t{+}1;\ \ x:= x{-}1;\ \ \underline{t:= t + 2^{\underline{x}} - 1;\ \ x'{=}x{+}1 \wedge t'{=}t}$

use substitution law (4) repeatedly from right to left

$=\quad x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ x:= x{+}1;\ \ t:= t{+}1;\ \ \underline{x:= x{-}1;\ \ x'{=}x{+}1 \wedge t'{=}t{+}2^{\underline{x}}\underline{-1}}$

$=\quad x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ x:= x{+}1;\ \ \underline{t:= t{+}1;\ \ x'{=}x \wedge t'{=}t{+}2^{\underline{x-1}}\underline{-1}}$

$=\quad x:= x{-}1;\ \ t:= t + 2^x - 1;\ \ \underline{x:= x{+}1;\ \ x'{=}x \wedge t'{=}t{+}2^{\underline{x-1}}}$

$=\quad x:= x{-}1;\ \ \underline{t:= t + 2^{\underline{x}} - 1;\ \ x'{=}x{+}1 \wedge t'{=}t{+}2^{\underline{x}}}$

$=\quad \underline{x:= x{-}1;\ \ x'{=}x{+}1 \wedge t'{=}t{+}2^{\underline{x}} -1{+}2^{\underline{x}}}$

$=\quad x'{=}x \wedge t'{=}t{+}\underline{2}^{\underline{x-1}}{-}1\underline{+2}^{\underline{x-1}}$ simpify

$=\quad x'{=}x \wedge t'{=}t{+}2^x{-}1$ use (1) to contract

$=\quad t:= t + 2^x - 1$

Proving (23) is easier.

(25) $x{=}0 \wedge ok$ use (18) to expand

$=\quad x{=}0 \wedge x'{=}x \wedge t'{=}t$ arithmetic

$\Longrightarrow\quad t:= t + 2^x - 1$

For the calculation of maximum space, we remove variable t , and add variables s (space) and m (maximum space). As before, let us suppose the recursive calls each cost one location (for the return address). We suppose that *MoveDisk* and the assignments do not require space. The maximum space in this example is the same as in the previous example. We prove

```
(26)  $m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x)) \ \Longleftarrow$
          **if** $x{>}0$
          **then** ( $x{:=}\ x{-}1;\ \ s{:=}\ s{+}1;\ \ m{:=}\ m{\uparrow}s;\ \ m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x));\ \ s{:=}\ s{-}1;\ \ x{:=}\ x{+}1;$
                 $ok;$
                 $x{:=}\ x{-}1;\ \ s{:=}\ s{+}1;\ \ m{:=}\ m{\uparrow}s;\ \ m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x));\ \ s{:=}\ s{-}1;\ \ x{:=}\ x{+}1$ )
          **else** $ok$
```

The specification $m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x))$ says that, if m is initially as large as s, then it will finally be the maximum of the initial values of m and $s{+}x$, and all other variables will be unchanged. In other words, the maximum space occupied by this computation is x. Before proving it, we simplify. First, the line within (26) that appears twice:

(27) $x{:=}\ x{-}1;\ \ s{:=}\ s{+}1;\ \ m{:=}\ m{\uparrow}s;\ \ m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x));\ \ s{:=}\ s{-}1;\ \ \underline{x{:=}\ x{+}1}$

use (1) to expand

$=$ $x{:=}x{-}1;\ s{:=}\ s{+}1;\ m{:=}\ m{\uparrow}s;\ m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x));\ \underline{s{:=}\ s{-}1;\ x'{=}x{+}1 \wedge s'{=}s \wedge m'{=}m}$

use substitution law (4) repeatedly from right to left

$=$ $x{:=}\ x{-}1;\ \ s{:=}\ s{+}1;\ \ m{:=}\ m{\uparrow}s;\ \ \underline{m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x));\ \ x'{=}x{+}1 \wedge s'{=}s{-}1 \wedge m'{=}m}$

$=$ $x{:=}\ x{-}1;\ \ s{:=}\ s{+}1;\ \ \underline{m{:=}\ m{\uparrow}s;\ \ m{\ge}s \Rightarrow x'{=}x{+}1 \wedge s'{=}s{-}1 \wedge m'{=}m{\uparrow}(s{+}x)}$

$=$ $x{:=}\ x{-}1;\ \ s{:=}\ s{+}1;\ \ \underline{m{\uparrow}s{\ge}s} \Rightarrow x'{=}x{+}1 \wedge s'{=}s{-}1 \wedge m'{=}m{\uparrow}s{\uparrow}(s{+}x)$ simplify

$=$ $x{:=}\ x{-}1;\ \ \underline{s{:=}\ s{+}1;\ \ x'{=}x{+}1 \wedge s'{=}s{-}1 \wedge m'{=}m{\uparrow}s{\uparrow}(s{+}x)}$ resume using subst law (4)

$=$ $\underline{x{:=}\ x{-}1;\ \ x'{=}x{+}1 \wedge s'{=}s{+}1{-}1 \wedge m'{=}m{\uparrow}(s{+}1){\uparrow}(s{+}1{+}x)}$

$=$ $x'{=}x\underline{{-}1{+}1} \wedge s'{=}s\underline{{+}1{-}1} \wedge m'{=}m{\uparrow}(s{+}1){\uparrow}(s\underline{{+}1}{+}x\underline{{-}1})$ simplify

$=$ $x'{=}x \wedge s'{=}s \wedge m'{=}m{\uparrow}(s{+}1){\uparrow}(s{+}x)$ use (1) to contract

$=$ $m{:=}\ m{\uparrow}(s{+}1){\uparrow}(s{+}x)$

The **then**-part becomes:

(28) $x{:=}\ x{-}1;\ \ s{:=}\ s{+}1;\ \ m{:=}\ m{\uparrow}s;\ \ m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x));\ \ s{:=}\ s{-}1;\ \ x{:=}\ x{+}1;$
$ok;$
$x{:=}\ x{-}1;\ \ s{:=}\ s{+}1;\ \ m{:=}\ m{\uparrow}s;\ \ m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x));\ \ s{:=}\ s{-}1;\ \ x{:=}\ x{+}1$

ok is identity for semi-colon; and use last two lines of (27)

$=$ $m{:=}\ m{\uparrow}(s{+}1){\uparrow}(s{+}x);\ \ x'{=}x \wedge s'{=}s \wedge m'{=}m{\uparrow}(s{+}1){\uparrow}(s{+}x)$ use substitution law (4)

$=$ $x'{=}x \wedge s'{=}s \wedge m'{=}m{\uparrow}(s{+}1){\uparrow}(s{+}x){\uparrow}(s{+}1){\uparrow}(s{+}x)$ simplify and use (1) to contract

$=$ $m{:=}\ m{\uparrow}(s{+}1){\uparrow}(s{+}x)$

Now the proof, in the usual two cases:

(29) $x{>}0 \wedge (m{:=}\ m{\uparrow}(s{+}1){\uparrow}(s{+}x))$ if $x{>}0$ then $s{+}x{\ge}s{+}1$

$=$ $x{>}0 \wedge (m{:=}\ m{\uparrow}(s{+}x))$ drop conjunct and add antecedent

$\Longrightarrow$ $m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x))$

(30) $x{=}0 \wedge ok$ use (18) to expand

$=$ $x{=}0 \wedge x'{=}x \wedge s'{=}s \wedge m'{=}m$ arithmetic

$\Longrightarrow$ $m{\ge}s \Rightarrow (m{:=}\ m{\uparrow}(s{+}x))$

The average space is quite different for this example than it was for the previous example. We need variables s (space) and p (space-time product). Where t was increased by 1, we now increase p by s. We prove

$$
\begin{array}{ll}
(31) & p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2 \ \Longleftarrow \\
 & \quad \textbf{if}\ x{>}0 \\
 & \quad \textbf{then}\ (\ x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ x{:=}\ x+1; \\
 & \qquad\qquad p{:=}\ p+s; \\
 & \qquad\qquad x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ x{:=}\ x+1\ ) \\
 & \quad \textbf{else}\ ok
\end{array}
$$

The specification $p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2$ says that p is increased by the product of the initial space s and total time $2^x - 1$, plus an additional amount $(x-2)2^x + 2$. The average space is this additional amount divided by the execution time. Thus the average space occupied by this computation is $x + x/(2^x - 1) - 2$. The proof, as always, in two parts:

$$
\begin{array}{lll}
(32) & \underline{x{>}0} \wedge (\ x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ x{:=}\ x+1; & \\
 & \qquad p{:=}\ p+s; & \\
 & \qquad x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ \underline{x{:=}\ x+1}\ ) & \\
 & & \text{drop conjunct; expand} \\
\Longrightarrow & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ x{:=}\ x+1;\ \ p{:=}\ p+s; & \\
 & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ \underline{s{:=}\ s-1};\ \ x'{=}x+1 \wedge s'{=}s \wedge p'{=}p & \\
 & & \text{repeatedly use substitution law (4) from right to left} \\
= & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ x{:=}\ x+1;\ \ p{:=}\ p+s; & \\
 & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ \underline{p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2};\ \ x'{=}x+1 \wedge s'{=}s-1 \wedge p'{=}p & \\
= & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ x{:=}\ x+1;\ \ p{:=}\ p+s; & \\
 & x{:=}\ x-1;\ \ \underline{s{:=}\ s+1};\ \ x'{=}x+1 \wedge s'{=}s-1 \wedge p' = p + s(2^x - 1) + (x-2)2^x + 2 & \\
= & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ x{:=}\ x+1;\ \ p{:=}\ p+s; & \\
 & \underline{x{:=}\ x-1};\ \ x'{=}x+1 \wedge s'{=}s \wedge p' = p + (s+1)(2^x - 1) + (x-2)2^x + 2 & \\
= & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ x{:=}\ x+1;\ \ \underline{p{:=}\ p+s}; & \\
 & x'{=}x \wedge s'{=}s \wedge p' = p + (s+1)(2^{x-1} - 1) + (x-3)2^{x-1} + 2 & \\
= & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ s{:=}\ s-1;\ \ \underline{x{:=}\ x+1}; & \\
 & x'{=}x \wedge s'{=}s \wedge p' = p + s + (s+1)(2^{x-1} - 1) + (x-3)2^{x-1} + 2 & \\
= & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2;\ \ \underline{s{:=}\ s-1}; & \\
 & x'{=}x+1 \wedge s'{=}s \wedge p' = p + s + (s+1)(2^x - 1) + (x-2)2^x + 2 & \\
= & x{:=}\ x-1;\ \ s{:=}\ s+1;\ \ \underline{p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2}; & \\
 & x'{=}x+1 \wedge s'{=}s-1 \wedge p' = p + s - 1 + s(2^x - 1) + (x-2)2^x + 2 & \\
= & x{:=}\ x-1;\ \ \underline{s{:=}\ s+1}; & \\
 & x'{=}x+1 \wedge s'{=}s-1 \wedge p' = p + s(2^x - 1) + (x-2)2^x + 2 + s - 1 + s(2^x - 1) + (x-2)2^x + 2 & \\
= & \underline{x{:=}\ x-1}; & \\
 & x'{=}x+1 \wedge s'{=}s \wedge p' = p + (s+1)(2^x-1) + (x-2)2^x + 2 + s + (s+1)(2^x-1) + (x-2)2^x + 2 & \\
= & x'{=}x \wedge s'{=}s \wedge p' = p\ \ + (s+1)(2^{x-1}-1) + (x-3)2^{x-1} + 2 + s + (s+1)(2^{x-1} - 1) & \\
 & \qquad\qquad + (x-3)2^{x-1} + 2 & \text{simplify} \\
= & x'{=}x \wedge s'{=}s \wedge p' = p + s(2^x - 1) + (x-2)2^x + 2 & \text{use (1) to contract} \\
= & p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2 &
\end{array}
$$

$$
\begin{array}{lll}
(33) & x{=}0 \wedge ok & \text{use (18) to expand} \\
= & x{=}0 \wedge x'{=}x \wedge s'{=}s \wedge p'{=}p & \text{arithmetic} \\
\Longrightarrow & p{:=}\ p + s(2^x - 1) + (x-2)2^x + 2 &
\end{array}
$$

Putting together all the proofs for the Towers of Hanoi problem, we have

```
(34)  MovePile  ⇐  if $x>0$
                   then ( $x:= x–1$;  $s:= s+1$;  $m:= m\uparrow s$;  MovePile;  $s:= s–1$;  $x:= x+1$;
                          $t:= t+1$;  $p:= p+s$;  ok;
                          $x:= x–1$;  $s:= s+1$;  $m:= m\uparrow s$;  MovePile;  $s:= s–1$;  $x:= x+1$ )
                   else ok
```

where *MovePile* is the specification

$$
\begin{array}{lll}
(35) & & x'=x \\
 & \wedge & t' = t + 2^x - 1 \\
 & \wedge & s'=s \\
 & \wedge & (m \ge s \Rightarrow m' = m \uparrow (s+x)) \\
 & \wedge & p' = p + s(2^x - 1) + (x-2)2^x + 2
\end{array}
$$

9. Approximate Hanoi

In our examples so far, we have proven exact execution time and space requirements (for an abstract measure of time and space). Sometimes it is easier and adequate to prove time and space bounds. For example, in the Towers of Hanoi problem, instead of proving that the average space is exactly $x + x/(2^x - 1) - 2$, it is easier to prove that the average space is bounded above by x . To do so, instead of proving that the space-time product is the complicated expression $s(2^x-1) + (x-2)2^x + 2$, we prove it is at most $(s+x)(2^x-1)$. We must prove

```
(36)  $x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s+x)(2^x-1)$  ⇐
         if $x>0$
         then ($x:= x–1$;  $s:= s+1$;  $x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s+x)(2^x-1)$;  $s:=s–1$; $x:=x+1$;
               $p:= p+$s;
               $x:= x–1$;  $s:= s+1$;  $x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s+x)(2^x-1)$; $s:=s–1$; $x:=x+1$ )
         else ok
```

The proof:

$$
\begin{array}{lll}
(37) & \underline{x>0} \wedge (x:= x-1;\ s:= s+1;\ x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s+x)(2^x - 1);\ s:= s-1;\ x:= x+1; & \\
 & \quad p:= p+s; & \\
 & \quad x:= x-1;\ s:= s+1;\ x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s+x)(2^x - 1);\ s:= s-1;\ x:=x+1\ ) & \\
 & & \text{drop conjunct; simplify} \\
\Longrightarrow & x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s\underline{+1}+x\underline{-1})(2^{x-1} - 1); & \\
 & \underline{p:= p+s}; & \\
 & x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + s(s\underline{+1}+x\underline{-1})(2^{x-1} - 1) & \text{simplify; substitution law (4)} \\
= & x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s+x)(2^{x-1} - 1); & \\
 & x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + s + (s+x)(2^{x-1} - 1) & \\
= & x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s+x)(2^{x-1} - 1) + s + (s+x)(2^{x-1} - 1) & \text{simplify} \\
= & x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s+x)(2^x - 1) - x & \\
\Longrightarrow & x'=x \ \wedge\ s'=s \ \wedge\ p' \le p + (s+x)(2^x - 1) &
\end{array}
$$

(38) $x{=}0 \wedge ok$ use (18) to expand
$=$ $x{=}0 \wedge x'{=}x \wedge s'{=}s \wedge p'{=}p$ arithmetic
$\Longrightarrow$ $x'{=}x \;\wedge\; s'{=}s \;\wedge\; p' \le p + (s{+}x)(2^x - 1)$

Lower bounds can be shown in a similar fashion.

10. Real-Time Hanoi

Let us suppose now that the output of the Towers of Hanoi program is the movement of a mechanical arm (to actually move the disks), and that we are interested in real-time performance rather than the abstract measure we have used so far. Suppose that the posts where the disks are placed are arranged in an equilateral triangle, so that the distance the arm moves each time is constant (one side of the triangle to get into position plus one side to move the disk), and not dependent on the disk being moved. Suppose the time to move a disk varies with the weight of the disk being moved, which varies with its area, which varies with the square of its radius, which varies with the disk number. For even more realism, suppose there is an uncertainty of ε in the time to move each disk. Then, for the purpose of calculating real-time, the specification *MoveDisk* is

(39) $x'{=}x \;\wedge\; t + ax^2 + bx + c - \varepsilon \le t' \le t + ax^2 + bx + c + \varepsilon$

for some constants a, b, and c, which we write more briefly as

(40) $t{:=}\ t + ax^2 + bx + c \pm \varepsilon$

Moving a disk does not change the number of disks, and it takes time $ax^2 + bx + c$ give or take ε. Then *MovePile* is

(41) $t{:=}\ t + (6a + 2b + c \pm \varepsilon)(2^x - 1) - ax^2 - (4a + b)x$

Using (40) and (41) in (17) we prove

```
(42)  $t{:=}\ t + (6a + 2b + c \pm \varepsilon)(2^x - 1) - ax^2 - (4a + b)x \;\Longleftarrow$
          **if** $x{>}0$
          **then** ($x{:=}\ x{-}1;\;\; t{:=}\ t + (6a + 2b + c \pm \varepsilon)(2^x - 1) - ax^2 - (4a + b)x;\;\; x{:=}\ x{+}1;$
                   $t{:=}\ t + ax^2 + bx + c \pm \varepsilon;$
                   $x{:=}\ x{-}1;\;\; t{:=}\ t + (6a + 2b + c \pm \varepsilon)(2^x - 1) - ax^2 - (4a + b)x;\;\; x{:=}\ x{+}1$ )
          **else** $ok$
```

in the usual two cases.

(43) $\underline{x{>}0} \wedge (\, x{:=}\ x{-}1;\;\; t{:=}\ t + (6a + 2b + c \pm \varepsilon)(2^x - 1) - ax^2 - (4a + b)x;\;\; x{:=}\ x{+}1;$
$\quad t{:=}\ t + ax^2 + bx + c \pm \varepsilon;$
$\quad x{:=}\ x{-}1;\;\; t{:=}\ t + (6a + 2b + c \pm \varepsilon)(2^x - 1) - ax^2 - (4a + b)x;\;\; \underline{x{:=}\ x{+}1}\ )$
drop conjunct; expand
$\Longrightarrow$ $x{:=}\ x{-}1;\;\; t{:=}\ t + (6a + 2b + c \pm \varepsilon)(2^x - 1) - ax^2 - (4a + b)x;\;\; x{:=}\ x{+}1;$
$t{:=}\ t + ax^2 + bx + c \pm \varepsilon;$
$x{:=}\ x{-}1;\;\; \underline{t{:=}\ t + (6a + 2b + c \pm \varepsilon)(2^x - 1) - ax^2 - (4a + b)x;}\;\; x'{=}x{+}1 \wedge t'{=}t$

repeatedly use substitution law (4) from right to left

$=\quad x:=x-1;\ \ t:=t+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x;\ \ x:=x+1;$

$\qquad t:=t+ax^2+bx+c\pm\varepsilon;$

$\qquad \underline{x:=x-1;}\ \ x'=x+1\wedge t'=t+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x$

$=\quad x:=x-1;\ \ t:=t+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x;\ \ x:=x+1;$

$\qquad \underline{t:=t+ax^2+bx+c\pm\varepsilon;}$

$\qquad x'=x\wedge t'=t+(6a+2b+c\pm\varepsilon)(2^{x-1}-1)-a(x-1)^2-(4a+b)(x-1)$

$=\quad x:=x-1;\ \ t:=t+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x;\ \ \underline{x:=x+1;}$

$\qquad x'=x\wedge t'=t+ax^2+bx+c\pm\varepsilon+(6a+2b+c\pm\varepsilon)(2^{x-1}-1)-a(x-1)^2-(4a+b)(x-1)$

$=\quad x:=x-1;\ \ \underline{t:=t+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x;}$

$\qquad x'=x+1\wedge t'=t+a(x+1)^2+b(x+1)+c\pm\varepsilon+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x$

$=\quad \underline{x:=x-1;}$

$\qquad x'=x+1\wedge t'=t+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x+a(x+1)^2+b(x+1)+c\pm\varepsilon$

$\qquad\qquad +(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x$

$=\quad x'=x\wedge t'=t\ +(6a+2b+c\pm\varepsilon)(2^{x-1}-1)-a(x-1)^2-(4a+b)(x-1)+ax^2+bx+c\pm\varepsilon$

$\qquad\qquad +(6a+2b+c\pm\varepsilon)(2^{x-1}-1)-a(x-1)^2-(4a+b)(x-1)$ simplify

$=\quad x'=x\wedge t'=t+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x$ use (1) to contract

$=\quad t:=t+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x$

(44) $x=0\wedge ok$ use (18) to expand

$=\quad x=0\wedge x'=x\wedge t'=t$ arithmetic

$\Longrightarrow\quad t:=t+(6a+2b+c\pm\varepsilon)(2^x-1)-ax^2-(4a+b)x$

11. Declaration and Allocation

In a block-structured language, variable and array declarations are accompanied by an increase to variable s, and the end of a block is preceded by a decrease to s. In a language with explicit **alloc**ation and **free** commands, they are accompanied by an appropriate increase or decrease to s. Maximum space and average space are proven in exactly the same way that results are proven.

12. Visible-State Semantics

We have been describing computations by the initial and final values of variables. Sequential composition was defined as

(3) $P;Q\ =\ \exists x'', y'', ...\cdot$ (for $x', y', ...$ substitute $x'', y'', ...$ in P)
$\qquad\qquad\qquad\wedge$ (for $x, y, ...$ substitute $x'', y'', ...$ in Q)

which says that there are intermediate values, but these intermediate values are local to the description of sequential composition (they are bound by the quantifier). Only the global (free) values are visible, and they are the initial and final values.

To specify performance, both in time and space, we have added some auxiliary variables, but like the variables that deliver results, the performance variables are visible only by their initial and final values. For infinite computations, final values are meaningless (with the exception of t' whose final value is ∞). Infinite computations can be described by communication histories as in [Heh93] or by making the intermediate states visible as in [Heh94]. Making the intermediate states visible also allows us to describe parallel processes that co-operate through shared memory. In this paper, our concern is time and space; making intermediate states visible allows us to refer directly to the space occupied at any time during a computation.

Here is an example, contrived to be as simple as possible while including time and space calculations in an infinite computation. Until now, our examples have used the time variable t just for calculating performance; but some computations are sensitive to time, and depend on a clock. This example shows that such computations pose no extra problems.

(45) $GrowSlow \;\Leftarrow\; \textbf{if } t{=}2{\times}x \textbf{ then } (\textbf{alloc} \parallel x{:=}\ 2{\times}x) \textbf{ else tick};\ GrowSlow$

If the time t is equal to $2x$, then some space is allocated, and in parallel x is doubled; otherwise the clock ticks. Then the process is repeated forever. Variable x is the time stamp of the previous allocation. We will prove that at all times (starting from the initial execution time), the space allocated is bounded by the logarithm of the time (base 2) if we initialize variable x reasonably.

(46) $x(t) < t \le 2\ x(t) \;\Rightarrow\; \forall t''\colon t{\le}t''\cdot\ s(t'')-s(t) < log(t'')$

We keep t and t' for the initial and final execution time, but variables x and s are now functions of time. The value of x at time t is $x(t)$. Purely for the sake of shortening lengthy formulas, we write x for $x(t)$, x' for $x(t')$, x'' for $x(t'')$, and so on, and similarly for variable s . Thus (46) can be rewritten

(47) $x < t \le 2x \;\Rightarrow\; \forall t''\colon t{\le}t''\cdot\ s''-s < log(t'')$

In the visible state semantics, an assignment such as $x{:=}\ 2{\times}x$ has the following meaning:

(48) $x{:=}\ 2{\times}x \;=\; t' = t{+}1 \;\wedge\; x' = 2x \;\wedge\; \forall t''\colon t{\le}t''{\le}t'\cdot\ s''{=}s$

In (48), an assignment is assumed to take time 1 , but that is easily changed if desired. If time is continuous (real-time), (48) says that the value of s does not change during the assignment to x (and similarly for other variables if there were any). To make the proof easier, we shall take time to be integer-valued, although the result (47) we are proving holds also for continuous time. (48) becomes

(49) $x{:=}\ 2{\times}x \;=\; t' = t{+}1 \;\wedge\; x' = 2x \;\wedge\; s'{=}s$

For **alloc** we will suppose that 1 unit of space is allocated, and that it takes time 1 . So

(50) $\textbf{alloc} \;=\; s{:=}\ s{+}1 \;=\; t' = t{+}1 \;\wedge\; x'{=}x \;\wedge\; s' = s{+}1$

For the semantics of parallel composition in general in a visible-state semantics, the reader is referred to [Heh94]. In this special case,

(51) $\textbf{alloc} \parallel x{:=}\ 2{\times}x \;=\; t' = t{+}1 \;\wedge\; x' = 2x \;\wedge\; s' = s{+}1$

We suppose that **tick** takes 1 unit of time; again, that is easily changed if desired.

(52) $\textbf{tick} \;=\; t' = t+1 \;\land\; x'=x \;\land\; s'=s$

$$
\begin{array}{ll}
(53) & \textbf{if } t=2x \textbf{ then } (\textbf{alloc} \parallel x:= 2\times x) \textbf{ else tick} \\
= & t=2x \;\land\; (\textbf{alloc} \parallel x:= 2\times x) \;\;\lor\;\; t\ne 2x \;\land\; \textbf{tick} \\
= & t=2x \;\land\; t' = t+1 \;\land\; x' = 2\times x \;\land\; s' = s+1 \;\;\lor\;\; t\ne 2x \;\land\; t' = t+1 \;\land\; x'=x \;\land\; s'=s
\end{array}
$$

In the visible state semantics, sequential composition is defined as follows.

(54) $P;Q \;=\; \exists t''\colon t\le t''\le t'\cdot\; (\text{for } t' \text{ substitute } t'' \text{ in } P\,) \land (\text{for } t \text{ substitute } t'' \text{ in } Q\,)$

Last, to prove (45), we must define $GrowSlow$.

(55) $GrowSlow \;=\; x<t\le 2x \;\Rightarrow\; \forall t''\colon t\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x''$

According to (53) and (55), what we are trying to prove (45) has the form

$$
\begin{array}{lll}
(56) & (A\Rightarrow B \;\Longleftarrow\; C\lor D;\, A\Rightarrow B) & ;\ \text{distributes over } \lor \\
= & (A\Rightarrow B \;\Longleftarrow\; (C;\, A\Rightarrow B) \lor (D;\, A\Rightarrow B)) & \\
= & (A\Rightarrow B \;\Longleftarrow\; (C;\, A\Rightarrow B)) \land (A\Rightarrow B \;\Longleftarrow\; (D;\, A\Rightarrow B)) & \\
= & (B \;\Longleftarrow\; A \land (C;\, A\Rightarrow B)) \land (B \;\Longleftarrow\; A \land (D;\, A\Rightarrow B)) &
\end{array}
$$

So we can break the proof into two cases, proving first

(57) $B \;\Longleftarrow\; A \land (C;\, A\Rightarrow B)$

and second

(58) $B \;\Longleftarrow\; A \land (D;\, A\Rightarrow B)$

We start with the right side of (57).

$$
\begin{array}{lll}
(59) & x<t\le 2x \;\land\; (\,t=2x \;\land\; t' = t+1 \;\land\; x' = 2\times x \;\land\; s' = s+1; & \\
 & \qquad x<t\le 2x \;\Rightarrow\; \forall t''\colon t\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x''\;\;) & \text{use (54)} \\
= & x<t\le 2x \;\land\; \exists t'''\cdot\;\; t=2x \;\land\; t''' = t+1 \;\land\; x''' = 2\times x \;\land\; s''' = s+1 & \\
 & \qquad \land\; (x'''<t'''\le 2x''' \;\Rightarrow\; \forall t''\colon t'''\le t''\cdot\; 2^{s''-s'''}x''' = x'' < t'' \le 2x'') & \\
 & \multicolumn{2}{r}{\text{use one-point to eliminate } t''' \text{, and use } x^+ \text{ and } s^+ \text{ to abbreviate } x(t+1) \text{ and } s(t+1)} \\
= & x<t\le 2x \;\land\; t=2x \;\land\; x^+=2x \;\land\; s^+=s+1 & \\
 & \qquad \land\; (x^+<t+1\le 2x^+ \;\Rightarrow\; \forall t''\colon t+1\le t''\cdot\; 2^{s''-s^+}x^+ = x'' < t'' \le 2x'') & \\
= & x<t=x^+=2x \;\land\; s^+=s+1 \;\land\; (2x<t+1\le 4x \;\Rightarrow\; \forall t''\colon t+1\le t''\cdot\; 2^{s''-s-1}2x = x'' < t'' \le 2x'') & \\
 & \multicolumn{2}{r}{\text{the conjunct } x<t=2x \text{ discharges the antecedent } 2x<t+1\le 4x} \\
= & x<t=x^+=2x \;\land\; s^+=s+1 \;\land\; \forall t''\colon t+1\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x'' & \\
 & \multicolumn{2}{r}{\text{when } t''=t \text{, also } x''=x \text{ and } s''=s \text{ , so the domain of } t'' \text{ can be increased}} \\
= & t=x^+=2x \;\land\; s^+=s+1 \;\land\; \forall t''\colon t\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x'' & \text{drop conjuncts} \\
\Longrightarrow & \forall t''\colon t\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x'' &
\end{array}
$$

which is the left side of (57). Next we prove (58), starting with its right side.

(60) $x<t\le 2x \;\wedge\; (t\ne 2x \;\wedge\; t'=t+1 \;\wedge\; x'=x \;\wedge\; s'=s;$
$\qquad x<t\le 2x \;\Rightarrow\; \forall t''\colon t\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x''\;)$ use (54)

$=\quad x<t\le 2x \;\wedge\; \exists t'''\cdot\;\; t\ne 2x \;\wedge\; t'''=t+1 \;\wedge\; x'''=x \;\wedge\; s'''=s$
$\qquad \wedge\; (x'''<t'''\le 2x''' \;\Rightarrow\; \forall t''\colon t'''\le t''\cdot\; 2^{s''-s'''}x''' = x'' < t'' \le 2x'')$

use one-point to eliminate t''', and use x^+ and s^+ to abbreviate $x(t+1)$ and $s(t+1)$

$=\quad x<t\le 2x \;\wedge\; t\ne 2x \;\wedge\; x^+=x \;\wedge\; s^+=s$
$\qquad \wedge\; (x^+<t+1\le 2x^+ \;\Rightarrow\; \forall t''\colon t+1\le t''\cdot\; 2^{s''-s^+}x^+ = x'' < t'' \le 2x'')$

$=\quad x<t<2x \;\wedge\; x^+=x \;\wedge\; s^+=s \;\wedge\; (x<t+1\le 2x \;\Rightarrow\; \forall t''\colon t+1\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x'')$

the conjunct $x<t<2x$ discharges the antecedent $x<t+1\le 2x$

$=\quad x<t<2x \;\wedge\; x^+=x \;\wedge\; s^+=s \;\wedge\; \forall t''\colon t+1\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x''$

when $t''=t$, also $x''=x$ and $s''=s$, so the domain of t'' can be increased

$=\quad t<2x \;\wedge\; x^+=x \;\wedge\; s^+=s \;\wedge\; \forall t''\colon t\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x''$ drop conjuncts

$\Longrightarrow\quad \forall t''\colon t\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x''$

which is the left side of (58). So we have proven that the computation as defined by (45) satisfies the specification *GrowSlow* as defined by (55). All that is left is to show that *GrowSlow* implies the desired result (47).

(61) *GrowSlow* use (55)

$=\quad x<t\le 2x \;\Rightarrow\; \forall t''\colon t\le t''\cdot\; 2^{s''-s}x = x'' < t'' \le 2x''$ drop conjunct

$\Longrightarrow\quad x<t\le 2x \;\Rightarrow\; \forall t''\colon t\le t''\cdot\; 2^{s''-s}x = x'' < t''$ take logarithms

$=\quad x<t\le 2x \;\Rightarrow\; \forall t''\colon t\le t''\cdot\; s''-s + log(x) < log(t'')$ if $x<2x$ then $1\le x$ and $0\le log(x)$

$\Longrightarrow\quad x<t\le 2x \;\Rightarrow\; \forall t''\colon t\le t''\cdot\; s''-s < log(t'')$

Thus we conclude that at all times, the space allocated is bounded by the logarithm of the execution time.

13. Gas Burner

The final example of this paper has been treated, with minor deviations, by many researchers [Sør89], so our solution can be compared with theirs. It is to specify the control of a gas burner. The inputs are:
• real *temp* , which comes from a thermometer and indicates the actual temperature.
• real *desired* , which comes from a thermostat and indicates the desired temperature.
• boolean *flame* , which comes from a flame sensor and indicates whether there is a flame.
Its outputs are:
• *gas*:= *on* , which turns the gas on.
• *gas*:= *off* , which turns the gas off.
• *spark* , which maintains the gas and causes a spark for the purpose of igniting the gas.
Heat is wanted when the desired temperature falls ε below the actual temperature, and not wanted when the desired temperature rises ε above the actual temperature, where ε is small enough to be unnoticeable, but large enough to prevent rapid oscillation. To obtain heat, the spark should be applied to the gas for at least 1 second to give it a chance to ignite and to allow the flame to become stable. But a safety regulation states that the gas must not remain on and unlit for more than 3 seconds. Another regulation says that when the gas is shut off, it must not be turned on again for at least 20 seconds to allow any accumulated gas to clear. And finally, the gas burner must respond to its inputs within 1 second.

Our specification is $GasIsOff \vee GasIsOn$, where

```
(62)  GasIsOff  =  if temp < desired – ε
                   then (gas:= on;  spark ∧ 1 ≤ t′–t ≤ 3;  GasIsOn)
                   else (t′–t < 1;  GasIsOff)

(63)  GasIsOn   =  if temp < desired + ε ∧ flame
                   then (t′–t < 1;  GasIsOn)
                   else (gas:= off;  20 ≤ t′–t < 21;  GasIsOff)
```

We are using the time variable to represent real time in seconds. The specification $1 \le t'-t \le 3$ represents the passage of at least 1 second but not more than 3. The specification $20 \le t'-t < 21$ is similar. A specification that a computation be slow enough is always easy to satisfy. A specification that it be fast enough requires us to build fast enough hardware; in this case it is easy since instruction times are microseconds and the time bounds are seconds.

One can always argue about whether a formal specification captures the intent of an informal specification. For example, if the gas is off, and heat becomes wanted, and the ignition sequence begins, and then heat is no longer wanted, this last input may not be noticed for up to 3 seconds. It may be argued that this is not responding to an input within 1 second, or it may be argued that the entire ignition sequence is the response to the first input, and until its completion no response to further inputs is required. At least the formal specification is unambiguous, and it compares favorably with specifications in other formalisms for clarity.

14. Conclusions

We have presented the means, with examples, by which time and space requirements can be specified and proven, along with results. We have not introduced any special notations or extra formalism for doing so. Rather, we have used ordinary variables for time and space, and ordinary boolean and arithmetical operators, and found them to be entirely adequate. Indeed, we believe this approach is better than formalisms that do not treat time and space as ordinary variables by being both simpler and more general. The same approach can be used, for example, to specify and prove the processor × time requrements. The examples were small, intended only to convince the reader that the theory works; the question of scaling up was not addressed. The proofs were shown in detail, making them rather long for the simple theorems being proved, but showing that there are no difficult steps, and that an automated prover could be expected to handle them.

Acknowledgements

I thank Michael Donat and Piotr Rudnicki for prompting me to write this paper. I thank IFIP Working Group 2.3 for being my research forum. I thank NSERC for support.

References

[HaU98] Hayes, I.J., Utting, M.: “Deadlines are Termination”, chapter 15 in *Programming Concepts and Methods*, edited by D.Gries and W.-P.deRoever, Chapman and Hall, 1998.

[Heh93] Hehner, E.C.R.: *A Practical Theory of Programming*, Springer-Verlag, New York, 1993.

[Heh94] Hehner, E.C.R.: “Abstractions of Time”, chapter 12 in *A Classical Mind*, edited by A.W.Roscoe, Prentice-Hall International, London, 1994.

[Pap94] Papadimitriou, C.H.: *Computational Complexity*, Addison-Wesley, 1994.

[Sha79] Shaw, M.: “A Formal System for Specifying and Verifying Program Performance”, technical report, Computer Science Department, Carnegie-Mellon University, 1979.

[Sør89] Sørensen, E.V., Ravn, A.P., Rischel, H.: “Control Program for a Gas Burner”, ProCoS ESPRIT BRA 3104, Technical Report ID/DTH EVS2, Computer Science Department, Technical University of Denmark, Lyngby Denmark, 1989.

Received May 1997
Accepted in revised form July 1998 by C.B.Jones