Copyright Notice

Semantics of straight-line programs. (Draft Chap. 4)

Program execution as change of machine state

This is a monograph about a theory of programming language semantics. A "programming language" is usually identified with all programs that can be written in it and the "semantics" refers to what would happen each time, were such a program fed into an appropriate computer.

Remark We regret the terminology we used in the preceding paragraph, but the terminology is by now so firmly established that there seems to be no point anymore in deviating from it. In retrospect we would have been much happier, had programming languages never been called "languages". Admittedly, the metaphor has been helpful in the very beginning: the linguistic analogy gave a hint what aspects to distinguish and provided a terminology for identifying them. Our regret comes from the fact that, beyond that initial service, the linguistic analogy has been more confusing than helpful. The term "programming language" was first a symptom of, and -once established- a driving force behind the habit of describing computing systems in anthropomorphic terminology, a habit we have to shed for all its misleading connotations if a mature computing science is to emerge. (End of Remark.)

Now, one aspect of doing science consists in ignoring what is chosen to be considered irrelevant, and the question is: what are we going to ignore and what are we going to take into account when referring to "what would happen, were such a program fed into a computer"?

We are going to ignore all physical characteristics of the computer: its technology, its size, its weight, its power consumption, its reliability, its maintenance contract, and its manufacturer, just to mention a few irrelevantia. Presumably the machine has a manual, but in that case we do not regard it as the manual's task to describe the machine, but as the machine's task to provide a -hopefully correct- physical model of the manual's contents. In case of a discrepancy, we would blame the machine and not the manual: it is the abstract machine we can think about that matters.

Having decided that it is the thinkable machine that really matters in our heads, we go one step further and ignore all sorts of constraints that are hard to avoid in physical machines, such as storage size and word length: in the machine we think about -also known as "the Good Lord's Machine"- any integer can be increased by 1. To simplify matters further, we ignore all devices for input and output of information: the value of the input absorbed by a computation is deemed

to be captured by the initial state of the machine -i.e. the state in which the machine starts the computation-, the value of the output is similarly deemed to be captured by the final state of the machine -i.e. the state in which the machine is left upon completion of the computation-.

Our next step is a little bit more radical. In the beginning we were very preoccupied with the machines; as a result, it was viewed as the task of the programs to instruct our machines, and the final state was not only viewed but actually defined as the result of the computation evoked under control of the program. In those days, the semantics of programming languages were defined in terms of the computational steps that would take place during the executions of the programs, the final state reflecting the cumulative effect of all the steps of a computation. Such an approach is now called "an operational definition" of the semantics.

Nowadays, we are no longer so preoccupied with the machines; we are more interested in our programs and view it as the task of the machines to execute our programs, i.e. to bring themselves, for any given initial state and given program, into a corresponding final state. We are not so interested any more in what happens during the computation, provided it establishes the desired nett effect. Accordingly we shall no longer define the semantics via the detour

of the nett effect of the computations a program may evoke; instead we shall define more directly how for any program in the language initial and final states are connected.

We do so, because such a non-operational definition of semantics has clear technical advantages. For instance, in order to establish the equivalence of two different programs, we can derive for both of them how initial and final states are connected and if those connections are the same, the programs are equivalent. The fact that, when executed, they establish the same nett effect via possibly different computations no longer complicates our reasoning since we don't need to consider the individual computations any more.

(Besides assisting us in reasoning about given programs – for instance to establish their equivalence – the non-operational definition of programming language semantics also serves as a basis for a calculus for the derivation of programs, their required nett effects being given. But that is another story.)

The only things of interest that remain are the initial state, the final state and how a program connects them. As computations no longer enter the picture, we can now forget about machines and treat programs as mathematical objects in their own right, temporarily ignoring that they

admit the interpretation of executable code.

The semantics of a program

One way of trying to capture the semantics of a given program S is to consider the final state as a function of the initial state in which S is started. This way has been followed (extensively), but it suffers from two disadvantages.

The one disadvantage is that there may be initial states for which S, when started, fails to terminate, i.e. the final state, considered as function of the initial one, is not defined for all points of state space. We then have two options, either we pay the price of dealing with what are called "partial functions" – i.e. functions defined on a smaller domain than the one under consideration – or we make them total functions by extending the state space by adding a special "point" – usually called "bottom" – that can be interpreted as "stuck in an endless computation". Both have their disadvantages. (The latter is really a coding trick: it buys the advantage of dealing with total functions at the price of destroying the homogeneity of the state space.)

The other disadvantage is more serious: the approach is really geared to what are called "de-

terministic programs", i.e. programs for which the computation – whether finite or not – is completely determined by the initial state in which the program has been started. For a variety of reasons – about which probably more later – we would like to include in our considerations nondeterministic programs as well (e.g. programs that could be implemented by occasionally tossing a coin), and such nondeterministic programs don't fit nicely in the functional framework. People have tried it in various ways, such as treating the "coin" as some sort of hidden input – again a coding trick! – or by introducing functions from states to subsets of states. The cleanest approach along these lines is probably to capture the semantics of a program as a relation between initial and final state; it has at least the advantage that the relational calculus as such is available. (Experience with the relational calculus has, however, not been too favourable in this context: firstly, "bottom" enters the picture constantly, secondly, many quantifications over the state space have to be written down explicitly, and, thirdly, it has proved to be conducive to errors: in the relational calculus the two arguments of a relation are treated on the same footing and, as a result, the essential asymmetry between initial and final state is insufficiently reflected.)

The long and the short of it is that we want to

capture the semantics of a program differently.

Consider a program S and a predicate X on the (final) state space; the predicate X then defines a dichotomy of the state space: in those states in which X does not hold, ¬X holds. Then each possible execution of S falls in precisely one of the following three - mutually exclusive - classes:

"S gets stuck" : the execution of S fails to terminate - does not lead to a final state -

"S establishes X" : the execution of S terminates - leads to a final state - in a state in which X holds

"S establishes ¬X": the execution of S terminates - leads to a final state - in a state in which ¬X holds.

The three classes are clearly mutually exclusive and exhaustive. Note that, since there are no states in which false holds, the class "S establishes false" had better be empty.

We now take the view that we know everything that is to be known about the semantics of S if we know for any predicate X and for any initial state which of the three executions are possible or, equivalently, which of the three executions are impossible.

To each class corresponds a dichotomy of the initial state space, viz. whether in such a state as starting point an execution of that class is possible or not. We characterize -as always- these dichotomies by predicates. The first one, corresponding to "S gets stuck" is independent of X.

TS : holds in all initial states for which "S gets stuck" is impossible

¬TS : holds in all initial states for which "S gets stuck" is possible.

¬wlp*(S,X) : holds in all initial states for which "S establishes X" is impossible

wlp*(S,X) : holds in all initial states for which "S establishes X" is possible

(and similarly for ¬X: just replace X by ¬X).

From the fact that the execution class "S establishes false" had better be empty, we deduce from the above that wlp had better satisfy [wlp*(S, false) ≡ false] or, equivalently, [wlp(S, true) ≡ true], or, for short

(0) [wlp(S, true)] .

Moreover, since [wlp(S,X) ≡ ¬wlp*(S,¬X)] we deduce

wlp(S,X): holds in all initial states for which "S establishes ¬X" is impossible .

But since the three classes are exhaustive and mutually exclusive, we can also formulate

wlp(S,X) : holds for all initial states for which for which the possibilities are restricted to "S gets stuck" and "S establishes X".

or holds for all initial states for which S establishes X if it does not get stuck.

Since an execution belonging to both classes "S establishes X" and "S establishes Y" belongs to the class "S establishes $X \wedge Y$" as well, and vice versa, i.e.

$$[wlp(S,X) \wedge wlp(S,Y) \equiv wlp(S, X \wedge Y)]$$

the predicate transformer wlp(S,?) had better be conjunctive, and in view of (0) even universally so.

In the sequel we shall define – recursively via the grammar – for any program S of a model programming language wlp(S,?). In view of the above we accept the obligation to prove for each extension of the language that Theorem 4.0 continues to hold.

Theorem 4.0 For any program S, the predicate transformer wlp(S,?) is universally conjunctive.

Its proof will be distributed through this monograph.

The second predicate transformer associated with S is $wp(S, ?)$, defined by

$$[wp(S,X) \equiv TS \wedge wlp(S,X)] \quad ,$$

from which we derive, by substituting $X := true$, on account of (0)

$$[wp(S, true) \equiv TS] \quad .$$

The name TS can thus be eliminated; it is customary to define the predicate transformer $wp(S, ?)$ by

(1) $$[wp(S,X) \equiv wp(S, true) \wedge wlp(S,X)] \quad .$$

In view of the definition of TS we have

$wp(S,X)$: holds for all initial states in which "S establishes X" is guaranteed (being the only class)

$\neg wp(S,X)$: holds for all initial states in which "S gets stuck" or "S establishes $\neg X$" are among the possibilities, i.e. all initial states in which S might fail to establish X .

(The names wp and wlp are short for "weakest precondition" and "weakest liberal precondition" respectively. These names having been in use now for more than a decade, we shall stick to them.)

Since S can be started in any initial state, and the ensuing execution falls in one of the three classes, everywhere in state space executions from at least one of the three classes had better be possible, i.e.

$$[\neg TS \vee wlp^*(S,X) \vee wlp^*(S,\neg X)]$$

should hold. Thanks to $[TS \equiv wp(S,true)]$, de Morgan's Law and the definition of the conjugate, the above condition is equivalent to

$$[wp(S,true) \wedge wlp(S,\neg X) \wedge wlp(S,X) \equiv false];$$

thanks to wlp's conjunctivity, this is equivalent to

$$[wp(S,true) \wedge wlp(S,false) \equiv false]$$

which, thanks to (1) can be shortened to

$$[wp(S,false) \equiv false].$$

In the sequel we shall define for any program besides the predicate transformer $wlp(S,?)$ the predicate $wp(S,true)$ as well — and hence (see (1)) the predicate transformer $wp(S,?)$ —. In view of the above we accept the obligation to prove for each extension of the language that Theorem 4.1 continues to hold.

Theorem 4.1 For any program S,

$$[wp(S,false) \equiv false].$$

Its proof will be distributed through this monograph.

Note Theorem 4.1 is better known as "The Law of the Excluded Miracle". (End of Note.)

* * *

In the remainder of this chapter we shall define the semantics of a number of statements with which we can write what is known as "straight-line programs" - briefly: programs without repetition or, more generally, recursion - . We shall define them by giving for them the predicate transformer wlp(S,?) and the predicate wp(S,true) ; in accordance with the above we shall prove for them the universal conjunctivity of wlp (Theorem 4.0) and The Law of the Excluded Miracle (Theorem 4.1). In view of later applications, we shall give their disjunctivity properties as well.

S = havoc

Def. [wp(havoc,true) ≡ true]

[wlp(havoc, X) ≡ [X]] .

Cons. [wp(havoc, X) ≡ [X]] .

This statement is the first one to illustrate a direct

consequence of (1): for a statement S with [wp(S, true)] , the predicate transformers wp and wlp are the same. For reasons that will become clear in a moment, most commercially available programming languages do not include the statement havoc (at least not intentionally); its execution should bring the machine in an arbitrary state, i.e. should assign to each variable spanning the state space an unpredictable value. Yet it is good to know the statement havoc , as it provides an important counterexample.

Proof 4.0 for havoc . On account of Lemma 3.10 , wlp(havoc, ?) is universally conjunctive. (End of Proof 4.0 for havoc .)

Proof 4.1 for havoc .

```
    wp(havoc, false)
=   {(1)}
    wp(havoc, true) ∧ wlp(havoc, false)
=   {def. of havoc }
    true ∧ [false]
=   {predicate calculus}
    false
```

(End of Proof 4.1 for havoc .)

Being universally conjunctive, wlp(havoc, ?) is monotonic , but that is, in general, its only disjunctivity property since, in general, wlp(havoc, ?) is neither disjunctive nor or-continuous.

For a state space large enough that there exists an X such that $\neg[X] \wedge \neg[\neg X]$ – i.e. a state space of 2 points, i.e. spanned by a single boolean variable suffices – we observe for such an X

$$
\begin{array}{ll}
 & wlp(havoc, X \vee \neg X) \\
= & \{\text{predicate calculus}\} \\
 & wlp(havoc, true) \\
= & \{\text{def. of havoc}\} \\
 & [true] \\
= & \{\text{predicate calculus}\} \\
 & true \\
= & \{\text{assumption about } X\} \\
 & \neg[X] \wedge \neg[\neg X] \\
\neq & \{\text{de Morgan}\} \\
 & [X] \vee [\neg X] \\
= & \{\text{def. of havoc}\} \\
 & wlp(havoc, X) \vee wlp(havoc, \neg X) \quad ,
\end{array}
$$

hence havoc is in general not disjunctive.

For a state space spanned by a single integer variable x, the predicates $X.i$ given by $x \leq i$ form a monotonic sequence. Square brackets corresponding to universal quantification over x, we have

$$
\begin{array}{ll}
 & wlp(havoc, (\underline{E}i :: X.i)) \\
= & \{\text{def. of havoc}\} \\
 & [(\underline{E}i :: X.i)] \\
\neq & \{\text{def. of } X.i\} \\
 & (\underline{E}i :: [X.i])
\end{array}
$$

$= \{ \text{def. of havoc} \}$
$(\underline{E}i :: wlp(havoc, X.i))$,

hence havoc is in general not or-continuous.

Note. In the jargon, havoc is a nondeterministic statement. (It is even the most nondeterministic one of all.) In an infinite state space it is unboundedly nondeterministic, and the above argument shows that unbounded nondeterminism excludes or-continuity of wlp . (End of Note.)

S= abort

Def. $[wp(abort, true) \equiv false]$
$[wlp(abort, X) \equiv true]$.

Cons. $[wp(abort, X) \equiv false]$.

For reasons that will become clear in a moment, all commercially available programming languages include the statement abort in one form or another (sometimes quite unintentionally so): it corresponds to the statement whose execution is guaranteed to fail to terminate properly. It is sometimes known under names such as "stop", "alarm", "error message"; as we shall see later we have defined it to be semantically equivalent to the nonterminating repetition.

Proof 4.0 for abort. On account of Lemma 3.12, wlp(abort, ?) is universally conjunctive. (End of Proof 4.0 for abort.)

Proof 4.1 for abort.

```
    true
=  { consequence of def. of abort }
   (A X:: [wp(abort, X) ≡ false])
⇒  { instantiation }
   [wp(abort, false) ≡ false]
```

(End of Proof 4.1 for abort.)

Furthermore we conclude on account of Lemma 3.12 that wlp(abort, ?) is unboundedly disjunctive and that wp(abort, ?) is universally disjunctive.

Note. Alternatively we could have defined the semantics of abort by postulating that its predicate transformers satisfy Theorems 4.0 and 4.1, plus

$$[\mathrm{wlp}(\mathrm{abort}, \mathrm{false}) \equiv \mathrm{true}]$$

which is the formal expression of the circumstance that all executions of abort fail to terminate. (End of Note.)

S = skip

Def. $[\mathrm{wp}(\mathrm{skip}, \mathrm{true}) \equiv \mathrm{true}]$

$[\mathrm{wlp}(\mathrm{skip}, X) \equiv X]$

Cons. $[wp(skip, X) \equiv X]$

The statement skip is known under many names, such as "the empty statement", "no-op", "continue"; in the syntax of ALGOL 60 the mistake was made of representing it by no character at all.

Since $[wp(skip, true)]$, the predicate transformers $wp(skip, ?)$ and $wlp(skip, ?)$ are the same.

Proof 4.0 for skip. On account of Lemma 3.11, $wlp(skip, ?)$ is universally conjunctive. (End of Proof 4.0 for skip.)

Proof 4.1 for skip.

$$
\begin{array}{ll}
 & true \\
= & \{ \text{consequence of def. of skip} \} \\
 & (\underline{A}\, X :: [wp(skip, X) \equiv X]) \\
\Rightarrow & \{ \text{instantiation} \} \\
 & [wp(skip, false) \equiv false]
\end{array}
$$

(End of Proof 4.1 for skip.)

Furthermore we conclude on account of Lemma 3.11 that $wlp(skip, ?)$ and $wp(skip, ?)$ are universally disjunctive.

Note. For the definition of the semantics of skip it would have sufficed to postulate

$$[wp(skip, X) \equiv X] \,,$$

which is the formal expression of the circumstance that each execution of skip leaves the machine

state unaffected. (End of Note.)

S = "x := E"

Def. $[wp("x := E", true) \equiv true]$

$[wlp("x := E", X) \equiv X^x_E]$.

Cons. $[wp("x := E", X) \equiv X^x_E]$.

Remark on notation If the statement is more complicated than a single identifier and should be parsed to retrieve its components, it is customary to surround it by a pair of quotes so as to make clear that the following comma does not belong to it. Though it is not a custom we would like to defend, we shall stick to it the few times it is applicable. (End of Remark on notation.)

The above gives the semantics of the assignment statement in its simplest form. On account of $[wp("x := E", true)]$, its wlp and wp are again the same predicate transformer. The definition of $wp("x := E", true)$ adopted here excludes that the execution of $x := E$ gets stuck, i.e. requires E to be a total expression. It is tempting to replace its definition by

$$[wp("x := E", true) \equiv def.E] \quad ,$$

where $def.E$ does not necessarily satisfy $[def.E]$

and holds precisely in those states for which E is defined. We have resisted that temptation since the introduction of def.E in the definition of the assignment statement does not really solve the problem of partially defined expressions. For instance, what would the value of X^x_E be where E is not defined? And also, if we wish to capture that evaluation of E might lead to a nonterminating computation, shouldn't we also try to capture the possibility that the evaluation of E leads to a nondeterministic process?

We would rather take here the opportunistic view that the use of partially defined expressions is the result of not caring to define them totally (i.e. "outside our domain of interest"), and that the problem of how to prevent that laziness - or "shorthand" if you prefer - from leading us astray in our reasoning is a more general problem of formal mathematics that should not clutter up our definition of the semantics of the assignment statement. How not to apply definitions outside their domain of validity is a general problem probably outside the scope of this monograph in which, at least for the time being, all expressions will be total.

Proof 4.0 for "x:= E". On account of Lemma 3.26, wlp("x:= E", ?) is universally conjunctive. (End of Proof 4.0 for "x:= E".)

Proof 4.1 for "x:= E"

$$
\begin{aligned}
& wp(\text{"}x := E\text{"}, false) \\
= & \{\text{consequence of def., with } X := false\} \\
& false^{x}_{E} \\
= & \{\text{def. of substitution}\} \\
& false
\end{aligned}
$$

(End of Proof 4.1 for "x:= E".)

We point out that the x in the above need not be one of the programs "concrete" variables, it could be a pair of them, say (y,z). The assignment statement could then be written

$$y, z := E0, E1 \quad ,$$

and X^{x}_{E} would have to be replaced by $X^{y,z}_{E0,E1}$, the multiple substitution (sometimes called the "simultaneous" substitution). This is extendible to n-tuples.

Disjunctivity missing

S = "S0; S1"

Def. [wp("S0;S1", true) ≡ wp(S0, wp(S1, true))]

[wlp("S0;S1", X) ≡ wlp(S0, wlp(S1, X))] .

Cons. [wp("S0; S1", X) ≡ wp(S0, wp(S1, X))] .

Because the Consequence is not immediately obvious we give first the

Proof of Consequence

$$
\begin{aligned}
& wp("S0;S1", X) \\
= & \{(1)\} \\
& wp("S0;S1", true) \wedge wlp("S0;S1", X) \\
= & \{\text{def. of } "S0;S1"\} \\
& wp(S0, wp(S1, true)) \wedge wlp(S0, wlp(S1, X)) \\
= & \{(1)\} \\
& wp(S0, true) \wedge wlp(S0, wp(S1, true)) \wedge wlp(S0, wlp(S1, X)) \\
= & \{ wlp(S0, ?) \text{ is conjunctive} \} \\
& wp(S0, true) \wedge wlp(S0, wp(S1, true) \wedge wlp(S1, X)) \\
= & \{(1)\} \\
& wp(S0, true) \wedge wlp(S0, wp(S1, X)) \\
= & \{(1)\} \\
& wp(S0, wp(S1, X))
\end{aligned}
$$

(End of Proof of Consequence.)

Compared to our previous definitions of the semantics of statements, our definition of the semantics of the semicolon introduces a novelty in the sense that the semantics of $S0;S1$ is given terms of those of $S0$ and $S1$: the properties of the predicate transformer $wlp("S0;S1", ?)$ depend on those of $wlp(S0, ?)$ and $wlp(S1, ?)$. So we cannot prove that $wlp("S0;S1", ?)$ is "always" universally conjunctive: locally we can only give

Proof 4.0 for "$S0;S1$" On account of Lemma 3.14,

wlp("S0; S1", ?) is universally conjunctive if both wlp(S0, ?) and wlp(S1, ?) are. (End of Proof 4.0 for "S0; S1".)

We are in a similar position with

Theorem 4.1 for "S0; S1". "S0; S1" satisfies the Law of the Excluded Miracle if both S0 and S1 do.

Proof 4.1 for "S0; S1"

```
    wp("S0; S1", false)
= { consequence of def. "S0; S1" with X := false }
    wp(S0, wp(S1, false))
= { Theorem 4.1 holds for S1 }
    wp(S0, false)
= { Theorem 4.1 holds for S0 }
    false.
```

(End of Proof 4.1 for "S0; S1".)

The above are only the induction steps. However, for each statement introduced all by itself we show the validity of Theorems 4.0 and 4.1; for each statement composed of other statements we show – as we just did – that the Theorems 4.0 and 4.1 are valid for the composite provided they are valid for the components. Via induction ("over the grammar", as the jargon says) we are then allowed to conclude that Theorems 4.0 and 4.1 are valid for all statements of the programming language.

From Lemma 3.14 we furthermore derive that

wlp("S0;S1",?) enjoys all disjunctive properties shared by wlp(S0,?) and wlp(S1,?), and similarly for wp.

S = IF

In this monograph, IF is used to denote "alternative constructs" of the form

if B0 → S0 fi
if B0 → S0 [] B1 → S1 fi
if B0 → S0 [] B1 → S1 [] B2 → S2 fi ,

in general if ([] i:: B.i → S.i) fi, where the i are taken from a usually non-empty, usually finite bag.

With the abbreviation BB given by [BB ≡ (E i:: B.i)], we have:

Def. [wp(IF,true) ≡ BB ∧ (A i:: ¬B.i ∨ wp(S.i,true))]

[wlp(IF,X) ≡ (A i:: ¬B.i ∨ wlp(S.i,X))] .

Cons. [wp(IF,X) ≡ BB ∧ (A i:: ¬B.i ∨ wp(S.i,X))] .

(In this case, the verification that the Consequence follows from the Definitions is again left to the reader.

We have to prove

Theorem 4.0 for IF Predicate transformer wlp(IF,?) is universally conjunctive if all the wlp(S.i,?) are.

Proof 4.0 for IF . Because disjunction distributes over universal quantification f given by $[f.X \equiv g.X \vee Y]$ enjoys each conjunctivity enjoyed by g - Note: should this have been a lemma of Chapter 3, e.g. a replacement of Lemma 3.13? - , $\neg B.i \vee wlp(S.i, ?)$ is universally conjunctive if $wlp(S.i, ?)$ is. From Lemma 3.15 it now follows that $wlp(IF, ?)$ is universally conjunctive if all $wlp(S.i, ?)$ are.

(End of Proof 4.0 for IF .)

Theorem 4.1 for IF . IF satisfies the Law of the Excluded Miracle if all the S.i do .

Proof 4.1 for IF

$$
\begin{aligned}
& wp(IF, false) \\
= & \{ \text{consequence of def. IF with } X := false \} \\
& BB \wedge (\underline{A}i :: \neg B.i \vee wp(S.i, false)) \\
= & \{ \text{Miracle Excluded for all } S.i \} \\
& BB \wedge (\underline{A}i :: \neg B.i) \\
= & \{ \text{def. of } BB \} \\
& BB \wedge \neg BB \\
= & \{ \text{predicate calculus} \} \\
& false \quad .
\end{aligned}
$$

(End of Proof 4.1 for IF .)

From Lemma 3.25 it follows that $wlp(IF, ?)$ is or-continuous provided all the $wlp(S.i, ?)$ are and their number is finite. Admission of an infinite number of alternatives between if and fi

in general destroys or-continuity! Since conjunction distributes over existential quantification, wp(IF,?) is or-continuous if wlp(IF,?) is .

And this concludes our treatment of straight-line programs.

Comments. In the previous chapter, the formulation of Lemma 3.25 should have been extended with "(Extendible to n-tuples)".

For Lemma 3.13 I propose

Lemma 3.13. Let f be given by [f.X ≡ g.X ∧ Y]; for arbitrary Y, f enjoys each junctivity enjoyed by g, universal conjunctivity excepted.

Proof 3.13 We observe for non-empty V over which g is conjunctive and any W over which g is disjunctive

```
  f.(A X: X∈V: X)
= {def. of f}
  g.(A X: X∈V: X) ∧ Y
= {g conjunctive over V}
  (A X: X∈V: g.X) ∧ Y
= {V not-empty}
  (A X: X∈V: g.X ∧ Y)
= {def. of f}
  (A X: X∈V: f.X)
```

```
  f.(E X: X∈W: X)
= {def. of f}
  g.(E X: X∈W: X) ∧ Y
= {g disjunctive over W}
  (E X: X∈W: g.X) ∧ Y
= {distribution}
  (E X: X∈W: g.X ∧ Y)
= {def. of f}
  (E X: X∈W: f.X)
```

(End of Proof of 3.13)

This draft is still young and so I am reasonably pleased with it. I had a few problems, e.g. how much to elaborate the remark at the bottom of EWD910-0: I know cultures in which no amount of elaboration would suffice, but also cultures in which the remark is superfluous.

Page 0-5 gave rise to a lecture that was greatly appreciated by the audience, which confirms to me such considerations should be included. Pages 6-10 had to be done very slowly, otherwise people get utterly confused about "it is impossible that something will not happen". The negations get a little bit overpowering; it is therefore good that these considerations are confined to a few pages. (Is this the reason why the relational calculus has invited mistakes?)

On p.11, a subchapter heading is missing. The rest is a bit enumerative, but that seems the nature of the topic.

Austin, 11 March 1985

prof.dr. Edsger W. Dykstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
United States of America