) CHAPTER NE

The Kernel and its Extensions

"Mathematical formulation allows us to remain much further from the computer than would otherwise be the case, and in this context any programming language is already too near."

Griffiths 1975

Introduction

The programming language we will use will include a new type of primitive statement called an <u>atomic</u> <u>description</u> which was first described in [Back 80]. It is essentially a general nondeterministic assignmet statement which affects the scope as well as the value of variables in the state. This is because it can add and remove variables to and from the state as well as change the values assigned to them. If **x** and **y** are lists of distinct variables which have no variables in common and **Q** is a formula of first order logic then the effect of the atomic description:

x/y.Q

is to add the variables in the list **x** to the state (those that are not already in the state), assign values to them in such a way that the formula **Q** becomes true, remove the variables in **y** from the state and terminate. If there is more than one possible assignment to the variables in **x** such that **Q** becomes true then one is chosen nondeterministically. If there is no such assignment then the statement does not terminate.

Using this statement any specification of a program can be expressed as a single statement in our language and the Atomic Description will thus be the <u>only</u> primitive statement we require. It has the added advantage of having a very straightforward mathematical semantics. This means that our programming language can also be used as our specification language so that the proof of the implementation of a specification is simply a special case of the proof of a refinement. This will enable us to transform specifications into programs and also take a given program and attempt to derive its specification (or at least the specifications of parts of the program). We give an example of using this process as an aid to program comprehension and program maintenance in Chapter Nine.

This statement is similar to the "specification statement" used by C.Morgan and others [Morgan 88] which is written:

v:[Pre/Post]

where **v** is a list of variables and **Pre** and **Post** are formulae, **Pre** is called the precondition, and **Post** is called the postcondition. If the precondition is not satisfied initially then the statement does not terminate. For <u>any</u> initial state which satisfies the precondition the statement <u>must</u> terminate in a state which satisfies the postcondition and in doing so it may only change the values of the variables in the list $\mathbf{v}$. References to variables $\mathbf{x}_0$ etc. where $\mathbf{x}$ is in $\mathbf{v}$ refer to the initial values of these variables. So to express this specification statement in terms of the atomic description we might write:

$$\mathbf{v}_0/\langle\rangle.(\mathbf{Pre} \wedge \mathbf{v}{=}\mathbf{v}_0);\ \mathbf{v}/\mathbf{v}_0.\mathbf{Post}$$

However, this program will not terminate if there is no assignment of values to the variables in $\mathbf{v}$ which causes **Post** to be satisfied, but the specification statement must terminate whenever **Pre** is satisfied initially. Thus a statement such as [**true/false**] (which terminates on any initial state, changes the values of no variables, but terminates in a state for which **false** is true) is allowed–such statements are called "Miracles" for obvious reasons. This requirement precludes the interpretation of specification statements as state transformations in the way we have done since no final state can satisfy **false**. Adding Miracles to the programming language is a bit like adding **0/0** to arithmetic, one can "prove" that **x=0/0** for any number **x** and similarly one can prove that a Miracle implements any specification. Such statements were specifically excluded by Dijkstra in [Dijkstra 76] by his "Law of the excluded miracle" which defines **wp(S,false)** to be **false** for any statement **S**.

Examples of Atomic Descriptions

$\langle\mathbf{A}\rangle/\langle\mathbf{B}\rangle.(\mathbf{A}{=}\ \wp(\mathbf{B})\cup\omega_1\cup\mathbf{C})$ sets **A** to $\wp(\mathbf{B})\cup\omega_1\cup\mathbf{C}$ and then removes **B** from the state space.
$\langle\mathbf{y}\rangle/\langle\rangle.(\mathbf{y}{=}\ \sum_{n<\omega}\mathbf{n}^{-2}\mathbf{sin}(\mathbf{nx}^2)\ \wedge\ \mathbf{x}{>}\mathbf{0})$ sets **y** to the value of the expression iff **x>0** initially.
$\langle\mathbf{y}\rangle/\langle\rangle.(\mathbf{y}{-}\varepsilon < \mathbf{Lim}\ _{n<\omega}(\sum_{1\leqslant i\leqslant n}\mathbf{1/i} - \mathbf{log}_e\mathbf{n}) < \mathbf{y}{+}\varepsilon)$
sets **y** to within ε of Euler's constant–provided $\varepsilon > \mathbf{0}$. If $\varepsilon \leqslant \mathbf{0}$ initially then this will not terminate.

These examples illustrate that the value assigned to a variable could be anything: including an infinite set or a real number–we are not limited to integer values. This illustrates a comment made by Landin in [Landin 66]: "this discussion . . . reveals the possibility that primitives might be sensationally non-algorithmic."

Proof Rules

Our transformations are proved by showing that the weakest preconditions corresponding to two statements are equivalent (or that one implies the other in the case of a refinement) as formulae of infinitary first order logic. From the theorem below we can then deduce that the two programs are equivalent in the sense of having the same denotational semantics. First a few definitions:

Defn: A model for programs is an interpretation of the constants, function symbols and relation symbols of the logical formulae used in the programs as elements, functions and relations on a given set of values. With such a model one can calculate give a truth value to a formula given the values of its free variables. A model provides an interpretation of programs as state transformations. If the state transformation defined by program **S** under model **M** is a refinement of the state transformation defined by $\mathbf{S'}$ then we write $\mathbf{S}\leqslant_M\mathbf{S'}$. Sentences are formulae with no free variables, so all sentences are either true or false with respect to a model. If Δ is a set of sentences (ie formulae with no free variables) then we say that a model satisfies Δ if all the sentences in Δ are true in the model. If for each such model **M** we have $\mathbf{S}\leqslant_M\mathbf{S'}$ then we write $\Delta\models\mathbf{S}\leqslant\mathbf{S'}$.

Defn: Suppose **S** and $\mathbf{S'}$ are programs which have the same initial set of variables and both assign values to the variables $\mathbf{w}_1,\ldots,\mathbf{w}_n$. Suppose we have a set Δ of sentences of infinitary first order logic (these define the properties we require of the function and relation symbols in the logic). Suppose we extend the logical language by adding a new **n**-ary relation symbol **G**.
Then if we can prove $\big(\mathbf{WP(S,G(w_1,\ldots,w_n))\Rightarrow WP(S',G(w_1,\ldots,w_n))}\big)$ by using the sentences in Δ as extra axioms we write $\Delta\vdash\mathbf{S}\leqslant\mathbf{S'}$.

The following theorem is crucial, and explains the similarity in notation of the last two definitions:

Theorem: (i) For any set Δ of sentences: $\Delta\vdash\mathbf{S}\leqslant\mathbf{S'}$ implies $\Delta\models\mathbf{S}\leqslant\mathbf{S'}$.
(ii) For any countable set Δ of sentences: $\Delta\models\mathbf{S}\leqslant\mathbf{S'}$ implies $\Delta\vdash\mathbf{S}\leqslant\mathbf{S'}$.

Proof: For the detailed proof of this theorem see [Ward 89a].

Cor: Proof Rule for Equivalence:
If $\Delta\models\mathbf{S}\leqslant\mathbf{S'}$ and $\Delta\models\mathbf{S'}\leqslant\mathbf{S}$ then we write $\Delta\models\mathbf{S}\approx\mathbf{S'}$. Similarly if $\Delta\vdash\mathbf{S}\leqslant\mathbf{S'}$ and $\Delta\vdash\mathbf{S'}\leqslant\mathbf{S}$ then we write $\Delta\vdash\mathbf{S}\approx\mathbf{S'}$. We have: (i) For any set Δ of sentences: $\Delta\vdash\mathbf{S}\approx\mathbf{S'}$ implies $\Delta\models\mathbf{S}\approx\mathbf{S'}$. (ii) For any countable set Δ of sentences: $\Delta\models\mathbf{S}\approx\mathbf{S'}$ implies $\Delta\vdash\mathbf{S}\approx\mathbf{S'}$.

This gives us our technique for proving refinement between statements. It is **complete** since if $\Delta\models\mathbf{S}\leqslant\mathbf{S'}$ then there **is** a proof of $\mathbf{S}\leqslant\mathbf{S'}$ from Δ (although the proof may be infinitely long -but only countably infinite!). It is **sound** since if we prove $\mathbf{S}\leqslant\mathbf{S'}$ from Δ then $\Delta\models\mathbf{S}\leqslant\mathbf{S'}$. (See [Karp 64], [Back 80] and [Ward 89b] for the detailed justification of these claims). Completeness means that all refinements are provable, soundness means that all proofs of refinements are reliable.

It is easy to see that $\mathbf{G(w)}$ above can be replaced by any formula **Q**. Thus:
(i) If $\mathbf{S}\leqslant_M\mathbf{S'}$ then $\mathbf{WP(S,Q)\Rightarrow WP(S',Q)}$ holds in **M**.
(ii) If $\Delta\models\mathbf{S}\leqslant\mathbf{S'}$ then $\Delta\models\mathbf{WP(S,Q)\Rightarrow WP(S',Q)}$.
(iii) If Δ is countable and $\Delta\vdash\mathbf{S}\leqslant\mathbf{S'}$ then $\Delta\vdash\mathbf{WP(S,Q)\Rightarrow WP(S',Q)}$.

The initial set of variables on which a program works is called its "initial state space", and similarly the final state space is the set of variables which are active after the program terminates. For a program to be legal it must be possible to determine the initial state space uniquely given the final state space. We write $\mathbf{S{:}V} \to \mathbf{W}$ for a program $\mathbf{S}$ which can legally have $\mathbf{V}$ as the initial state space and $\mathbf{W}$ as the final state space (where $\mathbf{V}$ and $\mathbf{W}$ are finite, non-empty sets of variables).

Lemma: Induction Rule for Recursion:
If Δ is a countable set of sentences, and $\mathbf{S{:}V} \to \mathbf{V}$ and $\mathbf{S'{:}V} \to \mathbf{V}$ are programs with the same initial and final state spaces, and $\mathbf{B}$ a formula with $\mathbf{var(B)} \subseteq \mathbf{V}$ then:
(i) If $\Delta \vdash$ **proc** $\mathbf{X} \equiv \mathbf{S.}^n \leqslant \mathbf{S'}$ for every $\mathbf{n} < \omega$ then $\Delta \vdash$ **proc** $\mathbf{X} \equiv \mathbf{S.} \leqslant \mathbf{S'}$
(ii) $\Delta \vdash$ **proc** $\mathbf{X} \equiv \mathbf{S.}^n \leqslant$ **proc** $\mathbf{X} \equiv \mathbf{S.}$ for every $\mathbf{n} < \omega$.
Proof: See [Ward 89b].

This theorem shows how truncations are used to prove refinements and transformations on recursive programs. We can use it to prove properties of infinitely long formulae by induction on their structure.

Properties_of_WP:

Let $\mathbf{S{:}V} \to \mathbf{W}$ be any statement. If $\mathbf{var(Q_\xi)} \subseteq \mathbf{W}$ for all $\xi < \omega_1$ where $\mathbf{Q}_\xi$ formulae of $\mathbf{L}$, then we can prove the following generalisations of the five properties of the weakest precondition given by Dijkstra in [Dijkstra 76]:

(i) $\mathbf{WP(S,false)} \iff \mathbf{false}$
(ii) $\forall \mathbf{x.}\big((\mathbf{Q}_0 \Rightarrow \mathbf{Q}_1) \Rightarrow (\mathbf{WP(S,Q_0)} \Rightarrow \mathbf{WP(S,Q_1)})\big)$
provided every variable in $\mathbf{W} - \tilde{x}$ is constant in $\mathbf{S}$.
(iii) $\mathbf{WP(S,}\bigwedge_{\xi<\delta}\mathbf{Q}_\xi\mathbf{)} \iff \bigwedge_{\xi<\delta}\mathbf{WP(S,Q_\xi)}$ for $\delta < \omega_1$
(iv) $\bigvee_{\xi<\delta}\mathbf{WP(S,Q_\xi)} \Rightarrow \mathbf{WP(S,}\bigvee_{\xi<\delta}\mathbf{Q}_\xi\mathbf{)}$ $\kappa\kappa\kappa\kappa\kappa$ for $\delta < \omega_1$
(v) If $\mathbf{Q}_i \Rightarrow \mathbf{Q}_{i+1}$ for each $\mathbf{i} < \omega$, then $\mathbf{WP(S,}\bigvee_{i<\omega}\mathbf{Q}_i\mathbf{)} \Rightarrow \bigvee_{i<\omega}\mathbf{WP(S,Q_i)}$

A variable is constant in $\mathbf{S}$ if it does not appear in the list of assigned variables in any atomic description in $\mathbf{S}$.

Property (v) (**continuity**) in general only holds for statements with **bounded** nondeterminacy. A sufficient condition for bounded nondeterminacy is that each atomic description is **finite**:

Defn: The atomic description $\mathbf{x/y.Q}$ is **finite** in the structure $\mathbf{M}$ if $\mathbf{M} \models \mathbf{finite(x,Q)}$ where $\mathbf{finite(x,Q)}$ is

$$\bigvee_{n<\omega} \forall \mathbf{x}_0\mathbf{x}_1\mathbf{\ldots x}_n\big(\bigwedge_{i\leqslant n}\mathbf{Q[x_i/x]} \Rightarrow \bigvee_{i<j\leqslant n}\mathbf{x}_i = \mathbf{x}_j\big)$$

ie there is only a finite number of distinct $\mathbf{x}_i$'s such that $\mathbf{Q[x_i/x]}$ holds, ie there is only a finite number of values for $\mathbf{x}$ for which $\mathbf{Q}$ can be interpreted as true.

We say $\mathbf{x/y.Q}$ is finite in Δ if $\Delta \models \mathbf{finite(x,Q)}$, so (v) holds if each atomic description in $\mathbf{S}$ is finite in Δ.

Theorem: Replacement in statements:
Let $\mathbf{S{:}V} \to \mathbf{W}$ contain the substatement $\mathbf{T{:}V'} \to \mathbf{W'}$. Let $\mathbf{S'{:}V} \to \mathbf{W}$ be the result of replacing $\mathbf{T}$ in $\mathbf{S}$ by $\mathbf{T'}$ where$\mathbf{T'{:}V'} \to \mathbf{W'}$. Then for any countable set of sentences Δ we have

$$\Delta \vdash \mathbf{T} \leqslant \mathbf{T'} \Rightarrow \Delta \vdash \mathbf{S} \leqslant \mathbf{S'}$$

Proof: By induction on the structure of $\mathbf{S}$. The induction is a double induction over (i) The maximum depth of recursion nesting, and (ii) the length of the statement.
To prove the case for recursion we note:
$\Delta \vdash$ **proc** $\mathbf{X} \equiv \mathbf{S}^n. \leqslant$ **proc** $\mathbf{X} \equiv \mathbf{S'}.^n$ for $\mathbf{n} < \omega$
since **proc** $\mathbf{X} \equiv \mathbf{S}.^n$ has a lower depth of recursion nesting than **proc** $\mathbf{X} \equiv \mathbf{S}.$
By the induction rule for recursion we get:
$\Delta \vdash$ **proc** $\mathbf{X} \equiv \mathbf{S'}.^n \leqslant$ **proc** $\mathbf{X} \equiv \mathbf{S'}.$ for $\mathbf{n} < \omega$. Hence
$\Delta \vdash$ **proc** $\mathbf{X} \equiv \mathbf{S}.^n \leqslant$ **proc** $\mathbf{X} \equiv \mathbf{S'}.$ for $\mathbf{n} < \omega$, by transitivity of refinement. Hence
$\Delta \vdash$ **proc** $\mathbf{X} \equiv \mathbf{S}. \leqslant$ **proc** $\mathbf{X} \equiv \mathbf{S'}.$ by induction rule again.
The other cases are simple applications of the induction hypothesis.

This theorem shows that we can replace any substatement of a statement with a refinement of it and get a refinement of the whole statement.

DEFINITIONAL TRANSFORMATIONS

The next stage is to extend the kernel language by means of definitional transformations– for example we define Dijkstra's guarded commands by expressing them in terms of deterministic and nondeterministic selection. We can then prove that two statements in the extended language are equivalent by proving that their kernel representations are equivalent. However a simpler method is to use the kernel representation to derive the weakest precondition of the new statement types; we can then prove equivalence of statements by going directly to the weakest preconditions. In some cases this can result in a great saving of labour. For example the kernel representations of Dijkstra's guarded command is rather complicated while the weakest precondition reduces to a simple extension of the weakest precondition of the **if** statement. The rest of this Chapter will investigate some important properties of **while** loops (which are defined as tail-recursive procedures). We develop induction rules which assist in proving the equivalence of statements involving **while** loops and recursion by considering their "truncations". We prove some generalisations of Dijkstra's theorems for proving termination of **while** loops and extend them to recursive statements. Finally we prove our "Theorem on recursive implementation of specifications" which provides the link between a recursively-defined specification statement and a recursive procedure which implements it. This theorem will prove extremely useful in later chapters, especially in the sections on the derivation of algorithms from their specification.

Assignments and Assertions

We cannot express the assignment $\mathbf{x{:=}x{+}y}$ by the statement $\langle \mathrm{x}\rangle/\langle\rangle.(\mathbf{x{=}x{+}y})$ since this attempts to assign a value to $\mathbf{x}$ such that $\mathbf{x{=}x{+}y}$. This is equivalent to **abort** if $\mathbf{y{\neq}0}$ and assigns an arbitrary value to $\mathbf{x}$ if $\mathbf{y{=}0}$. To get the right effect we need the sequence of two primitive statements: $\langle \mathrm{z}\rangle/\langle\rangle.(\mathbf{z{=}x{+}y});\ \langle \mathrm{x}\rangle/\langle \mathrm{z}\rangle.(\mathbf{x{=}z})$ where $\mathbf{z}$ is a new variable. With this interpretation the assignment $\mathbf{x{:=}x/y}$ will not terminate if $\mathbf{y{=}0}$ and $\mathbf{x{\neq}0}$ initially as there is then no value for $\mathbf{z}$ such that $\mathbf{z.y{=}x}$ –in our model an error is indicated by non-termination. If $\mathbf{y{=}x{=}0}$ initially then $\mathbf{x{:=}x/y}$ will assign an arbitrary value to $\mathbf{x}$.

An assertion $\mathbf{\{R\}}$ is expressed by the statement $\langle\rangle/\langle\rangle.\mathbf{R}$, which cannot change the value of any variable. It acts as a partial **skip** statement -if the initial state satisfies $\mathbf{R}$ then it has no effect, otherwise it acts as an **abort** (ie does not terminate). Note the different role our assertions play to that in Hoare's axiomatic method of program proving. In our formulation, assertions are statements rather then annotations of a program. A proof of a program using his methods corresponds to a proof that a program is refined by the same program with certain assertions added. For example proving $\mathbf{\{R_1\}\ P\ \{R_2\}}$ by Hoare's method corresponds to proving the refinement $\mathbf{\{R_1\};P} \leqslant \mathbf{\{R_1\};P;\{R_2\}}$. Our method is more general than his in the sense that we can prove total correctness: if we prove that $\mathbf{S_1} \leqslant \mathbf{S_2}$ then $\mathbf{S_1}$ terminates whenever $\mathbf{S_2}$ does.

Suppose statement $\mathbf{S}$ contains statement $\mathbf{T}$ within it, ie we have $\mathbf{S{=}...T...}$ and suppose we want to replace $\mathbf{T}$ by $\mathbf{T'}$. If $\mathbf{T}{\leqslant}\mathbf{T'}$ then there is no problem; the replacement theorem shows that we will get a refinement of $\mathbf{S}$. If not then we can try to find an assertion $\mathbf{\{R\}}$ such that $\mathbf{S}{\leqslant}\mathbf{S'}$ where $\mathbf{S'} = \mathbf{...\{R\};T...}$ then if we can prove $\mathbf{\{R\};T}{\leqslant}\mathbf{T'}$ then $\mathbf{S'} \leqslant \mathbf{S''}$ where $\mathbf{S''} = \mathbf{...T'...}$ and then by transitivity we
get $\mathbf{S}{\leqslant}\mathbf{S''}$. This illustrates how assertions can supply information about the conditions under which a statement operates and thus the conditions under which a refinement of the statement is required to work; which may ease the task of proving the refinement. This is one way in which assertion transformations "migrate" useful information through the program.

Weak and Strong Termination

One problem with the particular interpretation of statements as state transformations we are using is that it does not deal with termination in quite the way one intuitively expects. To see this we consider an example taken from [Dijkstra 76]: Let **S** be the statement:

```
proc X ≡ if x≠0 then if x⩾0 then x:=x−1
                else x:=x.(x⩾0) fi;
                 X
             else skip fi.
```

This is a simple tail-recursion which must terminate for any initial value of **x** –positive, negative or zero. However for negative values the interpretation of this statement as a state transformation has $\perp$ in its set of final states. This is interpreted as meaning that the statement is not guaranteed to terminate.

The interpretation function actually formalises strong termination of recursive procedures instead of the usual notion of termination. A procedure is strongly terminating iff for each initial state **s** there is an integer $\mathbf{n}_s$ such that the procedure is guaranteed to terminate in less than $\mathbf{n}_s$ steps. We shall call a procedure that is not strongly terminating weakly terminating.

Termination is always strong when the nondeterminism of the program is bounded: this is a consequence of Konig's Lemma. If a program has bounded nondeterminism then the set of all possible execution paths of a program from some initial state will form a finitely branching tree and if the program is guaranteed to terminate for the given initial state then each branch will be finite. By Konig's Lemma this means that the tree itself is finite and therefore has a finite number of execution paths and thus a limit to the length of its execution paths which is a limit to the number of iterations before termination. Thus termination is strong.

However with an infinitely branching tree Konig's Lemma no longer applies and it is possible for each branch of the tree to be finite but the whole tree to be infinite, for example the execution tree of the program **S** on initial state **x**$= -\mathbf{1}$ is:

-the fact that there could still be unfinished computation after any number of recursive calls does not mean that there must be a nonterminating recursive call.

We cannot solve this by simply changing the approximation ordering since the following program has the same set of approximations $\{\mathbf{s}_0,\perp\}$ for $\mathbf{x}= -\mathbf{1}$ initially but will not be guaranteed to terminate:

```
proc X ≡  if x≠0 then x:=x'.(x' =0 ∨ x' =1); X
               else skip fi.
```

In other words, if we want to be able to deduce the behaviour of a program from the behaviour of all its truncations we shall have to be content with formalising strong termination. We would like to be able to use induction to prove properties of all the truncations and then deduce that the property holds for the general program.

A different solution to the problem is outlined in [Back 80a] where an operational semantics is defined in which the state transformations map states to sets of execution paths. The execution paths are finite or infinite sequences of states. There are three kinds of execution paths:
(i) Terminal paths, which are of the form $\langle \mathbf{s}_1,\ldots,\mathbf{s}_n\rangle$, $\mathbf{n}\geqslant\mathbf{1}$, $\mathbf{s}_i$ proper states.
(ii) Unfinished paths, which are of the form $\langle \mathbf{s}_1,\ldots,\mathbf{s}_n,\perp\rangle$, $\mathbf{n}\geqslant\mathbf{1}$, $\mathbf{s}_i$ proper states.
(iii) Infinite paths, which are of the form $\langle \mathbf{s}_1,\mathbf{s}_2\ldots\rangle$, $\mathbf{s}_i$ proper states.

The approximation ordering on execution paths is:
$\sigma \sqsubseteq \sigma'$ iff either σ is terminal or infinite and $\sigma = \sigma'$
or σ is unfinished and $\bar{\sigma} \leqslant \sigma'$
where $\bar{\sigma}$ is σ with the possibly trailing element $\perp$ removed and $\sigma \leqslant \sigma'$ means σ is an initial segment of σ'.

However, this solution makes the semantics much more complex than the "black box" approach of our denotational semantics which only models the input-output behaviour of a statement rather than the internal details of the computation. Instead we will avoid unbounded nondeterminism by restricting all assignments to choose from a finite set of possibilities with the exception of those which change from an abstract to a concrete state space.

EXTENDING THE LANGUAGE

We will now present the first set of extensions to our kernel language. In the following, $\mathbf{S}_1,\ldots,\mathbf{S}_n$ are statements from $\mathbf{V}$ to $\mathbf{W}$, $\mathbf{x}$ is a list of distinct variables in $\mathbf{V}$ and $\mathbf{Q}$ and $\mathbf{B}_1,\ldots,\mathbf{B}_n$ are formulae of $\mathbf{L}$ with $\mathbf{var(Q)}\subseteq\mathbf{V}\cup\underline{\mathbf{x}}'$. and $\mathbf{var(B}_i\mathbf{)}\subseteq\mathbf{V}$ for $\mathbf{i=1},\ldots,\mathbf{n}$. The symbol "$=_{DF}$" (read "is defined as”) introduces an extension to the language by means of a definitional transformation. The first four extensions are taken from [Back 80], we have built on this by adding recursive functions and unbounded loops which can be terminated from the middle.

Assertions: The assertion $\mathbf{\{Q\}} =_{DF} \langle\rangle/\langle\rangle.\mathbf{Q}$.

$\mathbf{skip} =_{DF} \mathbf{\{true\}}$ and $\mathbf{abort} =_{DF} \mathbf{\{false\}}$

We have: $\mathbf{WP(\{Q\},R)}\kappa\kappa\kappa\kappa\kappa\kappa\kappa\kappa\kappa\kappa\kappa = \mathbf{Q} \wedge \mathbf{R}$, $\mathbf{WP(skip,R)}\kappa\kappa\kappa\kappa\kappa\kappa = \mathbf{R}$ and $\mathbf{WP(abort,R)} = \mathbf{false}$.

Assignment: Suppose $\mathbf{Q}$ be a formula of $\mathbf{L}$ with $\mathbf{var(Q)} \subseteq \mathbf{V} \cup \underline{\mathbf{x}}'$ (where $\mathbf{x}'$ is the list of primed variables corresponding to $\mathbf{x}$) with $\Delta \vdash \mathbf{finite(x',Q)}$. Then the (finite nondeterministic) assignment:

$$\mathbf{x{:=}x'.Q} =_{DF} \mathbf{x'/\langle\rangle.Q;\ x/x'.(x{=}x')}$$

is a statement which assigns new values to the variables in $\mathbf{x}$ in such a way that the condition $\mathbf{Q}$ holds. The requirement $\Delta \vdash \mathbf{finite(x',Q)}$ ensures that there is only a finite number of ways of making the assignment. The simple assignment is defined:

$$\kappa\kappa\kappa\kappa\mathbf{x{:=}t} =_{DF} \mathbf{x{:=}x'.(x'=t)}$$

where $\mathbf{t}$ is a list of terms of $\mathbf{L}$ with $\ell(\mathbf{x}) = \ell(\mathbf{t})$ and $\mathbf{var(t}_i\mathbf{)} \subseteq \mathbf{V}$ for $\mathbf{i{=}1{\ldots}}\ell(\mathbf{t})$.

We have:

$\mathbf{WP(x{:=}x'.Q,R)} = \exists \mathbf{x'.Q} \wedge \forall \mathbf{x'.(Q \Rightarrow R[x'/x])}$

$\mathbf{WP(x{:=}t,R)} = \mathbf{R[t/x]}$

Nondeterministic Selection:

$\underline{\mathbf{if}}\ \mathbf{B}_1 \rightarrow \mathbf{S}_1 \ \square \ldots \square\ \mathbf{B}_n \rightarrow \mathbf{S}_n\ \underline{\mathbf{fi}}$ is defined as:

$\underline{\mathbf{if}}\ \underline{\mathbf{fi}} =_{DF} \mathbf{abort}$

$\underline{\mathbf{if}}\ \mathbf{B}_1 \rightarrow \mathbf{S}_1\ \underline{\mathbf{fi}} =_{DF} \underline{\mathbf{if}}\ \mathbf{B}_1\ \underline{\mathbf{then}}\ \mathbf{S}_1\ \underline{\mathbf{else}}\ \mathbf{abort}\ \underline{\mathbf{fi}}$

$\underline{\mathbf{if}}\ \mathbf{B}_1 \rightarrow \mathbf{S}_1 \ \square\ \mathbf{B}_2 \rightarrow \mathbf{S}_2\ \underline{\mathbf{fi}}$

$=_{DF} \underline{\mathbf{if}}\ \mathbf{B}_1 \wedge \neg\mathbf{B}_2\ \underline{\mathbf{then}}\ \mathbf{S}_1\ \underline{\mathbf{else}}\ \underline{\mathbf{if}}\ \mathbf{B}_2 \wedge \neg\mathbf{B}_1\ \underline{\mathbf{then}}\ \mathbf{S}_2$

$\underline{\mathbf{else}}\ \underline{\mathbf{if}}\ \mathbf{B}_1 \wedge \mathbf{B}_2 \rightarrow \underline{\mathbf{oneof}}\ \mathbf{S}_1 \ \square\ \mathbf{S}_2\ \underline{\mathbf{foeno}}\ \underline{\mathbf{fi}}\ \underline{\mathbf{fi}}\ \underline{\mathbf{fi}}$

$\underline{\mathbf{if}}\ \mathbf{B}_1 \rightarrow \mathbf{S}_1 \ \square \ldots \square\ \mathbf{B}_n \rightarrow \mathbf{S}_n\ \underline{\mathbf{fi}} =_{DF} \underline{\mathbf{if}}\ \mathbf{B}_1 \rightarrow \mathbf{S}_1$

$\square\ \mathbf{B}_2 \vee \ldots \vee \mathbf{B}_n \rightarrow \underline{\mathbf{if}}\ \mathbf{B}_2 \rightarrow \mathbf{S}_2 \ \square \ldots \square\ \mathbf{B}_n \rightarrow \mathbf{S}_n\ \underline{\mathbf{fi}}\ \underline{\mathbf{fi}}$ for $\mathbf{n{>}2}$

For the weakest precondition we calculate that:

$$\mathbf{WP(}\underline{\mathbf{if}}\ \mathbf{B}_1 \rightarrow \mathbf{S}_1 \ \square \ldots \square\ \mathbf{B}_n \rightarrow \mathbf{S}_n\ \underline{\mathbf{fi}}\ \mathbf{,R)} = \bigvee_{1 \leqslant i \leqslant n} \mathbf{B}_i \wedge \bigwedge_{1 \leqslant i \leqslant n} (\mathbf{B}_i \Rightarrow \mathbf{WP(S}_i\mathbf{,R)})$$

We abbreviate $\underline{\mathbf{if}}\ \mathbf{B}\ \underline{\mathbf{then}}\ \mathbf{S}\ \underline{\mathbf{else}}\ \mathbf{skip}\ \underline{\mathbf{fi}}$ by $\underline{\mathbf{if}}\ \mathbf{B}\ \underline{\mathbf{then}}\ \mathbf{S}\ \underline{\mathbf{fi}}$. Note that this differs from $\underline{\mathbf{if}}\ \mathbf{B} \rightarrow \mathbf{S}\ \underline{\mathbf{fi}}$ since the latter does not terminate when $\neg\mathbf{B}$ holds while the former is equivalent to $\mathbf{skip}$.

<u>Blocks:</u> Let **S** be a statement in **V**∪**<u>x</u>**, with **<u>x</u>**∩**V**= ∅. Then the block

<u>begin</u> x: S <u>end</u> $=_{DF}$ **x/⟨⟩.true; S; ⟨⟩/x.true**

is a statement in **V**. The weakest precondition is

WP(<u>begin</u> x:S <u>end</u> ,R)= ∀x.WP(S,R)

The block allows the introduction of temporary variables, these are not true local variables since we do not allow the same name to be used more than once. True local variables will be introduced later. We require all local variables of a block to be properly initialised before their values are referred to, but we will not give any specific scheme (such as that given in [Dijkstra 76]) to guarantee this. Neglecting proper initialisation allows unbounded nondeterminism to creep in. For example the block **<u>begin</u> y: x:=y <u>end</u>** sets the variable **x** to any value.

If the local variable is initialised immediately (eg to the value of the term **t**) then we have the abbreviation:

<u>begin</u> x:=t: S <u>end</u> for **<u>begin</u> x: x:=t; S <u>end</u>**.

<u>Deterministic</u> <u>Iteration:</u> We define a **<u>while</u>** loop by using tail-recursion. If **S:V** → **V** then:
<u>while</u> B <u>do</u> S <u>od</u> $=_{DF}$ **<u>proc</u> X ≡ <u>if</u> B <u>then</u> S; X <u>fi</u>.**

<u>Nondeterministic</u> <u>Iteration:</u> If $\mathbf{S}_1,\ldots,\mathbf{S}_m$**:V** → **V** then:
<u>do</u> $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>od</u>** $=_{DF}$ **<u>while</u>** $\mathbf{B}_1 \vee \ldots \vee \mathbf{B}_m$ **<u>do</u> <u>if</u>** $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>fi</u> <u>od</u>**.

We have: **WP(<u>do</u>** $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>od</u> ,R)** = $\bigvee_{n<\omega}$**WP(<u>do</u>** $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>od</u>**^{n} **,R)**
where **<u>do</u>** $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>od</u>**^{n}
$=_{DF}$ **<u>while</u>** $(\mathbf{B}_1 \vee \ldots \vee \mathbf{B}_m)$ **<u>do</u> <u>if</u>** $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>fi</u> <u>od</u>**^{n}

Note that **<u>do</u>** $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>od</u>**^{0} = **abort**
and **<u>do</u>** $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>od</u>**^{n} ≈ **<u>if</u> BB** → **<u>if</u>** $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>fi</u>;**
<u>do</u> $\mathbf{B}_1 \to \mathbf{S}_1 \square \ldots \square \mathbf{B}_m \to \mathbf{S}_m$ **<u>od</u>**$^{n-1}$
□ ¬**BB** → **skip <u>fi</u>**
for **n>0**, where **BB** = $\mathbf{B}_1 \vee \ldots \vee \mathbf{B}_m$.

<u>Counted</u> <u>Repetition:</u> Suppose **b,s** and **f** are terms with all their variables in **V** and **i** is a variable in **V**. Suppose **S:V** → **V** is a statement which does not assign to **i** or any of the variables in **b,s** or **f**. Suppose we have a binary function + and a total order ⩽. Define the sequence of terms $\mathbf{s}_n$ by:

$\mathbf{s}_0 = \mathbf{b}, \quad \mathbf{s}_{n+1} = \mathbf{s}_n + \mathbf{s}$

Then if the condition $\bigvee_{n<\omega} \neg(\mathbf{s}_n \leqslant \mathbf{f})$ is true initially we can use **b**, **f**, **s**, **i**, + and ⩽ to define a **<u>for</u>**

loop as follows:
for i from b step s to f do S od $=_{DF}$ **begin i:=b; while i⩽f do S; i:=i+s od end**

This means that any **for** statement is guaranteed to terminate provided the statement inside the loop terminates. Note that the counting variable **i** is a local variable to the block surrounding **S** and so is not in the state space outside the loop.

Exit Statements:

Our programming language will include statements of the form **exit(n)**, where **n** is an integer, (not a variable) which occur within loops of the form **do S od** where **S** is a statement. These were described in [Knuth 74] and more recently in [Taylor 84]. They are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form **exit(n)** which causes the program to exit the **n** enclosing loops. To simplify the exposition we will disallow **exit**s from which would terminate a block or a loop other than an unbounded loop).

Previously the only formal treatments of **exit** statements have treated them in the same way as unstructured **goto** statements by adding "continuations" to the denotational semantics of all the other statements. This adds greatly to the complexity of the semantics and also means that all the results obtained prior to this modification will have to be re-proved with respect to the new semantics. The approach taken in this thesis, which does not seem to have been tried before, is to express every program which uses **exit** statements and unbounded loop in terms of the kernel language we have already developed. This means that the new statements will not change the denotational semantics of the kernel so all the transformations developed without reference to **exit** statements will still apply in the more general case. In fact we will be making much use of the transformations derived without reference to **exit**s in the derivation of transformations of statements which use the **exit** statement.

The interpretation of these statements in terms of the kernel language is as follows:

We have an integer variable **depth** which records the current depth of nesting of loops. At the beginning of the program we have **depth:=0** and each **exit** statement **exit(k)** is translated:

depth:=depth−k

since it changes the depth of "current execution" by moving out of **k** enclosing loops. To prevent any more statements at the current depth being executed after an **exit** statement has been executed we surround all statements by "guards" which are **if** statements which will test **depth** and only allow the statement to be executed if **depth** has the correct value. Each unbounded loop **do S od** is translated:

depth:=n ; while depth=n do guard$_n$**(S) od**

where **n** is an integer constant representing the depth of the loop (**1** for an outermost loop, **2** for double nested loops etc.) and **guard**$_n$**(S)** is the statement **S** with each component statement guarded so that if the depth is changed by an **exit** statement then no more statements in the loop will be executed and the loop will terminate. Formally we define **guard**$_n$**(S)** by induction on the structure of **S** as follows:

```
            guard_n({Q}) = {depth=n ⇒ Q}
          guard_n(x:=x′.Q) = if depth=n then x:=x′.Q fi
            guard_n(S1;S2) = guard_n(S1); guard_n(S2)
guard_n(if B then S1 else S2fi) = if B then guard_n(S1) else guard_n(S2) fi
   guard_n(while B do S1 od) = if depth=n then while B do guard_n(S1) od fi
    guard_n(begin x:  S1 end) = begin x: guard_n(S1) end
       guard_n(proc X ≡ S1.) = if depth=n then proc X ≡ guard_n(S1). fi
              guard_n(X) = if depth=n then X fi
          guard_n(exit(k)) = if depth=n then depth:=depth−k fi
         guard_n(do S od) = if depth=n then depth:=n+1;
                                  while depth=n+1 do guard_{n+1}(S1) od fi
```

Note:
(i) The guards on the statements inside a **while** loop are not strictly needed (except within further **do** loops) since we are not allowed to **exit** out of a **while** loop or a **for** loop.
(ii) **exit** statements are guarded like any other assignment.
(iii) We are not allowed to **exit** from a block.

Thus the weakest precondition of a program description involving **do** loops is:

WP(S,R) $\iff$ **WP(depth:=0; guard**$_0$**(S), R)**

An **exit** statement which attempts to leave more loops than the number of loops enclosing it causes termination of the whole program.

The important property of a guarded statement is that it will only be executed if **depth** has the correct value. Thus: **{depth≠n}; guard**$_n$**(S)** $\approx$ **skip**
So for exampl: **exit; S** $\approx$ **exit**
since: **{depth=n}; guard**$_n$**(exit; S)**
$\approx$ **{depth=n}; if depth=n then depth:=depth-1 fi; guard**$_n$**(S)**
$\approx$ **{depth=n}; depth:=depth-1; {depth≠n}; guard**$_n$**(S)**
$\approx$ **{depth=n}; depth:=depth-1; skip**
$\approx$ **{depth=n}; if depth=n then depth:=depth-1 fi**
$\approx$ **{depth=n}; guard**$_n$**(exit)**

PROPERTIES OF WHILE

We define a "truncated **while** loop" **while B do S od**^{n} to be the nth truncation of the tail-recursive procedure ie:

while B do S od^0 = **abort**

while B do S od^{n+1} = **if B then S; while B do S od**^{n} **fi**.

From this we get the induction rule for iteration:

(i) If $\Delta \vdash$ **while B do S od**$^n \leqslant \mathbf{S}'$ for every $\mathbf{n} < \omega$ then $\Delta \vdash$**while B do S od** $\leqslant \mathbf{S}'$.

(ii) $\Delta \vdash$ **while B do S od**$^n \leqslant$ **while B do S od** for every $\mathbf{n} < \omega$.

This was proved by Back in [Back 80] for his version of **while** loop. However, we have found the following much more general induction rule for recursion to be more useful:

Lemma: General Induction Rule for Recursion:
If $\Delta \vdash \mathbf{S}^n \leqslant \mathbf{S}'$ for all $\mathbf{n} < \omega$ then $\Delta \vdash \mathbf{S} \leqslant \mathbf{S}'$
where $\mathbf{S}^n$ is **S** with each procedure replaced by its **n**'th truncation -so that if **S** contains **proc** $\mathbf{X} \equiv \mathbf{S}_1$**.** then $\mathbf{S}^n$ contains **proc** $\mathbf{X} \equiv \mathbf{S}_1.^n$.

Proof: We will in fact prove $\Delta \vdash \mathbf{S} \approx \bigvee_{n<\omega} \mathbf{S}^n$, then $\Delta \vdash \mathbf{S}^n \leqslant \mathbf{S}'$ for all $\mathbf{n} < \omega$ gives $\Delta \vdash \mathbf{S} \approx \bigvee_{n<\omega} \mathbf{S}^n \leqslant \mathbf{S}'$ by the inference rule for infinite disjunction.
The proof is by induction on the structure of **S** using the following lexical order:
(i) Depth of recursion nesting.
(ii) Length of program text.

Consider Cases:
(i) **S** is **x/y.Q**. Then $\mathbf{S}^n = \mathbf{S}$ and the result is trivial.
(ii) $\mathbf{WP}(\mathbf{S}_1;\mathbf{S}_2,\mathbf{R}) \iff \mathbf{WP}(\mathbf{S}_1,\mathbf{WP}(\mathbf{S}_2,\mathbf{R})) \iff \bigvee_{n<\omega} \mathbf{WP}(\mathbf{S}_1^n, \bigvee_{m<\omega} \mathbf{WP}(\mathbf{S}_2^m,\mathbf{R}))$
$\iff \bigvee_{n,m<\omega} \mathbf{WP}(\mathbf{S}_1^n,\mathbf{WP}(\mathbf{S}_2^m,\mathbf{R})) \iff \bigvee_{n,m<\omega} \mathbf{WP}(\mathbf{S}_1^n;\mathbf{S}_2^m,\mathbf{R})$
by properties of **WP** and induction hypothesis.
For all $\mathbf{n},\mathbf{m} < \omega$,
$\mathbf{WP}(\mathbf{S}_1^n,\mathbf{WP}(\mathbf{S}_2^m,\mathbf{R})) \Rightarrow \mathbf{WP}(\mathbf{S}_1^{n+m+1},\mathbf{WP}(\mathbf{S}_2^{n+m+1},\mathbf{R}))$ since $\mathbf{S}_1^n \leqslant \mathbf{S}_1^{n+m+1}$ etc.
$\Rightarrow \bigvee_{n<\omega} \mathbf{WP}(\mathbf{S}_1^n,\mathbf{WP}(\mathbf{S}_2^n,\mathbf{R})) \iff \bigvee_{n<\omega} \mathbf{WP}((\mathbf{S}_1;\mathbf{S}_2)^n,\mathbf{R})$

So $\mathbf{WP}(\mathbf{S}_1;\mathbf{S}_2,\mathbf{R}) \Rightarrow \bigvee_{n<\omega} \mathbf{WP}((\mathbf{S}_1;\mathbf{S}_2)^n,\mathbf{R})$ by the inference rule for disjunction.
Conversely for any $\mathbf{n} < \omega$: $\mathbf{WP}((\mathbf{S}_1;\mathbf{S}_2)^n,\mathbf{R}) \iff \mathbf{WP}(\mathbf{S}_1^n,\mathbf{WP}(\mathbf{S}_2^n,\mathbf{R}))$
$\Rightarrow \bigvee_{n,m<\omega} \mathbf{WP}(\mathbf{S}_1^n,\mathbf{WP}(\mathbf{S}_2^m,\mathbf{R})) \iff \mathbf{WP}(\mathbf{S}_1;\mathbf{S}_2,\mathbf{R})$

So $\mathbf{WP(S_1;S_2,R)} \iff \bigvee_{n<\omega}\mathbf{WP((S_1;S_2)^{\mathit{n}},R)}$.

(iii) $\mathbf{WP(}\ \underline{\mathbf{oneof}}\ \mathbf{S_1} \ \Box\ \mathbf{S_2}\ \underline{\mathbf{foeno}}\ \mathbf{,R)}$
$\iff \bigvee_{n<\omega}\mathbf{WP(S_1^{\mathit{n}},R)} \wedge \bigvee_{m<\omega}\mathbf{WP(S_2^{\mathit{m}},R)}$
$\iff \bigvee_{n,m<\omega}\big(\mathbf{WP(S_1^{\mathit{n}},R)} \wedge \mathbf{WP(S_2^{\mathit{m}},R)}\big)$ by the general distributive law in $\mathbf{L}_{\omega_1\omega}$.
$\iff \bigvee_{n<\omega}\big(\mathbf{WP(}\ \underline{\mathbf{oneof}}\ \mathbf{S_1} \ \Box\ \mathbf{S_2}\ \underline{\mathbf{foeno}}^{nn}\mathbf{,R)}\big)$
by an argument similar to the previous case.

(iv) $\mathbf{WP(}\underline{\mathbf{if}}\ \mathbf{B}\ \underline{\mathbf{then}}\ \mathbf{S_1}\ \underline{\mathbf{else}}\ \mathbf{S_2}\ \underline{\mathbf{fi}}\ \mathbf{,R)}$
$\iff \big(\mathbf{B{\Rightarrow}WP(S_1,R)}\big) \wedge \big(\neg\mathbf{B{\Rightarrow}WP(S_2,R)}\big)$
$\iff \big(\mathbf{B{\Rightarrow}} \bigvee_{n<\omega}\mathbf{WP(S_1^{\mathit{n}},R)}\big) \wedge \big(\neg\mathbf{B{\Rightarrow}} \bigvee_{m<\omega}\mathbf{WP(S_2^{\mathit{m}},R)}\big)$
$\iff \bigvee_{n,m<\omega}\big(\big(\mathbf{B{\Rightarrow}WP(S_1^{\mathit{n}},R)}\big) \wedge \big(\neg\mathbf{B{\Rightarrow}WP(S_2^{\mathit{m}},R)}\big)\big)$
$\iff \bigvee_{n<\omega}\big(\big(\mathbf{B{\Rightarrow}WP(S_1^{\mathit{n}},R)}\big) \wedge \big(\neg\mathbf{B{\Rightarrow}WP(S_2^{\mathit{n}},R)}\big)\big)$
$\iff \bigvee_{n<\omega}\mathbf{WP(}\ \underline{\mathbf{if}}\ \mathbf{B}\ \underline{\mathbf{then}}\ \mathbf{S_1}\ \underline{\mathbf{else}}\ \mathbf{S_2}\ \underline{\mathbf{fi}}^{n}\ \mathbf{,R)}$

(v) $\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1.}\ \mathbf{,R)} \iff \bigvee_{n<\omega}\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1.}^{n}\ \mathbf{,R)}$

Claim: $\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1.}^{n}\ \mathbf{,R)} \iff \bigvee_{k<\omega}\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1^{\mathit{k}}.}^{n}\ \mathbf{,R)}$
Proof of claim: by induction on **n**, for **n=0** both sides are **false**.
Suppose true for **n**
$\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1.}^{n+1}\ \mathbf{,R)} \iff \mathbf{WP(S_1[}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1.}^{n}\mathbf{/X],\ R)}$
$\iff \mathbf{WP(S_1,R)[WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1.}^{n}\ \mathbf{,R)/WP(X,R)]}$
$\iff \mathbf{WP(S_1,R)[}\bigvee_{k<\omega}\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1^{\mathit{k}}.}^{n}\ \mathbf{,R)/WP(X,R)]}$ by ind hyp.
$\iff \bigvee_{k<\omega}\mathbf{WP(S_1,R)[}\bigvee_{k<\omega}\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1^{\mathit{k}}.}^{n}\ \mathbf{,R)/WP(X,R)]}$
by the main induction hypothesis on $\mathbf{S_1}$ since $\mathbf{S_1}$ has a lower depth of recursion nesting than **S**.
$\iff \bigvee_{k<\omega}\mathbf{WP(S_1,R)[WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1^{\mathit{k}}.}^{n}\ \mathbf{,R)/WP(X,R)]}$
by a simple induction on $\mathbf{S_1}$.
$\iff \bigvee_{k<\omega}\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1^{\mathit{k}}.}^{n+1}\ \mathbf{,R)}$
which proves the claim.

So $\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1.}\ \mathbf{,R)} \iff \bigvee_{n<\omega}\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1.}^{n}\ \mathbf{,R)}$ by the induction rule for recursion.
$\iff \bigvee_{n,k<\omega}\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1^{\mathit{k}}.}^{n}\ \mathbf{,R)}$ from the claim
$\iff \bigvee_{k<\omega}\mathbf{WP(}\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1^{\mathit{k}}.}^{k}\ \mathbf{,R)}$
since $\mathbf{S_1^{\mathit{k}}} \leqslant \mathbf{S_1^{\mathit{k}+1}}$ and $\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1^{\mathit{k}}.}^{n} \leqslant \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S_1^{\mathit{k}}.}^{n+1}$

This result gives rise to a general induction rule for iteration by replacing the **while** loops by the equivalent recursive procedures.

PROVING TERMINATION

In this section we prove some important theorems which show how to prove that a loop preserves an invariant, and which show how termination of an iterative or recursive statement can be proved by using a well-founded order relation on the state and using a function on the variables which yields a smaller value than the original value for each recursive call.

Dijkstra's "Basic Theorems"

The following two theorems were derived by Dijkstra in [Dijkstra 76] for his version of the weakest precondition. We will give proofs within $\mathbf{L}_{\omega_1\omega}$ for our **WP**.
Let $\mathbf{BB}{=}\mathbf{B}_1 \vee \ldots \vee \mathbf{B}_n$, $\mathbf{IF}{=}\underline{\mathbf{if}}\ \mathbf{B}_1 \ \rightarrow \mathbf{S}_1 \ \Box \ \ldots \ \Box \ \mathbf{B}_n \ \rightarrow \mathbf{S}_n\ \underline{\mathbf{fi}}$, $\mathbf{DO}{=}\underline{\mathbf{do}}\ \mathbf{B}_1 \ \rightarrow \mathbf{S}_1 \ \Box \ \ldots \ \Box \ \mathbf{B}_n \ \rightarrow \mathbf{S}_n\ \underline{\mathbf{od}}$.

Theorem A: $\big(\mathbf{Q}{\Rightarrow}\mathbf{BB}\big) \wedge \bigwedge_{1\leqslant j\leqslant n}\big(\mathbf{Q}\wedge\mathbf{B}_j \Rightarrow \mathbf{WP}(\mathbf{S}_j,\mathbf{R})\big) \iff \mathbf{Q}{\Rightarrow}\mathbf{WP}(\mathbf{IF},\mathbf{R})$.
This theorem will enable us to prove things about the **if** statement by dealing with each guarded command in turn.

Proof: $\big(\mathbf{Q}{\Rightarrow}\mathbf{BB}\big) \wedge \bigwedge_{1\leqslant j\leqslant n}\big(\mathbf{Q}\wedge\mathbf{B}_j \Rightarrow \mathbf{WP}(\mathbf{S}_j,\mathbf{R})\big)$
$\iff \big(\mathbf{Q}{\Rightarrow}\mathbf{BB}\big) \wedge \bigwedge_{1\leqslant j\leqslant n}\big(\mathbf{Q}{\Rightarrow} \big(\mathbf{B}_j \Rightarrow \mathbf{WP}(\mathbf{S}_j,\mathbf{R})\big)\big)$
$\iff \big(\mathbf{Q}{\Rightarrow}\mathbf{BB}\big) \wedge \mathbf{Q}{\Rightarrow} \bigwedge_{1\leqslant j\leqslant n}\big(\mathbf{B}_j \Rightarrow \mathbf{WP}(\mathbf{S}_j,\mathbf{R})\big)$
$\iff \mathbf{Q}{\Rightarrow} \big(\mathbf{BB} \wedge \bigwedge_{1\leqslant j\leqslant n}\big(\mathbf{B}_j \Rightarrow \mathbf{WP}(\mathbf{S}_j,\mathbf{R})\big)\big)$
$\iff \mathbf{Q}{\Rightarrow}\mathbf{WP}(\mathbf{IF},\mathbf{R})$

Cor: $\big(\mathbf{Q}\wedge\mathbf{B} \Rightarrow \mathbf{WP}(\mathbf{S}_1,\mathbf{R})\big) \wedge \big(\mathbf{Q}\wedge \neg\mathbf{B} \Rightarrow \mathbf{WP}(\mathbf{S}_2,\mathbf{R})\big)$
$\iff \mathbf{Q} \Rightarrow \big(\big(\mathbf{B}{\Rightarrow}\mathbf{WP}(\mathbf{S}_1,\mathbf{R})\big) \wedge \big(\neg\mathbf{B}{\Rightarrow}\mathbf{WP}(\mathbf{S}_2,\mathbf{R})\big)\big)$
$\iff \mathbf{Q} \Rightarrow \mathbf{WP}(\underline{\mathbf{if}}\ \mathbf{B}\ \underline{\mathbf{then}}\ \mathbf{S}_1\ \underline{\mathbf{else}}\ \mathbf{S}_2\ \underline{\mathbf{fi}},\ \mathbf{R})$

Theorem B: If $\mathbf{P}\wedge\mathbf{BB} \Rightarrow \mathbf{WP}(\mathbf{IF},\mathbf{P})$ then $\mathbf{P}\wedge\mathbf{WP}(\mathbf{DO},\mathbf{true}) \Rightarrow \mathbf{WP}(\mathbf{DO},\mathbf{P}\wedge \neg\mathbf{BB})$.
This says that if a loop terminates and if the body of the loop preserves the invariant **P** then if the loop is started in a state which satisfies **P** then it preserves **P** and terminates in a state with **¬BB** true.

Proof: $\mathbf{WP}(\mathbf{DO},\mathbf{true}) \iff \bigvee_{n<\omega}\mathbf{WP}(\mathbf{DO}^n,\mathbf{true})$, $\mathbf{DO}^{n+1} \approx \underline{\mathbf{if}}\ \mathbf{BB}\ \underline{\mathbf{then}}\ \mathbf{IF};\ \mathbf{DO}^n\ \underline{\mathbf{fi}}$

We have $\mathbf{P} \wedge \mathbf{BB} \Rightarrow \mathbf{WP(IF,P)}$

$$\begin{aligned}\mathbf{P} \wedge \mathbf{WP(DO,true)} &\iff \mathbf{P} \wedge \bigvee_{n<\omega} \mathbf{WP(DO}^n\mathbf{,true)}\\ &\iff \bigvee_{n<\omega}\big(\mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF;DO}^{n-1}\mathbf{,true)}\big) \wedge \big(\neg\mathbf{BB} \Rightarrow \mathbf{true}\big)\big)\\ &\iff \bigvee_{n<\omega}\big(\mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF;DO}^{n-1}\mathbf{,true)}\big)\big)\end{aligned}$$

Claim: $\mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF;DO}^n\mathbf{,true)}\big) \Rightarrow \mathbf{WP(DO}^{n+1}\mathbf{,P} \wedge \neg\mathbf{BB)}$ for all $\mathbf{n} < \omega$.
Then $\mathbf{P} \wedge \mathbf{WP(DO,true)} \Rightarrow \bigvee_{n<\omega} \mathbf{WP(DO}^n\mathbf{,P} \wedge \neg\mathbf{BB)} \iff \mathbf{WP(DO,P} \wedge \neg\mathbf{BB)}$ as required.

Proof of claim: By induction on $\mathbf{n}$.
For $\mathbf{n=0}$: $\mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF;DO}^0\mathbf{,true)}\big) \iff \mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{false}\big) \iff \mathbf{P} \wedge \neg\mathbf{BB}$

$$\begin{aligned}\mathbf{WP(DO}^1\mathbf{,P} \wedge \neg\mathbf{BB)} &\iff \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF;DO}^0\mathbf{,P} \wedge \neg\mathbf{BB)}\big) \wedge \big(\neg\mathbf{BB} \Rightarrow \mathbf{P} \wedge \neg\mathbf{BB}\big)\\ &\iff \big(\mathbf{BB} \Rightarrow \mathbf{false}\big) \wedge \big(\neg\mathbf{BB} \Rightarrow \mathbf{P}\big) \iff \mathbf{P} \wedge \neg\mathbf{BB}\end{aligned}$$

So suppose the result holds for $\mathbf{n}$:
$\mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF;DO}^{n+1}\mathbf{,true)}\big) \iff \mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF,} \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF;DO}^n\mathbf{,true)}\big)\,\mathbf{)}\big)$
We have $\mathbf{P} \wedge \mathbf{BB} \Rightarrow \mathbf{WP(IF,P)}$ so: $\Rightarrow \mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF,P)} \wedge \mathbf{WP(IF,}\big(\mathbf{BB} \Rightarrow \mathbf{WP(IF;DO}^n\mathbf{,true)}\big)\,\mathbf{)}\big)$

$\Rightarrow \mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF,\ P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF;DO}^n\mathbf{,true)}\big)\,\mathbf{)}\big)$

$\Rightarrow \mathbf{P} \wedge \big(\mathbf{BB} \Rightarrow \mathbf{WP(IF,\ WP(DO}^{n+1}\mathbf{,P} \wedge \neg\mathbf{BB)\ )}\big)$

by induction hypothesis.

$\Rightarrow \mathbf{WP(DO}^{n+2}\mathbf{,P} \wedge \neg\mathbf{BB)}$

which proves the result by induction.

<u>**Cor:**</u> If $\bigwedge_{1\leqslant j\leqslant n}\big(\mathbf{P} \wedge \mathbf{B}_j \Rightarrow \mathbf{WP(S}_j\mathbf{,P)}\big)$ then $\mathbf{P} \wedge \mathbf{WP(DO,true)} \Rightarrow \mathbf{WP(DO,P} \wedge \neg\mathbf{BB)}$

Proof: Use Theorem A with $\mathbf{Q} = \mathbf{P} \wedge \mathbf{BB}$ and $\mathbf{R} = \mathbf{P}$, the theorem gives $\mathbf{P} \wedge \mathbf{BB} \iff \mathbf{WP(IF,P)}$ and
Theorem B then gives $\mathbf{P} \wedge \mathbf{WP(DO,true)} \Rightarrow \mathbf{WP(DO,P} \wedge \neg\mathbf{BB)}$.

Our next theorem will give us a method by which we can prove that a loop terminates provided each iteration of the body decreases the value of some function of the state under some well-founded ordering.

<u>**Defn:**</u> A partial order $\preccurlyeq$ on some set Γ is **well-founded** if every non-empty subset of Γ has a minimal element under $\preccurlyeq$. ie:

$$\forall \Gamma' \subseteq \Gamma. \big(\Gamma' \neq \varnothing \Rightarrow \exists \zeta \in \Gamma'. \forall \lambda \in \Gamma' (\zeta \preccurlyeq \lambda)\big).$$

We write $\zeta \prec \lambda$ if $\zeta \preccurlyeq \lambda \wedge \zeta \neq \lambda$. We also define $\mathbf{minimal}(\zeta)$ to mean "ζ is a minimal element" ie:

$$\mathbf{minimal}(\zeta) =_{DF} \forall \zeta' \in \Gamma. \big(\zeta' \preccurlyeq \zeta \Rightarrow \zeta' = \zeta\big)$$

__Theorem:__ **Proving Termination:** If $\preccurlyeq$ is a well-founded partial order on some set Γ and $\mathbf{t}$ is a term giving values in Γ and $\mathbf{t}_0$ is a variable which does not occur in **IF** then if
(i) $\mathbf{P}\wedge\mathbf{BB} \Rightarrow \mathbf{WP(IF,P)}$ and
(ii) $\forall\mathbf{t}_0.\big((\mathbf{P}\wedge\mathbf{BB}\wedge\mathbf{t}\preccurlyeq\mathbf{t}_0\big) \Rightarrow \mathbf{WP(IF,t}\prec\mathbf{t}_0))$
then $\mathbf{P}\Rightarrow\mathbf{WP(DO,true)}$.

Proof: $\mathbf{WP(DO,true)} \iff \bigvee_{n<\omega}\mathbf{WP(DO}^n\mathbf{,true)} \iff \bigvee_{n<\omega}\big(\mathbf{BB} \Rightarrow \mathbf{WP(IF,WP(DO}^{n-1}\mathbf{,true)))}\big)$
If λ is any minimal element of Γ then $\mathbf{t}\prec\lambda$ is false whatever the value of $\mathbf{t}$ is. Also $\mathbf{t}\preccurlyeq\lambda \iff \mathbf{t}=\lambda$ so putting $\mathbf{t}_0=\lambda$ in (ii) gives

$$\big(\mathbf{P}\wedge\mathbf{BB}\wedge\mathbf{t}=\lambda\big) \Rightarrow \mathbf{WP(IF,t}\prec\lambda) \iff \mathbf{WP(IF,false)} \iff \mathbf{false}$$

This is true for all λ such that $\mathbf{minimal}(\lambda)$ holds ie

$$\forall\lambda.\,\big((\mathbf{minimal}(\lambda)\wedge\mathbf{P}\wedge\mathbf{BB}\wedge\mathbf{t}=\lambda\big)\Rightarrow\mathbf{false}\big)$$

ie $\mathbf{minimal(t)}\wedge\mathbf{P}\wedge\mathbf{BB} \Rightarrow \mathbf{false}$
ie $\neg\big(\mathbf{minimal(t)}\wedge\mathbf{P}\wedge\mathbf{BB}\big)$

ie (iii) $\mathbf{P}\wedge\mathbf{BB} \Rightarrow \neg\mathbf{minimal(t)}$.

Claim: For any $\lambda\in\Gamma$, $\mathbf{P}\wedge\mathbf{t}\preccurlyeq\lambda \Rightarrow \mathbf{WP(DO,true)}$
Proof of claim:
$\mathbf{WP(DO^1,true)} \iff \mathbf{BB}\Rightarrow\mathbf{WP(IF,WP(DO^0,true))}$
$\iff \mathbf{BB}\Rightarrow\mathbf{WP(IF,false)} \iff \mathbf{BB}\Rightarrow\mathbf{false} \iff \neg\mathbf{BB}$
$\mathbf{WP(DO^1,true)}\Rightarrow\mathbf{WP(DO,true)}$ so $\neg\mathbf{BB}\Rightarrow\mathbf{WP(DO,true)}$

(a): If λ is a minimal element of Γ then:
From (iii) we have $\mathbf{P}\wedge\mathbf{BB}\Rightarrow\neg\mathbf{minimal(t)}$. $\neg\mathbf{minimal(t)} \Rightarrow \big(\lambda\preccurlyeq\mathbf{t}\Rightarrow\lambda\prec\mathbf{t}\big)$ so
$\mathbf{P}\wedge\mathbf{BB}\Rightarrow\neg\mathbf{minimal(t)} \Rightarrow \neg\big(\mathbf{P}\wedge\mathbf{BB}\big) \vee \big(\lambda\preccurlyeq\mathbf{t}\Rightarrow\lambda\prec\mathbf{t}\big)$
$\iff \neg\mathbf{P} \vee \neg\mathbf{BB} \vee \big(\lambda\preccurlyeq\mathbf{t}\Rightarrow\lambda\prec\mathbf{t}\big)$
$\iff \big(\mathbf{P}\wedge\neg\big(\lambda\preccurlyeq\mathbf{t}\Rightarrow\lambda\prec\mathbf{t}\big)\big)\Rightarrow\neg\mathbf{BB}$
$\iff \big(\mathbf{P}\wedge\neg\big(\mathbf{t}\prec\lambda \vee \lambda\prec\mathbf{t}\big)\big)\Rightarrow\neg\mathbf{BB}$
$\iff \big(\mathbf{P}\wedge\neg\big(\mathbf{t}\prec\lambda \vee \lambda\prec\mathbf{t}\big)\big)\Rightarrow\neg\mathbf{BB}$
$\iff \big(\mathbf{P}\wedge\lambda\preccurlyeq\mathbf{t}\wedge\mathbf{t}\preccurlyeq\lambda\big)\Rightarrow\neg\mathbf{BB}$

λ is a minimal element so $\mathbf{t}\preccurlyeq\lambda \Rightarrow \mathbf{t}=\lambda \Rightarrow \lambda\preccurlyeq\mathbf{t}$. So
$\mathbf{P}\wedge\mathbf{BB}\Rightarrow\neg\mathbf{minimal(t)} \Rightarrow \big(\mathbf{P}\wedge\lambda\preccurlyeq\mathbf{t}\big)\Rightarrow\neg\mathbf{BB}$.
So $\mathbf{P}\wedge\mathbf{BB}\Rightarrow\neg\mathbf{minimal(t)} \Rightarrow \big(\mathbf{P}\wedge\lambda\preccurlyeq\mathbf{t}\big)\Rightarrow\mathbf{WP(DO,true)}$ as required.

(b): Suppose λ is not a minimal element of Γ and suppose the result holds for all $\delta \prec \lambda$. Then:
$\big(\mathbf{P} \wedge \mathbf{BB} \wedge \mathbf{t} \preccurlyeq \lambda\big) \Rightarrow \mathbf{WP(IF,\ P} \wedge \mathbf{t} \prec \lambda\mathbf{)}$ by (i) and (ii).
$\quad\Rightarrow \mathbf{WP(IF,\ P} \wedge\ \exists\delta\boldsymbol{.}\big(\delta \prec \lambda\ \wedge \mathbf{t} \preccurlyeq \delta\big)\mathbf{)}$ since λ is not minimal.
$\quad\Rightarrow \mathbf{WP(IF,}\ \exists\delta\boldsymbol{.}\big(\delta \prec \lambda\ \wedge \mathbf{P} \wedge \mathbf{t} \preccurlyeq \delta\big)\mathbf{)}$ since δ does not occur free in $\mathbf{P}$.

$\mathbf{P} \wedge\ \delta \prec \lambda\ \wedge \mathbf{t} \preccurlyeq \delta \Rightarrow \mathbf{WP(DO,true)}$ by assumption.
So $\big(\mathbf{P} \wedge \mathbf{BB} \wedge \mathbf{t} \preccurlyeq \lambda\big) \Rightarrow \mathbf{WP(IF,}\ \exists\delta\boldsymbol{.}\big(\delta \prec \lambda\ \wedge \mathbf{WP(DO,true)}\big)\mathbf{)}$
$\quad\Rightarrow \mathbf{WP(IF,\ WP(DO,true)} \wedge\ \exists\delta\boldsymbol{.}\delta \prec \lambda\mathbf{)}$
since δ does not occur free in $\mathbf{WP(DO,true)}$
$\quad\Rightarrow \mathbf{WP(IF,\ WP(DO,true))}$
since $\exists\delta\boldsymbol{.}\delta \prec \lambda$ is true as λ is not minimal.
$\big(\mathbf{P} \wedge\ \neg\mathbf{BB} \wedge \mathbf{t} \preccurlyeq \lambda\big) \Rightarrow \neg\mathbf{BB}$
Combining the two implications:
$\big(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda\big) \quad \Rightarrow \neg\mathbf{BB} \vee \mathbf{WP(IF,WP(DO,true))}$
$\quad\Longleftrightarrow\ \mathbf{BB} \Rightarrow \mathbf{WP(IF,WP(DO,true))}$
$\quad\Longleftrightarrow\ \mathbf{BB} \Rightarrow \mathbf{WP(IF,}\bigvee_{n<\omega}\mathbf{WP(DO}^n\mathbf{,true))}$
$\quad\Longleftrightarrow\ \bigvee_{n<\omega}\big(\mathbf{BB} \Rightarrow \mathbf{WP(IF,WP(DO}^n\mathbf{,true))}\big)$
$\quad\Longleftrightarrow\ \bigvee_{n<\omega}\mathbf{WP(DO}^{n+1}\mathbf{,true)}$
$\quad\Longleftrightarrow\ \mathbf{WP(DO,true)}$ as required.

The proof of the claim is by a form of "well-founded induction":
Consider the set $\Gamma' = \{\lambda \in \Gamma | \neg\big(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{WP(DO,true)}\big)\}$. Suppose this is non-empty, then since $\preccurlyeq$ is well-founded there exists a minimal element $\zeta \in \Gamma'$. ζ is not a minimal element of Γ by **(a)** above. If $\delta \prec \zeta$ then $\big(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \delta \Rightarrow \mathbf{WP(DO,true)}\big)$ holds. But then by **(b)** above we have $\big(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \zeta \Rightarrow \mathbf{WP(DO,true)}\big)$ holds. Contradiction.
So Γ' must be empty and so $\big(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{WP(DO,true)}\big)$ holds for all $\lambda \in \Gamma$ which proves the claim.

Finally to prove the theorem we start with the claim:
$\forall\lambda \in \Gamma\boldsymbol{.}\big(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{WP(DO,true)}\big)$
$\quad\Longleftrightarrow\ \forall\lambda \in \Gamma\boldsymbol{.}\big(\neg\big(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda\big) \vee \mathbf{WP(DO,true)}\big)$
$\quad\Longleftrightarrow\ \forall\lambda \in \Gamma\boldsymbol{.}\big(\neg\mathbf{P} \vee \lambda \prec \mathbf{t} \vee \mathbf{WP(DO,true)}\big)$
$\quad\Longleftrightarrow\ \forall\lambda \in \Gamma\boldsymbol{.}\big(\neg\big(\mathbf{P} \wedge\ \neg\mathbf{WP(DO,true)}\big) \vee \lambda \prec \mathbf{t}\big)$
$\quad\Longleftrightarrow\ \forall\lambda \in \Gamma\boldsymbol{.}\big(\big(\mathbf{P} \wedge\ \neg\mathbf{WP(DO,true)}\big) \Rightarrow \lambda \prec \mathbf{t}\big)$
$\quad\Longleftrightarrow\ \big(\mathbf{P} \wedge\ \neg\mathbf{WP(DO,true)}\big) \Rightarrow \forall\lambda\boldsymbol{.}\big(\lambda \prec \mathbf{t}\big)$

$\Longleftrightarrow$ $\big(\mathbf{P} \wedge \neg\mathbf{WP(DO,true)}\big) \Rightarrow$ **false** since $\exists\lambda.\mathbf{t}\preccurlyeq\lambda$ (**t** gives values in Γ)
$\Longleftrightarrow$ $\neg\big(\mathbf{P} \wedge \neg\mathbf{WP(DO,true)}\big)$
$\Longleftrightarrow$ $\neg\mathbf{P} \vee \mathbf{WP(DO,true)}$
$\Longleftrightarrow$ $\mathbf{P} \Rightarrow \mathbf{WP(DO,true)}$ as required.

Note: Dijkstra's version of the theorem had $\Gamma = \omega$ with $\preccurlyeq$ as $\leqslant$ (the usual order on integers) and
(iia) $\forall\mathbf{t}_0.\big((\mathbf{P} \wedge \mathbf{BB} \wedge \mathbf{t}\leqslant\mathbf{t}_0\mathbf{+1}\big) \Rightarrow \mathbf{WP(IF,t}\leqslant\mathbf{t}_0)\big)$ and
(iiia) $\mathbf{P} \wedge \mathbf{BB} \Rightarrow \mathbf{t}\neq\mathbf{0}$
as premises. Our premise (ii) is equivalent to (iia) $\wedge$ (iiia) in this case. However, if we take Γ to be a larger ordinal than ω the theorem fails with his version of the premises:
For example: take $\mathbf{P} \Longleftrightarrow \mathbf{t}=\omega$, $\mathbf{BB} \Longleftrightarrow$ **true**, and $\mathbf{S}_i$ =**skip**. Then (i), (iia) and (iiia) hold but the loop obviously fails to terminate and so $\mathbf{P}\Rightarrow\mathbf{WP(DO,true)}$ fails.
Our premise (ii) does not hold for this example, it fails for $\mathbf{t}_0 = \omega$.

This is a useful extension of Dijkstra's theorem because it allows us to use any well-founded relation to prove termination -for instance the order relation on any ordinal number is well-founded. For example if we can show that the body of a loop decreases the list $\langle\mathbf{a}_1,\ldots,\mathbf{a}_k\rangle$ of integer variables according to their lexicographical order then we can use this tuple as our term **t** and take $\Gamma = \mathbb{N}^k$ and $\preccurlyeq$ as the lexical order on Γ. Our theorem can then be applied to prove termination. It is not possible to find an integer function of a list of integers which preserves the lexicographical order since a list of values may have an infinite number of other lists below it in the order. Hence Dijkstra's theorem cannot be used in this common instance.

Using theorem A we see that (ii) holds if
$\bigwedge_{1\leqslant j\leqslant n} \forall\mathbf{t}_0.\big((\mathbf{P} \wedge \mathbf{B}_j \wedge \mathbf{t}\preccurlyeq\mathbf{t}_0\big) \Rightarrow \mathbf{WP(S}_j\mathbf{,t}\prec\mathbf{t}_0)\big)$.
Let $\mathbf{t}_{min}$ be the least element of $\{\mathbf{t}_0 \in \Gamma | \mathbf{WP(S,t}\prec\mathbf{t}_0)\}$ (if this is non-empty). If $\mathbf{t}_{min} \preccurlyeq\mathbf{t}$ holds initially then the execution of **S** must decrease **t** because the smallest bound greater than any final value of **t** is less than or equal to the initial value of **t**. We call the predicate $\mathbf{t}_{min} \preccurlyeq\mathbf{t}$ **Wdec(S,t)**, it is false if there is no $\mathbf{t}_0$ such that $\mathbf{WP(S,t}\prec\mathbf{t}_0)$ holds, so we have:

$$\mathbf{Wdec(S,t)} =_{DF} \exists\mathbf{t}_0.\mathbf{WP(S,t}\prec\mathbf{t}_0) \\ \wedge\ \forall\mathbf{t}_{min}.\big(\ \big(\mathbf{WP(S,t}\prec\mathbf{t}_{min}) \ \wedge\ \forall\mathbf{t}_0.\big(\mathbf{WP(S,t}\prec\mathbf{t}_0)\Rightarrow\mathbf{t}_{min} \preccurlyeq\mathbf{t}_0)\big) \Rightarrow \mathbf{t}_{min} \preccurlyeq\mathbf{t}\big)$$

Theorem: $\forall\mathbf{t}_0.\big(\mathbf{t}\preccurlyeq\mathbf{t}_0 \Rightarrow \mathbf{WP(S,t}\prec\mathbf{t}_0)\big) \Longleftrightarrow \mathbf{Wdec(S,t)}$

Proof: "$\Rightarrow$":
Let $\mathbf{t}_{min}$ be arbitrary such that $\mathbf{WP(S,t}\prec\mathbf{t}_{min})$ and $\forall\mathbf{t}_0.\big(\mathbf{WP(S,t}\prec\mathbf{t}_0)\Rightarrow\mathbf{t}_{min}\preccurlyeq\mathbf{t}_0\big)$.
Assume for contradiction that $\mathbf{t}\prec\mathbf{t}_{min}$, then there exists $\lambda\in\Gamma$ such that $\mathbf{t}\preccurlyeq\lambda\prec\mathbf{t}_{min}$.
$\mathbf{t}\preccurlyeq\lambda\Rightarrow\mathbf{WP(S,t}\prec\lambda)$ since $\forall\mathbf{t}_0.\big(\mathbf{t}\preccurlyeq\mathbf{t}_0\Rightarrow\mathbf{WP(S,t}\prec\mathbf{t}_0)\big)$
$\lambda\prec\mathbf{t}_{min}\Rightarrow\neg\mathbf{WP(S,t}\prec\lambda)$ since $\forall\mathbf{t}_0.\big(\mathbf{WP(S,t}\prec\mathbf{t}_0)\Rightarrow\mathbf{t}_{min}\preccurlyeq\mathbf{t}_0\big)$. Contradiction.
So $\mathbf{t}_{min}\preccurlyeq\mathbf{t}_0$ and $\mathbf{Wdec(S,t)}$ holds since $\mathbf{t}_{min}$ was arbitrary.

"$\Longrightarrow$": Let $\mathbf{t}_0$ be arbitrary such that $\mathbf{t}\preccurlyeq\mathbf{t}_0$ and assume $\mathbf{Wdec(S,t)}$.
$\{\lambda|\mathbf{WP(S,t}\prec\lambda)\}\neq\varnothing$ since $\exists\lambda.\mathbf{WP(S,t}\prec\lambda)$ since $\mathbf{Wdec(S,t)}$ holds.
Let $\mathbf{t}_{min}$ be any minimal element of $\{\lambda|\mathbf{WP(S,t}\prec\lambda)\}$, one exists since $\preccurlyeq$ is well-founded. Then $\mathbf{WP(S,t}\prec\mathbf{t}_{min})$ and $\lambda\prec\mathbf{t}_{min}\Rightarrow\neg\mathbf{WP(S,t}\prec\lambda)$ since $\mathbf{t}_{min}$ is a minimal element.
So $\forall\lambda.\big(\mathbf{WP(S,t}\prec\lambda)\Rightarrow\mathbf{t}_{min}\preccurlyeq\lambda\big)$.
So by premise $\mathbf{t}_{min}\preccurlyeq\mathbf{t}$. $\mathbf{t}\preccurlyeq\mathbf{t}_0$ by assumption so $\mathbf{t}_{min}\preccurlyeq\mathbf{t}_0$ so
$\mathbf{WP(S,t}\prec\mathbf{t}_{min})\Rightarrow\mathbf{WP(S,t}\prec\mathbf{t}_0)$ so we have $\mathbf{WP(S,t}\prec\mathbf{t}_0)$ as required.

Putting these results together we have:

<u>Theorem:</u> If $\mathbf{P}\Rightarrow\bigwedge_{1\leqslant j\leqslant n}\big(\mathbf{B}_j\Rightarrow\mathbf{WP(S}_j\ \mathbf{,P)}\wedge\mathbf{Wdec(S}_j\ \mathbf{,t)}\big)$ then $\mathbf{P}\Rightarrow\mathbf{WP(DO,P}\wedge\neg\mathbf{BB)}$.
which can be read as: if each $\mathbf{S}_j$ preserves $\mathbf{P}$ and decreases $\mathbf{t}$ when $\mathbf{B}_j$ and $\mathbf{P}$ are true initially, then if **DO** is started in a state which satisfies **P** it will terminate in a state with **P** true and all $\mathbf{B}_j$ false.

<u>Proving Termination of Recursive Statements</u>

<u>Theorem:</u> Invariant maintenance:
(i) If for any $\mathbf{S}_1\Delta\vdash\mathbf{\{P\};S[S_1/X]}\leqslant\mathbf{S[\{P\};S_1/X]}$
then $\Delta\vdash\mathbf{\{P\};}$ **<u>proc</u>** $\mathbf{X}\equiv\mathbf{S.}\leqslant$ **<u>proc</u>** $\mathbf{X}\equiv\mathbf{\{P\};S.}$
(ii) If in addition $\Delta\vdash\mathbf{\{P\};S_1}\leqslant\mathbf{S_1;\{P\}}$ implies $\Delta\vdash\mathbf{\{P\};S[S_1/X]}\leqslant\mathbf{S[S_1/X];\{P\}}$
then $\Delta\vdash\mathbf{\{P\};}$ **<u>proc</u>** $\mathbf{X}\equiv\mathbf{S.}\leqslant$ **<u>proc</u>** $\mathbf{X}\equiv\mathbf{S.\ ;\{P\}}$

Proof:
(i) **Claim:** $\mathbf{\{P\};}$ **<u>proc</u>** $\mathbf{X}\equiv\mathbf{S.}^n\leqslant$ **<u>proc</u>** $\mathbf{X}\equiv\mathbf{\{P\};S.}^n$ for all **n**.
Then result follows from the induction rule for recursion.
Proof of Claim: For **n=0** both sides are **abort**.
So suppose the result holds for **n**
$\mathbf{\{P\};}$ **<u>proc</u>** $\mathbf{X}\equiv\mathbf{S.}^{n+1}\approx\mathbf{\{P\};S[}$ **<u>proc</u>** $\mathbf{X}\equiv\mathbf{S.}^n\mathbf{/X]}$ Put $\mathbf{S}_1=$ **<u>proc</u>** $\mathbf{X}\equiv\mathbf{S.}^n$ in the premise

$\leqslant$ **S[{P}; <u>proc</u> X $\equiv$ S.^{n}/X]**
$\leqslant$ **({P};S)[{P}; <u>proc</u> X $\equiv$ S.^{n}/X]**
$\leqslant$ **({P};S)[<u>proc</u> X $\equiv$ {P};S.^{n}/X]** by the induction hypothesis and replacement
$\leqslant$ **<u>proc</u> X $\equiv$ {P};S.$^{n+1}$**.
Hence the result by induction.

(ii) **Claim: {P}; <u>proc</u> X $\equiv$ S.^{n} $\leqslant$ <u>proc</u> X $\equiv$ S.^{n} ;{P}** for all **n**.
Then result follows from the induction rule for recursion.
Proof of Claim: For **n=0** both sides are **abort**.
So suppose the result holds for **n**
{P};<u>proc</u> X $\equiv$ S.$^{n+1}$ $\approx$ **{P};S[<u>proc</u> X $\equiv$ S.^{n}/X]**
$\leqslant$ **{P};S[{P};<u>proc</u> X $\equiv$ S.^{n}/X]** by part (i).
Put **S**$_1$ =**<u>proc</u> X $\equiv$ S.^{n}** in the premise:
{P};<u>proc</u> X $\equiv$ S.^{n} $\leqslant$ <u>proc</u> X $\equiv$ S.^{n};{P} by induction hypothesis so
{P};<u>proc</u> X $\equiv$ S.$^{n+1}$ $\leqslant$ <u>proc</u> X $\equiv$ S.$^{n+1}$;{P}
Hence the result by induction.

<u>Theorem:</u> If $\preccurlyeq$ is a well-founded partial order on some set Γ and **t** is a term giving values in Γ and **t**$_0$ is a variable which does not occur in **S** then if
(i) $\forall$**t**$_0$.$\big($(**P** $\wedge$ **t**$\preccurlyeq$**t**$_0$) $\Rightarrow$ **WP(S[{P $\wedge$ t$\prec$t$_0$}/X],true)**$\big)$
then **P** $\Rightarrow$ **WP(<u>proc</u> X $\equiv$ S.,true)**.
Proof: Putting **t**$_0$ =**t** in (i) gives
(ii) **P** $\Rightarrow$ **WP(S[{P}/X],true)**
For any minimal element λ of Γ we can put **t**$_0$ = λ in (ii) as above and get
$\big($**P** $\wedge$ **t**$\preccurlyeq \lambda\big)$ $\Rightarrow$ **WP(S[{t$\prec$ λ}/X],true)**$\big)$ $\Rightarrow$
WP(S[{false}/X],true) $\Rightarrow$ **WP(S[abort/X],true)**
t$\preccurlyeq \lambda \Rightarrow$ **t**$= \lambda$ and the above holds for any minimal λ so we get
(iii) **P** $\wedge$ **minimal(t)** $\Rightarrow$ **WP(S[abort/X],true)**.
WP(<u>proc</u> X $\equiv$ S. ,true) $\iff \bigvee_{n<\omega}$**WP(<u>proc</u> X $\equiv$ S.^{n}, true)**
$\iff \bigvee_{n<\omega}$**WP(S[<u>proc</u> X $\equiv$ S.$^{n-1}$/X], true)**

Claim: For any $\lambda \in \Gamma$, **P** $\wedge$ **t**$\preccurlyeq \lambda$ $\Rightarrow$ **WP(<u>proc</u> X $\equiv$ S., true)**.
Proof of Claim: Assume **P** $\wedge$ **t**$\preccurlyeq \lambda$.
(a) If λ is a minimal element of Γ then (iii) gives
WP(S[abort/X],true) (under our assumption).

ie $\mathbf{WP(S[abort/X],true)} \iff \mathbf{WP(\underline{proc}\ X \equiv S.^1,true)} \Rightarrow \bigvee_{n<\omega}\mathbf{WP(\underline{proc}\ X \equiv S.^n,\ true)}$
$\Rightarrow \mathbf{WP(\underline{proc}\ X \equiv S.,\ true)}$

(b) Suppose λ is not a minimal element of Γ and suppose the result holds for all $\delta \prec \lambda$. Then putting $\mathbf{t}_0 = \lambda$ in (i) gives

$$\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{WP(S[\{t} \prec \lambda\mathbf{\}/X],true)}$$

$\mathbf{t} \prec \lambda \Rightarrow \exists\delta.\ \mathbf{t} \preccurlyeq \delta \prec \lambda \Rightarrow \exists\delta \prec \lambda.\ \mathbf{t} \preccurlyeq \delta$

By the induction hypothesis we have $\mathbf{P} \wedge \delta \prec \lambda \wedge \mathbf{t} \preccurlyeq \delta \Rightarrow \mathbf{WP(\underline{proc}\ X \equiv S.,true)}$.

So $\mathbf{WP(S[\{t} \prec \lambda\mathbf{\}/X],true)} \Rightarrow \mathbf{WP(S[\{P} \wedge \mathbf{t} \prec \lambda\mathbf{\}/X],true)}$ by (i)
$\Rightarrow \mathbf{WP(S[\{P} \wedge \exists\delta.\delta \prec \lambda \wedge \mathbf{t} \preccurlyeq \delta\mathbf{\}/X],true)}$
$\Rightarrow \mathbf{WP(S[\{}\exists\delta.\delta \prec \lambda \wedge \mathbf{WP(\underline{proc}\ X \equiv S\ ,true)\}/X],true)}$

by hypothesis and the Replacement Theorem.

λ not minimal implies $\exists\delta.\delta \prec \lambda$ and as δ and λ do not occur in **S** this is invariantly true so can be removed from any assertion. We get
$\Rightarrow \mathbf{WP(S[\{WP(\underline{proc}\ X \equiv S.,true)\}/X],true)}$
$\Rightarrow \mathbf{WP(S,true)[WP(\ \{WP(\underline{proc}\ X \equiv S.,true)\}\ ,true)/WP(X,true)]}$

Now $\mathbf{WP(\{Q\},R)} \iff \mathbf{Q} \wedge \mathbf{R}$ by definition so this is
$\Rightarrow \mathbf{WP(S,true)[WP(\underline{proc}\ X \equiv S.,true)\}/WP(X,true)]}$
$\Rightarrow \mathbf{WP(S[\underline{proc}\ X \equiv S./X],true)}$
$\Rightarrow \mathbf{WP(\underline{proc}\ X \equiv S.,true)}$ by folding.

Consider the set $\Gamma' = \{\lambda \in \Gamma | \neg(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{WP(\underline{proc}\ X \equiv S.,true)})\}$. Suppose this is non-empty, then since $\preccurlyeq$ is well-founded there exists a minimal element $\zeta \in \Gamma'$. ζ is not a minimal element of Γ by **(a)** above. If $\delta \prec \zeta$ then $(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \delta \Rightarrow \mathbf{WP(\underline{proc}\ X \equiv S.,true)})$ holds. But then by **(b)** above we have $(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \zeta \Rightarrow \mathbf{WP(\underline{proc}\ X \equiv S.,true)})$ holds which is a contradiction.
So Γ' must be empty and so $(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{WP(\underline{proc}\ X \equiv S.,true)})$ holds for all $\lambda \in \Gamma$ which proves the claim.

Finally: $\forall\lambda \in \Gamma.(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{WP(\underline{proc}\ X \equiv S.,true)})$
$\iff \mathbf{P} \Rightarrow \mathbf{WP(\underline{proc}\ X \equiv S.,true)}$ as was proved above for **DO**.

Let $\mathbf{t}_{min}$ be the least element of $\{\mathbf{t}_0 \in \Gamma | \mathbf{WP(S[\{t} \prec \mathbf{t}_0\mathbf{\}/X],true)}\}$ (if this is non-empty).
If $\mathbf{t}_{min}$ exists and $\mathbf{t}_{min} \preccurlyeq \mathbf{t}$ initially then any calls of **X** in the execution of **S** must be started in a state in which **t** is smaller than it was initially because the smallest bound greater than any possible value of **t** at a call of **X** is less than or equal to the initial value of **t**.

We call the predicate $\mathbf{t}_{min} \preccurlyeq \mathbf{t}$ $\mathbf{Wdec}_X\mathbf{(S,t)}$, it is false if there is no $\mathbf{t}_0$ such that $\mathbf{WP(S[\{t \prec t_0\}/X],true)}$ holds, so we have:
$\mathbf{Wdec}_X\mathbf{(S,t)} =_{DF} \exists \mathbf{t}_0.\ \mathbf{WP(S[\{t \prec t_0\}/X],true)}$
$\quad \wedge\ \forall \mathbf{t}_{min}.\big(\ \big(\mathbf{WP(S[\{t \prec t}_{min}\mathbf{\}/X],true)}$
$\qquad \wedge\ \forall \mathbf{t}_0.\big(\mathbf{WP(S[\{t \prec t_0\}/X],true)} \Rightarrow \mathbf{t}_{min} \preccurlyeq \mathbf{t}_0\big)\big) \Rightarrow \mathbf{t}_{min} \preccurlyeq \mathbf{t}\big)$

<u>**Theorem:**</u> $\forall \mathbf{t}_0.\big(\mathbf{t} \preccurlyeq \mathbf{t}_0 \Rightarrow \mathbf{WP(S[\{t \prec t_0\}/X],true)}\big) \iff \mathbf{Wdec}_X\mathbf{(S,t)}$

Proof: "$\Rightarrow$": Let $\mathbf{t}_{min}$ be arbitrary such that $\mathbf{WP\ (S[\{t \prec t}_{min}\mathbf{\}/X],true)}$ and $\forall \mathbf{t}_0.\big(\mathbf{WP(S[\{t \prec t_0\}/X],true)} \Rightarrow \mathbf{t}_{min} \preccurlyeq \mathbf{t}_0\big)$.
Assume for contradiction that $\mathbf{t} \prec \mathbf{t}_{min}$.
Then there exists $\lambda \in \Gamma$ such that $\mathbf{t} \preccurlyeq \lambda \prec \mathbf{t}_{min}$.
$\mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{WP(S[\{t \prec}\ \lambda\mathbf{\}/X],true)}$ since $\forall \mathbf{t}_0.\big(\mathbf{t} \preccurlyeq \mathbf{t}_0 \Rightarrow \mathbf{WP(S[\{t \prec t_0\}/X],true)}\big)$
$\lambda \prec \mathbf{t}_{min} \Rightarrow \neg \mathbf{WP(S[\{t \prec}\ \lambda\mathbf{\}/X],true)}$ since $\forall \mathbf{t}_0.\big(\mathbf{WP(S[\{t \prec t_0\}/X],true)} \Rightarrow \mathbf{t}_{min} \preccurlyeq \mathbf{t}_0\big)$.
Contradiction.
So $\mathbf{t}_{min} \preccurlyeq \mathbf{t}_0$ and $\mathbf{Wdec}_X\mathbf{(S,t)}$ holds since $\mathbf{t}_{min}$ was arbitrary.

"$\Longrightarrow$": Let $\mathbf{t}_0$ be arbitrary such that $\mathbf{t} \preccurlyeq \mathbf{t}_0$ and assume $\mathbf{Wdec}_X\mathbf{(S,t)}$.
$\{\lambda | \mathbf{WP(S[\{t \prec}\ \lambda\mathbf{\}/X],true)}\} \neq \varnothing$ since $\exists \lambda. \mathbf{WP(S[\{t \prec}\ \lambda\mathbf{\}/X],true)}$ since $\mathbf{Wdec}_X\mathbf{(S,t)}$ holds.
Let $\mathbf{t}_{min}$ be any minimal element of $\{\lambda | \mathbf{WP(S[\{t \prec}\ \lambda\mathbf{\}/X],true)}\}$, one exists since $\preccurlyeq$ is well-founded.
Then $\mathbf{WP(S[\{t \prec t}_{min}\mathbf{\}/X],true)}$ and $\lambda \prec \mathbf{t}_{min} \Rightarrow \neg \mathbf{WP(S[\{t \prec}\ \lambda\mathbf{\}/X],true)}$ since $\mathbf{t}_{min}$ is a minimal element.
So $\forall \lambda.\big(\mathbf{WP(S[\{t \prec}\ \lambda\mathbf{\}/X],true)} \Rightarrow \mathbf{t}_{min} \preccurlyeq \lambda\big)$.

So by premise $\mathbf{t}_{min} \preccurlyeq \mathbf{t}$. $\mathbf{t} \preccurlyeq \mathbf{t}_0$ by assumption so $\mathbf{t}_{min} \preccurlyeq \mathbf{t}_0$ so $\mathbf{WP(S[\{t \prec t}_{min}\mathbf{\}/X],true)} \Rightarrow \mathbf{WP(S[\{t \prec t_0\}/X],true)}$ which was required.

Putting these results together we have:

<u>**Theorem:**</u> If
(i) $\mathbf{P} \Rightarrow \mathbf{WP(S[\{P\}/X],true)}$ and
(ii) For any $\mathbf{S}_1$: $\mathbf{\{P\};S}_1 \leqslant \mathbf{S}_1\mathbf{;\{P\}} \Rightarrow \mathbf{\{P\};S[S}_1\mathbf{/X]} \leqslant \mathbf{S[S}_1\mathbf{/X];\{P\}}$
(iii) For any $\mathbf{S}_1$: $\mathbf{\{P\};S[S}_1\mathbf{/X]} \leqslant \mathbf{S[\{P\};S}_1\mathbf{/X]}$
(iv) $\mathbf{P} \Rightarrow \mathbf{Wdec}_X\mathbf{(S,t)}$

Then $\mathbf{P} \Rightarrow \mathbf{WP}(\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S.}\ ,\mathbf{P})$.

The second premise states that if all recursive calls of **S** in $\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S.}$ were replaced by any statement which preserved **P** then the statement so formed would preserve **P**.

<u>PROOF</u> <u>RULES</u> <u>FOR</u> <u>IMPLEMENTATION</u>

Having introduced the programming language and proof method we will illustrate it by some examples of simple program refinements and transformations. In the next Chapter we develop some basic transformation rules and the conditions under which they will work. First however we develop a general proof rule by which the correctness of an implementation of a specification such as $\mathbf{\{P\};x{:}=x'.Q}$ may be shown and a proof rule for proving that a given recursive procedure statement is a correct implementation of a given statement. This latter rule is very important in the process of transforming a specification expressed using recursion into a recursive procedure which implements it. The recursive procedure may then be further transformed into an iterative program, if required, using the techniques presented later in this Thesis.

<u>Theorem:</u> Let Δ be a countable set of sentences of **L**. Let **V** be a finite nonempty set of program variables and **S** a program description or abstraction in **V**. Let **y** be a list of all the variables in $\mathbf{V}-\underline{\mathbf{x}}$ not constant in **S**. Let $\mathbf{x}_0$, $\mathbf{y}_0$ be lists of distinct program variables not in **S** or **V** with $\ell(\mathbf{x}_0)= \ell(\mathbf{x})$ and $\ell(\mathbf{y}_0)= \ell(\mathbf{y})$.
If $\Delta \vdash \mathbf{P} \wedge \mathbf{x}{=}\mathbf{x}_0\ \wedge \mathbf{y}{=}\mathbf{y}_0 \Rightarrow \mathbf{WP}(\mathbf{S},\mathbf{Q}[\mathbf{x}_0/\mathbf{x},\mathbf{x}/\mathbf{x}'] \wedge \mathbf{y}{=}\mathbf{y}_0)$
then $\Delta \vdash \mathbf{\{P\};x{:}=x'.Q} \leqslant \mathbf{S}$

The premise states that if $\mathbf{x}_0$ and $\mathbf{y}_0$ contain the initial values of **x** and **y** then **S** preserves the value of **y** and sets **x** to a value $\mathbf{x}'$ such that the relationship between the initial value of **x** and $\mathbf{x}'$ satisfies **Q**. This was proved in [Back 80] for iterative programs, the extension to include recursive programs is straightforward.

<u>Cor</u> <u>A:</u> By the same assumptions as above we have:
$\kappa\Delta \vdash \big(\exists \mathbf{x}'.\mathbf{Q}\big)\ \wedge\ \big(\mathbf{x}{=}\mathbf{x}_0\big)\ \wedge\ \big(\mathbf{y}{=}\mathbf{y}_0\big) \Rightarrow \mathbf{WP}(\mathbf{S},\mathbf{Q}[\mathbf{x}_0/\mathbf{x},\mathbf{x}/\mathbf{x}'] \wedge \mathbf{y}{=}\mathbf{y}_0)$ implies $\vdash \mathbf{x{:}=x'.Q} \leqslant \mathbf{S}$

<u>Cor</u> <u>B:</u> For the assignment $\mathbf{x{:}=t}$ in **V** we have:
$\kappa\Delta \vdash \mathbf{P} \wedge \mathbf{x}{=}\mathbf{x}_0\ \wedge \mathbf{y}{=}\mathbf{y}_0 \Rightarrow \mathbf{WP}(\mathbf{S},\mathbf{x}{=}\mathbf{t}[\mathbf{x}_0/\mathbf{x}] \wedge \mathbf{y}{=}\mathbf{y}_0)$ implies $\vdash \mathbf{\{P\};x{:}=t} \leqslant \mathbf{S}$

This theorem is useful for implementing simple specifications. More complex specifications will be implemented as recursive or iterative procedures: in either case we can use the following theorem to develop a recursive implementation as the first stage. This can be transformed into an iterative program (if required) using the techniques on recursion removal which we will develop in later chapters.

<u>Recursive</u> <u>Implementation</u> <u>of</u> <u>Specifications</u>

Suppose we have a specification $\mathbf{\{P\};x{=}x'.Q}$ we wish to implement as a recursive procedure. If we have a statement **S** which contains the statement variable **X** (representing recursive calls to **S**) and can prove $\mathbf{S'} \leqslant \mathbf{S[S'/X]}$ ie if given that the recursive calls of **S** "work" then so does **S**, and in addition we can find some term which is decreased for the inner calls then we can deduce $\mathbf{S'} \leqslant$ **<u>proc</u>** $\mathbf{X} \equiv \mathbf{S.}$. Thus we can gradually transform the specification $\mathbf{S'}$, by splitting it into cases etc. until we get a statement **S** which is defined in terms of $\mathbf{S'}$ but for which we can find a term which is reduced before each occurrence of $\mathbf{S'}$. Then using the next theorem we can immediately deduce $\mathbf{S'} \leqslant$ **<u>proc</u>** $\mathbf{X} \equiv \mathbf{S.}$ which no longer contains $\mathbf{S'}$. This is the motivation for the next theorem which we will prove for any type of statement $\mathbf{S'}$ (ie we will not restrict ourselves to specifications).

<u>Theorem:</u> If $\preccurlyeq$ is a well-founded partial order on some set Γ and **t** is a term giving values in Γ and $\mathbf{t}_0$ is a variable which does not occur in **S** then if
(i) $\forall \mathbf{t}_0.\big((\mathbf{P} \ \wedge \ \mathbf{t} \preccurlyeq \mathbf{t}_0) \Rightarrow \mathbf{S'} \leqslant \mathbf{S[\{P} \ \wedge \ \mathbf{t} \prec \mathbf{t}_0\mathbf{\}; \ S'/X]}\big)$

then $\mathbf{P} \Rightarrow \mathbf{S'} \leqslant$**<u>proc</u>** $\mathbf{X} \equiv \mathbf{S.}$.

Proof: From (i) we get $\forall \mathbf{t}_0.\big((\mathbf{P} \ \wedge \ \mathbf{t} \preccurlyeq \mathbf{t}_0) \Rightarrow \mathbf{S'} \leqslant \mathbf{S[\{P\};S'/X]}\big)$ by removing an assertion and putting $\mathbf{t}{=}\mathbf{t}_0$ in this gives:
(ii) $\mathbf{P} \Rightarrow \mathbf{S'} \leqslant \mathbf{S[\{P\};S'/X]}$

If λ is any minimal element of Γ then putting $\mathbf{t}_0 = \lambda$ in (i) gives
$\big(\mathbf{P} \ \wedge \ \mathbf{t} \preccurlyeq \lambda\big) \Rightarrow \mathbf{S'} \leqslant \mathbf{S[\{P} \ \wedge \ \mathbf{t} \prec \lambda\mathbf{\};S'/X]} \Rightarrow \mathbf{S'} \leqslant \mathbf{S[abort;S'/X]} \Rightarrow \mathbf{S'} \leqslant \mathbf{S[abort/X]}$
since $\mathbf{t} \prec \lambda \iff$ **false**. So as before we get
(iii) $\mathbf{P} \wedge \mathbf{minimal(t)} \Rightarrow \mathbf{S'} \leqslant \mathbf{S[abort/X]}$

Claim: For any $\lambda \in \Gamma$, $\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{S'} \leqslant$**<u>proc</u>** $\mathbf{X} \equiv \mathbf{S.}$.

Proof of Claim: Assume $\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda$.
(a) If λ is a minimal element of Γ then (iii) gives
$\mathbf{S'} \leqslant \mathbf{S[abort/X]} \ \approx$ **<u>proc</u>** $\mathbf{X} \equiv \mathbf{S.}^1 \leqslant \bigvee_{n<\omega}$**<u>proc</u>** $\mathbf{X} \equiv \mathbf{S.}^n \ \approx$ **<u>proc</u>** $\mathbf{X} \equiv \mathbf{S.}$ as required.

(b) Suppose λ is not a minimal element of Γ and suppose the result holds for all $\delta \prec \lambda$.
Then putting $\mathbf{t}_0 = \lambda$ in (i) gives

$$\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \Rightarrow \mathbf{S}' \leqslant \mathbf{S}[\{\mathbf{P} \ \wedge \ \mathbf{t} \prec \lambda\};\mathbf{S}'/\mathbf{X}])$$

$\mathbf{t} \prec \lambda \Rightarrow \exists\delta.\ \mathbf{t} \preccurlyeq \delta \prec \lambda \Rightarrow \exists\delta \prec \lambda.\ \mathbf{t} \preccurlyeq \delta$

By the induction hypothesis we have $\mathbf{P} \ \wedge \ \delta \prec \lambda \ \wedge \ \mathbf{t} \preccurlyeq \delta \Rightarrow \mathbf{S}' \leqslant \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.$
ie $\{\mathbf{P} \ \wedge \ \mathbf{t} \prec \lambda\};\mathbf{S}' \leqslant \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.$
So $\mathbf{S}' \leqslant \mathbf{S}[\{\mathbf{P} \ \wedge \ \exists\delta \prec \lambda.\ \mathbf{t} \preccurlyeq \delta\};\mathbf{S}'/\mathbf{X}])$
$\quad \leqslant \mathbf{S}[\{\exists\delta \prec \lambda.\ \mathbf{P} \ \wedge \ \mathbf{t} \preccurlyeq \delta\};\mathbf{S}'/\mathbf{X}])$
$\quad \leqslant \mathbf{S}[\{\exists\delta \prec \lambda\};\ \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}./\mathbf{X}])$ by hypothesis and Replacement.

λ not minimal implies $\exists\delta.\delta \prec \lambda$ and as δ and λ do not occur in **S** this is invariantly true so can be removed from any assertion. We get
$\mathbf{S}' \leqslant \mathbf{S}[\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}./\mathbf{X}]) \ \approx \ \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.$ by folding.

Consider the set $\Gamma' = \{\lambda \in \Gamma | \neg(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \ \Rightarrow \ \mathbf{S}' \leqslant \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.)\}$. Suppose this is non-empty, then since $\preccurlyeq$ is well-founded there exists a minimal element $\zeta \in \Gamma'$. ζ is not a minimal element of Γ by **(a)** above. If $\delta \prec \zeta$ then $(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \delta \ \Rightarrow \ \mathbf{S}' \leqslant \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.)$ holds. But then by **(b)** above we have $(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \zeta \Rightarrow \ \mathbf{S}' \leqslant \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.)$ holds which is a contradiction.
So Γ' must be empty and so $(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \ \Rightarrow \ \mathbf{S}' \leqslant \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.)$ holds for all $\lambda \in \Gamma$ which proves the claim.

Finally $\forall\lambda \in \Gamma.(\mathbf{P} \wedge \mathbf{t} \preccurlyeq \lambda \ \Rightarrow \ \mathbf{S}' \leqslant \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.)$
$\quad \Longleftrightarrow \ (\mathbf{P} \ \Rightarrow \ \mathbf{S}' \leqslant \underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.)$ as was proved above for **DO**.

At first sight it might appear that the theorem on proving termination of a recursive statement is a special case of this theorem with $\mathbf{S}'$ =**skip** but this is not the case because the premise of that theorem is weaker (only requiring $\mathbf{WP(S[\{P \wedge t \prec t_0\}/X],true)}$ rather than $\mathbf{skip \leqslant S[\{P \wedge t \prec t_0\}/X]}$ which is equivalent to $\mathbf{R \Rightarrow WP(S[\{P \wedge t \prec t_0\}/X],R)}$ -in the first case we don't care what else $\mathbf{S[\{P \wedge t \prec t_0\}/X]}$ does so long as it terminates) and the conclusion is also weaker (the first heorem merely proves that $\underline{\mathbf{proc}}\ \mathbf{X} \equiv \mathbf{S}.$ terminates, the second also proves that it preserves the state). The first theorem is solely concerned with termination (and hence is useful if partial correctness has been established independently) the second with the implementation of a specification (and hence proving termination wherever the specification terminates).

To summarise:

(i) If $\mathbf{P} \wedge \mathbf{x{=}x_0} \wedge \mathbf{y{=}y_0} \Rightarrow \mathbf{WP(S,Q[x_0/x,x/x'] \wedge y{=}y_0)}$
   then $\mathbf{\{P\};\ x{:=}x'.Q \leqslant S}$.

where **y** are the variables other than those in **x** which are not constant in **S**.

(ii) If $\forall \mathbf{t_0}\big((\mathbf{P} \wedge \mathbf{t{\preccurlyeq}t_0}) \Rightarrow \mathbf{S' \leqslant S[\{P \wedge t{\prec}t_0\}/X]}\big)$
   then $\mathbf{\{P\};\ S' \leqslant}$ **proc** $\mathbf{X \equiv S.}$.

Putting these together gives:

Cor: Recursive Implementation of a Specification:

If $\big(\mathbf{P} \wedge \mathbf{t{\preccurlyeq}t_0} \wedge \mathbf{x{=}x_0} \wedge \mathbf{y{=}y_0}\big) \Rightarrow \mathbf{WP(S[[\{P \wedge t{\prec}t_0\};\ x{:=}x'.Q\ /\ X],\ Q[x_0/x,x/x'] \wedge y{=}y_0)}$
then $\mathbf{\{P\};\ x{:=}x'.Q \leqslant}$ **proc** $\mathbf{X \equiv S.}$.

This theorem is a fundamental result towards our aim of a system for transforming specifications into programs since it "bridges the gap" between a recursively defined specification and a recursive procedure which implements it. It is of use even when the final program is iterative rather than recursive since many iterative algorithms may be more easily and clearly specified as recursive functions–even if they may be more efficiently implemented as iterative procedures. This theorem may be used by the programmer to transform the recursively defined specification into a recursive procedure or function which can then be transformed into an iterative procedure. In subsequent chapters we will prove many transformations which will make the transition from recursion to iteration very straightforward.

Example:

Suppose we have the specification: **x:=n!** ie $\mathbf{x{:=}x'.\big(x'{=}n!\big)}$ with the precondition $\mathbf{P} = \mathbf{n{\geqslant}0} \wedge \mathbf{n} {\in} \mathbb{N}$ (where **n!** is the usual factorial function). We claim that the recursive statement **proc X ≡ S.** implements this specification where:

```
S= if n=0 then x:=1
   else n:=n−1; X; n:=n+1; x:=x.n fi.
```

For the term **t** we simply take **n** itself. The only variable other than **x** which is not constant in **S** is **n**. We need to prove:

```
(n⩾0 ∧ n⩽t₀ ∧ x=x₀ ∧ n=n₀) ⇒
   WP(if n=0 then x:=1
      else n:=n−1; {n⩾0 ∧ n<t₀}; x:=n! ;
        n:=n+1; x:=x.n fi , (x=n! ∧ n=n₀))
```

Assume $\mathbf{n \geqslant 0 \ \wedge \ n \leqslant t_0 \ \wedge \ x = x_0 \ \wedge \ n = n_0}$.
Then $\mathbf{WP(\underline{if}\ldots\ \underline{fi},\ (x{=}n! \ \wedge \ n{=}n_0))}$
$\iff \mathbf{(n{=}0 \Rightarrow WP(x{:=}1,\ (x{=}n! \ \wedge \ n{=}n_0))}$
$\quad \wedge \ \mathbf{(n{\neq}0 \Rightarrow WP(n{:=}n{-}1;\ \{n{\geqslant}0 \ \wedge \ n{<}t_0\};\ x{:=}n!\ ;\ n{:=}n{+}1;\ x{:=}x.n,\ (x{=}n! \ \wedge \ n{=}n_0))\ )}$

Assume $\mathbf{n{=}0}$. Then:
$\mathbf{WP(x{:=}1,\ (x{=}n! \ \wedge \ n{=}n_0))} \iff \mathbf{1{=}n! \ \wedge \ n{=}n_0}$ which follows from the assumptions.

On the other hand, assume $\mathbf{n{\neq}0}$. Then:
$\mathbf{WP(n{:=}n{-}1;\ \{n{\geqslant}0 \ \wedge \ n{<}t_0\};\ x{:=}n!\ ;\ n{:=}n{+}1;\ x{:=}x.n,\ (x{=}n! \ \wedge \ n{=}n_0))}$
$\quad \iff \mathbf{WP(n{:=}n{-}1;\ \{n{\geqslant}0 \ \wedge \ n{<}t_0\},\ (x{=}n! \ \wedge \ n{=}n_0)[x.n/x][n{+}1/n][n!/x])}$
(from the theorem giving the **WP** of an assignment statement.)
$\quad \iff \mathbf{WP(n{:=}n{-}1,\ (n{\geqslant}0 \ \wedge \ n{<}t_0) \ \wedge \ (n!.(n{+}1) = (n{+}1)! \ \wedge \ n{+}1{=}n_0))}$
(from the theorem giving the **WP** of an assertion.)
$\quad \iff \mathbf{(n{-}1{\geqslant}0 \ \wedge \ n{-}1{<}t_0) \ \wedge \ ((n{-}1)!.((n{-}1){+}1) = ((n{-}1){+}1)! \ \wedge \ (n{-}1){+}1{=}n_0)}$
$\quad \iff \mathbf{n{-}1{\geqslant}0 \ \wedge \ n{-}1{<}t_0 \ \wedge \ (n{-}1)!.n{=}n! \ \wedge \ n{=}n_0}$
These all follow from the assumptions $\mathbf{n{>}0}$ and $\mathbf{n{\leqslant}t_0}$ and the definition of the factorial function.

Hence (by the corollary above):
x:=n! ⩽ <u>proc</u> X ≡ <u>if</u> n=0 <u>then</u> x:=1
<u>else</u> n:=n−1; X; n:=n+1; x:=x.n <u>fi</u>.

Although this has enabled us to rigorously prove that this recursive procedure implements the factorial function the theorems we have proved so far do not give us any help in <u>deriving</u> such a recursive procedure given the specification –here we simply “pulled it out of a hat” and proved that it worked (following a long tradition of examples of program verification!). Much of the rest of this Thesis is devoted to devising methods and results, rigorously supported by their foundation in infinitary logic, which will assist in transforming specifications into programs. For example, we will be able to <u>derive</u> this version of the factorial function from the specification and then transform it to an iterative form using a **<u>for</u>** loop.