ASSOCIATIONS AND THE CLOSURE STATEMENT

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE TECHNISCHE WETENSCHAPPEN AAN DE TECHNISCHE HOGESCHOOL EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF. DR. P. VAN DER LEEDEN, VOOR EEN COMMISSIE AANGEWEEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEIGEN OP DINSDAG 12 OKTOBER 1976 TE 16.00 UUR.

door

MARTINUS REM
GEBOREN TE KOOG AAN DE ZAAN
Dit proefschrift is goedgekeurd
door de promotoren
Prof. dr. E.W. Dijkstra
en
Prof. dr. F.E.J. Kuiper en Arends
CHAPTER 1

PROLOGUE

Programming languages enable us to abstract from the machines we are using. It is the purpose of the implementation to map the programming language constructs on the machine instructions. This implementation should be a truthful one, i.e. it should not hide properties, the knowledge of which is indispensable for the construction of correct and efficient programs. A programming language should be such that it allows for a truthful implementation.

Present-day programming languages reflect present-day machine technology. New techniques — associative addressing, large scale integration (LSI) — are being developed. These new techniques may very well allow for a truthful implementation of radically different programming languages.

In stores that are realized with LSI-techniques the information is (usually) kept in essentially active components. Such a store virtually consists of a large number of little machines, that do nothing else but remembering some value and on command reproducing it or replacing it by some other value. It may well be that someday these little machines will be able to do "more intelligent" work than the mere simulation of a core store. It may well be that the major part of the logical manipulations will take place distributed all through the "store", thus realizing a very high degree of concurrency.

Questions that then arise are: "How can we, when it is desired to program for such a machine, exploit this potential ultraconcurrency?", "Can we think of useful language constructs whose execution may involve such a distributed activity?" and "Can we do this in such a way that the implied programming task remains intellectually manageable?". It is to such questions that this monograph is addressed.

We would like to stress that it is not our intention to design a machine. Our primary concern is the manageability from the programmer's point of view. If so desired, one can, of course, interpret the semantics of our programming language as the functional specifications of such a machine.
We shall write down programs under control of which a highly concurrent activity is possible, but not obligatory. We shall, furthermore, arrange our programs in such a fashion that, in spite of the high potential concurrency, most of the thus far developed techniques for the programming of sequential processes remain applicable. (We maintain the semicolons, but allow more powerful statements in between.)

The basic idea of the research (essentially: the massaging of a set of n-tuples) is due to E.W. Dijkstra. A preliminary design, in the realization of which W.H.J. Feijen and the author participated as well, has been reported in [7] and [8].

```
```

Most of the syntax of our programming language will be given in NNF [13]. We have extended BNF with the convention that the braces "{\ldots\}" should be read as "zero or more instances of the enclosed". E.g., the production rule

```
<statement list> ::= <statement> ( ; ; <statement> )
```

defines <statement list> to denote a sequence of one or more instances of <statement>, separated by semicolons.

There is little sense in introducing a programming language, if its semantics cannot be formalized. There are several reasons for this. A formal definition of the semantics is indispensable if one wants to prove the correctness of a program. But even if one does not intend to prove the correctness of one's programs explicitly, formally defined semantics make it possible to understand and appreciate programs without being forced to think in terms of specific implementations. We, furthermore, wish to design our programs in such a fashion that they are a priori known to meet the requirements of their correctness proofs. By doing so correctness concerns can, by their guiding role in program design, contribute to the alleviation of the programming task.

For the definition of the semantics of our programming language we shall use the concept of the weakest pre-condition [6]:
If \( S \) denotes a mechanism (statement list), and \( R \) some condition on the state of the system, then "wp\((S,R)\)" denotes the weakest pre-condition for the initial state of the system, such that activation of \( S \) is guaranteed to lead to a properly terminating activity, leaving the system in a final state satisfying the post-condition \( R \).

(A condition on the state -- or simply a condition -- is a boolean function defined on all states.) As in [6], we shall restrict ourselves to "wp's" that satisfy the following three properties for any statement list \( S \) and for all states.

**PROPERTY 1.1.** \( \text{wp}(S,\text{false}) = \text{false} \).

**PROPERTY 1.2.** For any two conditions \( P \) and \( Q \):

\[
(\text{wp}(S,P) \land \text{wp}(S,Q)) \equiv \text{wp}(S,P \land Q),
\]

**PROPERTY 1.3.** For any infinite sequence of conditions \( B_0, B_1, B_2, \ldots \), such that \( B_i \Rightarrow B_{i+1} \) (\( i \geq 0 \)),

\[
\text{wp}(S, (\exists i: i \geq 0: B_i)) \Rightarrow (\exists i: i \geq 0: \text{wp}(S,B_i)).
\]

From the above we can derive the following two properties. For any two conditions \( P \) and \( Q \):

**PROPERTY 1.4.** \( P \Rightarrow Q \) implies \( \text{wp}(S,P) \Rightarrow \text{wp}(S,Q) \).

**PROPERTY 1.5.** \( (\text{wp}(S,P) \lor \text{wp}(S,Q)) \Rightarrow \text{wp}(S,P \lor Q) \).

Property 1.4 is proved with Property 1.3. Suppose \( P \Rightarrow Q \). Define \( B_i \) (\( i \geq 0 \)) by \( B_0 \equiv P \), and \( B_{i+1} \equiv Q \). Then

\[
\text{wp}(S, (\exists i: i \geq 0: B_i)) \Rightarrow (\exists i: i \geq 0: \text{wp}(S,B_i)),
\]

or

\[
\text{wp}(S,Q) \Rightarrow (\text{wp}(S,P) \lor \text{wp}(S,Q)),
\]

from which \( \text{wp}(S,P) \Rightarrow \text{wp}(S,Q) \) follows.

Property 1.5 is proved using Property 1.4:

\[
P \Rightarrow (P \lor Q) \text{ implies } \text{wp}(S,P) \Rightarrow \text{wp}(S,P \lor Q) \quad (1)
\]

\[
Q \Rightarrow (P \lor Q) \text{ implies } \text{wp}(S,Q) \Rightarrow \text{wp}(S,P \lor Q) \quad (2)
\]
From (1) and (2) it follows that \( \wp(S, P) \lor \wp(S, Q) \Rightarrow \wp(S, P \lor Q) \).

A mechanism \( S \) is said to be \textit{deterministic} if and only if for any two conditions \( P \) and \( Q \) and for all states
\[
\wp(S, P \lor Q) \Rightarrow (\wp(S, P) \lor \wp(S, Q))
\]

We take the position that we know the semantics of a mechanism \( S \) sufficiently well if we know its predicate transformer, i.e. if we know how to derive \( \wp(S, R) \) for any post-condition \( R \).

**EXAMPLE 1.1.** The assignment statement is the basic statement of most programming languages. Its semantics are given by
\[
\wp("x := R", R) \equiv R^x
\]
in which \( R^x \) denotes a copy of the predicate defining the post-condition \( R \) in which each occurrence of the variable "\( x \)" is replaced by the expression "(E)". This definition is known as the \textit{Axiom of Assignment}. (End of example.)

**EXAMPLE 1.2.** With "skip" denoting the empty statement, and with "abort" denoting the statement that cannot terminate properly, we have, for all conditions \( P \),
\[
\wp("skip", P) = P,
\wp("abort", P) = \text{false}
\]
(End of example.)

In accordance with our earlier remark that we maintain the semicolons in our programming languages, we define the semantics on the \textit{semicolon} as in [6]:

If \( S_1 \) and \( S_2 \) denote arbitrary statement lists, and \( R \) some post-condition, then
\[
\wp("S_1;S_2", R) \equiv \wp(S_1, \wp(S_2, R))
\]
CHAPTER 2
ASSOCIATION

This chapter is devoted to the recording of states of computations in our "active" store. It is our intention to realize this recording in such a way that at each moment all of the storage contents (so to speak) are involved in the computational process.

To achieve this high degree of concurrency by asking the programmer to synchronize explicitly the co-operation between a huge number of possibly all different concurrent sequential processes, seems a blind alley in the sense that the implied programming task will quickly exceed our abilities. It seems more attractive to look for a simple and systematic instruction repertoire, such that each instruction can interfere in a homogeneous fashion with the total contents of the store.

In conventional stores the components record values of variables. The represented state is changed by altering the value recorded in one --explicitly addressed-- component. In our active store we shall not address the components to be activated in the execution of a state change explicitly. We wish to achieve the above stated homogeneous interference by broadcasting commands through the store, telling that all components of which the contents satisfy a certain condition, should do something. The storage cells are anonymous, they are characterized by their contents only, i.e. we are assuming an associative store.

Having abolished addresses, we have to introduce names. The storage cells record relations between these names. We assume the machine to be able to test the equality of two names. If all entities to be referred to are identified by mutually distinct names, then any relation between some of these entities can be represented by a relation between their names. If, e.g., \( x \) and \( y \) stand for names of persons, we could have relations like

\[
\text{fatherof}(x,y) \quad \text{meaning} \quad "x \text{ is the father of } y" , \quad \text{and} \\
\text{olderthan}(x,y) \quad \text{meaning} \quad "x \text{ is older than } y" .
\]

We then know, for instance, that

\[
\text{fatherof}(x,y) \Rightarrow \text{olderthan}(x,y) .
\]
Instead of representing all sorts of relations—such as "fatherof" or "olderthan"—we choose the more general technique of considering different relations as named entities as well—e.g. named by "fatherof" and "olderthan", respectively—leaving us with a single universal relation—which, therefore, can remain anonymous—and represent

\[ (\text{fatherof}, x, y) \text{ and} \]
\[ (\text{olderthan}, x, y) . \]

The knowledge that

\[ (\text{fatherof}, x, y) \Rightarrow (\text{olderthan}, x, y) \]

could be represented by

\[ (\text{implies}, \text{fatherof}, \text{olderthan}) . \]

Note that what from one point of view was regarded as "the name of a relation", from another point of view can be regarded as "an argument".

Such an ordered n-tuple of names is called an association. We consider the contents of the store to be an unordered set of (different) associations. The presence of an association in store will be interpreted as the truth of the universal relation applied to the entities denoted by the members of the n-tuple.
CHAPTER 3
CHARACTERIZATION OF STATES

In Chapter 1 we have said that a condition \( p \) is a boolean function of the state. If \( p \) is applied to the state \( x \) — notation \( "p|x" \)—, it yields either the value true or the value false. When no confusion seems possible, we may forget to mention the state and write \( "p" \) instead of \( "p|x" \). This was, as a matter of fact, done in Chapter 1. We shall be explicit in cases where precision is necessary.

When programming for conventional machines, the state of a computation is uniquely defined by a state vector, the components of which comprise the values of the individual program variables. The connection between conditions on the state and these state vectors is that names of variables (identifying components of the state vector) may occur in a condition on the state. The predicate \( p|x \) is then satisfied if and only if the condition \( p \) yields the value true if in \( p \) all names of variables are replaced by their value in the state vector (representing the state) \( x \).

When programming for automata, the connection between states and conditions on the state is slightly different. The automata that are present in the store represent instances of relations between names. The evaluation of a computation is viewed as the creation of new automata, recording relations that are implied by already existing relations. One relation is fairly universal, it is the truth of "true", this will be recorded by the irrevocable presence in store of the empty automaton \( "()" \). The state of a computation is then uniquely defined by an unordered set of automata, viz. the set of all present automata. If \( (a_0, \ldots, a_{k-1}) \) — i.e. 0— is an automaton, then \( [a_0, \ldots, a_{k-1}] \) is its corresponding presence condition. Presence conditions may occur in conditions on the state. If \( u \) denotes a set of automata (e.g. the state of a computation), then the predicate \( p|u \) is satisfied if and only if the condition \( p \) yields the value true if in \( p \) all presence conditions \( "[a_0, \ldots, a_{k-1}]" \) are replaced by \( "(a_0, \ldots, a_{k-1}) \in u" \).

REMARK 3.1. From the way in which the predicate \( p|u \) is defined, it follows that
1) true | u = true ,
2) false | u = false ,
3) (p \land q) | u = (p | u \land q | u),
4) (p \lor q) | u = (p | u \lor q | u),
5) (\forall) | u = \forall(p | u).

(End of remark.)

Not every name in a presence condition has to be specified. Those names that we do not wish to specify should be replaced by a question-mark "?".

If L denotes zero or more names and unknowns, each followed by a comma, and if M denotes zero or more names, unknowns, and question-marks, each preceded by a comma, then

[L ? M] denotes \exists a: (L \land M).

NOTE. Above and in the sequel all quantified variables of which the range is not specified, are assumed to be quantified over the (in principle infinite) set of names. (In some formulae --where no confusion seems possible-- over a Cartesian product of the set of names.)

(End of note.)

EXAMPLE 3.1. Let u denote the state \{(a, b, c), (a, b, d), (a, c, c)\}. Then the following propositions are satisfied.

[[\exists x_1: [a, c, x_1 \land \{a, b, x_1\}] u ,
[[\exists x_1: [a, c, x_1 \land \{a, b, x_1\}]] u.

(End of example.)

EXAMPLE 3.2. For any state

[[x]] u = true ,
[[\neg x]] u = false.

(End of example.)

The state of a computation is (represented by) an unordered set of associations, containing the empty association. The most basic functions on unordered sets seem to be the membership test and the cardinality. Above the member-
ship test has been exploited as a "building block" for conditions on the state. To characterize states by the number of associations present, on the other hand, requires a characterization that bears little fruits, as it tells us nothing about which associations are present. More is to be expected from the number of associations present of a certain type. We shall use equations to characterize types of associations. An equation is a kind of condition on the state that may contain unknowns. As equations will play a role in the "closures statement" --to be introduced later--, we give the definition of their syntax in DNF.

\[
\text{<equation> ::= <unknowns> <term\{v<term>\}}
\]

\[
\text{<unknowns> ::= <unknown>\{|<unknown>\}; | <empty>}
\]

\[
\text{<term> ::= <factor>\{|<factor>\}}
\]

\[
\text{<factor> ::= <primary> | "<primary>}
\]

\[
\text{<primary> ::= <presence condition> | <nu> = <nu> | <nu> = \{<nu>\}}
\]

\[
\text{<presence condition> ::= [1] | [<nu>\{,<nu>\}]
\]

\[
\text{<nu> ::= <name> | <unknown>}
\]

\[
\text{<nu> ::= <nu> | ?}
\]

\[
\text{<name> ::= <identifier>}
\]

\[
\text{<unknown> ::= <identifier>}
\]

The logical operators "\wedge", "\vee", and "\neg" have their usual meaning. We can rewrite an equation into a disjunction of terms in which no parentheses "(" and ")" occur anymore. A presence condition in an equation is called negative if, after such a rewrite, it is preceded by "\neg", and it is called positive otherwise.

An equation \( E \) will in general contain unknowns. By substituting names for these unknowns, we obtain a condition on the state, that, for any given state, will either be satisfied or not. The set of all substitution instances for which the resulting condition is satisfied by \( U \), is called the solution set of \( E \) in \( U \), notation "\( Z(E,U) \)".

If \( E \) denotes the equation

\[
\overline{a_0,\ldots,a_{i-1}}: D(U_0,\ldots,U_{i-1})
\]

\((i \geq 0)\), and \( U \) some set of associations, then "\( Z(E,U) \)" denotes the set of all \( i \)-tuples of names \( \overline{a_0,\ldots,a_{i-1}} \), such that \( D(\overline{a_0,\ldots,a_{i-1}}) \subseteq U \).
In order to guarantee that for finite sets $U$, the set $Z(E,U)$ will be finite as well -- to be proven in Theorem 3.1. --, we have to be restrictive as to the way in which unknowns may occur in an equation:

After the elimination of parentheses, we may only have terms in which for each unknown of the equation there exists at least one positive presence condition in which the unknown occurs.

(We, for instance, do not allow the equation $x: [s,x] \lor \neg([b,x]$.)

**EXAMPLE 3.3.** Let $U$ denote the set $\{() , (r,s,b),(r,b,s),(s,a)\}$, $V$ the set $\{() , (r,s,b),(r,b,s),(s,b)\}$, and let $E$ denote the equation

$$x : y \bullet (r,x,y) \land \neg([s,x] .$$

Then

$Z(E,U) = \{(b,a)\}$,

$Z(E,V) = \{(a,b)\}$,

$Z(E,U \cap V) = \{(a,b),(b,a)\}$,

$Z(E,U \cup V) = \emptyset .$

(End of example.)

**EXAMPLE 3.4.** Let $U$ denote the set $\{(a,b,c)\}$, $V$ the set $\{(a,b,c)\}$, and let $E$ denote the equation

$$x: [a,?,x] .$$

Then

$Z(E,U) = \{(c)\}$,

$Z(E,V) = \{(c)\}$,

$Z(E,U \cap V) = \emptyset ,

Z(E,U \cup V) = \{(c)\}$.

(End of example.)

Note that a decrease of the number of associations (by taking the intersection) effects an increase of the number of solutions in Example 3.3, and a decrease of the number of solutions in Example 3.4.

**THEOREM 3.1.** For any equation $E$, and for any finite set of associations $U$, $Z(E,U)$ is finite.
PROOF. As a consequence of the above restrictions on occurrences of unknowns in equations, only names that are members of sets of equations in $U$ can occur in solutions of equations. As $U$ is finite, the number of names in equations in $U$ is finite as well, say $k$. If $E$ has $i$ unknowns, then the cardinality of $Z(E,U)$ does not exceed $k^i$.

(End of proof.)

The solution set $Z(E,U)$ tells us something about the set (the state) $U$. In particular, the cardinality of the solution set is a concept that will be used for characterizing states. We introduce a special notation for it:

If $U$ denotes the state of a computation, and $E$ some equation, then $"(N E)|U"$ denotes the cardinality of $Z(E,U)$.

EXAMPLE 3.5. Let $U$ denote the state $\{(),(r,a,b),(r,b,c),(r,c,d)\}$. Then

\[
\begin{align*}
(Nx,y; [r,x,y] \land [r,y,?])|U & = 2 , \\
(N [x,a,b])|U & = 1 , \\
(N [x,b,a])|U & = 0 , \\
(N [x,?,?])|U & = 1 , \\
(Nx,y; [r,x,y])|U & = 3 .
\end{align*}
\]

(End of example.)

(Usually we shall not mention the state explicitly, and write "$ (N E) "$ instead of "$ (N E)|U "$.)

REMARK 3.2. $(N E)$ is a nonnegative function of the state. As such it will often be used as the variant function in termination proofs.

(End of remark.)

The test whether the solution set $Z(E,U)$ is empty or not -- i.e., whether $(N E) = 0$ -- can yield the value true or the value false, depending on the state $U$. In the future we shall use this test like the "boolean expression" in classical programming.

* * *
In the sequel we shall use the terms set difference and symmetric set difference. If \( U \) and \( V \) denote sets, then the set difference of \( U \) and \( V \), notation \( "U \setminus V" \), is the set \( \{x : x \in U \land x \notin V\} \). (With "\( \{x : P(x)\}\)" we denote the set of all \( x \) such that \( P(x) \).) The symmetric set difference of \( U \) and \( V \), notation \( "U \symdif V" \), is the set \( (U \setminus V) \cup (V \setminus U) \).

Another concept that will be used is the match of an association and an equation. A presence condition in an equation may contain unknowns. Let a presence condition \( C \) contain the unknowns \( u_0, \ldots, u_{l-1} \) \((l \geq 0)\). An association \( A \) fits the presence condition \( C \) if and only if

\[
Z(\"u_0, \ldots, u_{l-1}\"; C(u_0', \ldots, u_{l-1}')); (A); \neq \emptyset.
\]

An association matches an equation if and only if it fits at least one presence condition in the equation. The match is called negative if the association fits a negative presence condition, and positive otherwise. (This is an asymmetric definition. Negative presence conditions can be viewed as "prohibitive regulations" on the presence of certain associations. We wish to characterize those associations that can -- for a given equation -- possibly violate such a "prohibitive regulation".)

**Example 3.6.** Let \( E \) denote the equation

\[
x, y, z \in [a, x, y] \land [b, x, y] \land \neg [b, y, z].
\]

The association \( (a, b, c) \) matches \( E \) positively. The association \( (b, a, c) \) matches \( E \) negatively.

(End of example.)

From Example 3.3 (p. 10) we know that \( Z(E, U) \) is not monotonic in \( U \), i.e. it is in general not true that

\[
U \subseteq V \implies Z(E, U) \subseteq Z(E, V).
\]

The following property, however, does hold.

**Property 3.1.** If \( E \) denotes an arbitrary equation, and \( U \) and \( V \) denote sets of associations, such that \( V \setminus U \) does not contain associations that negatively match \( E \), then

\[
U \subseteq V \implies Z(E, U) \subseteq Z(E, V).
\]

**Proof.** If \( Z(E, U) \) is empty, then the assertion of the theorem is trivially satisfied. Otherwise, let \( a \) be an arbitrary element of \( Z(E, U) \).
Writing $E$ without parentheses "(" and ")", it must contain a term $T$ such that $a \in Z(T,U)$. We prove $a \in Z(T,V)$, and hence, $a \in Z(E,V)$.

Let $v$ denote the list of unknowns of $E$, and let $T$ contain 1 $(i \geq 0)$ question-marks:

$$T(v,?,\ldots,?)$$

Let $b_0^{\prime}, \ldots, b_{i-1}$ be such that $T(a,b_0^{\prime},\ldots,b_{i-1})|U$. (From $a \in Z(T,U)$ we know that such an $i$-tuple $b_0^{\prime}, \ldots, b_{i-1}$ must exist.)

For any positive presence condition

$$\neg Cn(v,?,\ldots,?)$$

in $T$, we know from $\neg Cn(a,b_0^{\prime},\ldots,b_{i-1})|U$ and $U \subseteq V$ that

$$\neg Cn(a,b_0^{\prime},\ldots,b_{i-1})|V.$$  \hspace{1cm} (1)

For any negative presence condition $\neg Cn(v,?,\ldots,?)$ in $T$, we know, as it is not fitted by associates in $V \setminus U$,

$$\neg Cn(a,b_0^{\prime},\ldots,b_{i-1})|(V \setminus U).$$  \hspace{1cm} (2)

We furthermore know

$$\neg Cn(a,b_0^{\prime},\ldots,b_{i-1})|U.$$  \hspace{1cm} (3)

and from (2) and (3)

$$\neg Cn(a,b_0^{\prime},\ldots,b_{i-1})|V.$$  \hspace{1cm} (4)

From (1) and (4) we conclude $T(a,b_0^{\prime},\ldots,b_{i-1})|V$. Hence $T(a,?,\ldots,?)|V$, or $a \in Z(T,V)$.

(End of proof.)

We can now prove the following theorem, that will be used in Chapter 5.

THEOREM 5.2. If $E$ denotes an arbitrary equation, and $U$ and $V$ denote sets of associates, such that $U \cap V$ does not contain associates that negatively match $E$, then

$$Z(E,U \cup V) = (Z(E,U) \cap Z(E,V)).$$
PROOF. Applying Property 3.1 (p. 12), we get
\[ Z(E, U \cap V) \subseteq Z(E, U), \] and
\[ Z(E, U \cap V) \subseteq Z(E, V) \, . \]
The combination of these two yields the desired result.

(End of proof.)

REMARK 3.3. From Example 3.4 (p. 19) we know that the above theorem with
the "inclusion" replaced by an "equality" does not hold. (This is a conse-
quence of the occurrence of the question-mark in the equation.)

(End of remark.)
CHAPTER 4

AN APPRECIATION OF THE CLOSURE STATEMENT

We can distinguish two ways of appreciating programs. One way is by regarding a program as "executable code", i.e. as instructions that control the way in which, upon execution of the program, the computation proceeds through its states. The other appreciation is that a program can be viewed as a "predicate transformer", or to be more precise: as the one argument of the function "wp" that, with a predicate on the state as the other argument, yields as its value another predicate on the state. The link between these two conceptions is that the execution of a program $S$ is guaranteed to terminate in a state satisfying the predicate $P$ if and only if it is initiated in a state satisfying the predicate $wp(S, P)$. In the first --"mechanistic" or "operational"-- interpretation we can talk about implementations of programming languages, and about the efficiency of a program for a given implementation. In the second --"formal"-- interpretation time considerations, and hence efficiency considerations, do not enter the picture.

In this chapter we shall appeal to a mechanistic appreciation of the closure statement. In the two subsequent chapters we shall introduce the closure statement formally. Some of the assertions that are made plausible in this chapter, will be proved there.

In traditional programming the state of a computation is identified by a point in the state space --the earlier mentioned "state vector"--, the state space being the Cartesian product of the value sets of all variables. The basic statement is the assignment statement. It moves the point identifying the current state, parallel to one of the axes of the state space.

When programming with associations the state of a computation is identified by the set of associations present. These associations represent instances of relations between names. We shall introduce one basic statement that can change the set of associations present, viz. the closure statement.

From the relations between names that are represented by the associations present we can conclude new relations between these names. We can, for instance, from the knowledge that $x$ is the name of an integer, and that $x$
is not the name of an even integer, conclude that \( x \) is the name of an odd integer.

Suppose that the above knowledge is represented in associons of the form \( \text{int}(?,?) \) and \( \text{even}(?,?) \), i.e. suppose that for all \( x \)

\[
\text{[int},x,\text{]} \equiv "x \text{ is the name of an integer}"	ext{, and}
\text{[even},x,\text{]} \equiv "x \text{ is the name of an even integer}"	ext{,}
\]

and suppose that we wish to represent the conclusion in associons \( \text{[odd},?,\text{]} \)
--the target associons of the computation--\(,\) such that for all \( x \)

\[
\text{[odd},x,\text{]} \equiv "x \text{ is the name of an odd integer}"	ext{.}
\]

Then, in order to establish the truth of (1), which is equivalent to

\[
\text{[odd},x,\text{]} \equiv ([\text{int},x,\text{]} \land \text{[even},x,\text{]}),
\]

we would like to create for all solutions \( x \) of the equation

\[
x : [\text{int},x,\text{]} \land \lnot [\text{even},x,]\]

the associons \( \text{[odd},x,\text{]} \). The closure statement that accomplishes this creation will be denoted by

\[
x : [\text{int},x,\text{]} \land \lnot [\text{even},x,] \equiv [\text{odd},x,]\text{.}
\]

(The dotted arrow "\( \equiv \)" is pronounced as "creates".) This statement establishes the truth of the implication

\[
(\forall x : ([\text{int},x,\text{]} \land \text{[even},x,\text{]}) \equiv [\text{odd},x,])
\]

(This is not really an implication, but rather a conjunction of implications. We shall apply this abuse of language more often.)

If initially the inverse implication, i.e.

\[
(\forall x : [\text{odd},x,] \equiv ([\text{int},x,\text{]} \land \lnot [\text{even},x,]))
\]

was satisfied, e.g. because \( \lnot [\text{odd},?] \), then (3) should still hold, and we have --by enlarging the set of associons present with target associons of the format \( \text{[odd},?,\text{]} \)-- established a state satisfying (2).

We propose the following syntax for the closure statement.
<closure statement> ::= <left-hand side> :=
<target association format set>

<left-hand side> ::= <equation>
<target association format set> ::= <target association format>
{,<target association format>}
<target association format> ::= (nop,,nop)

We shall apply the notational convention that, if A denotes an assoc-
ion, then \( \bar{A} \) denotes the corresponding presence condition. (The tilde \(~\) re-
places the parentheses by brackets; e.g. if \( A \) denotes the assoc-
ion \( "(a,b)" \), then \( \bar{A} \) denotes the condition \( "[a,b]" \).) By substituting names for
the unknowns in the target association format set \( \bar{T}(x) \) of a closure statement
\( x: E(x) \Rightarrow \bar{T}(x) \)
we obtain a set of associations, say \( \bar{T}(a) \). We shall also use the tilde on
such sets of associations. \( \bar{T}(a) \) is a condition: for any set \( U \) of associations
the predicate \( \bar{T}(a) \cup U \) is satisfied if and only if \( \bar{T}(a) \in U \). (For instance,
if \( \bar{T} \) denotes the set \( "(a,b),(c,d),(e,f)" \), then \( \bar{T} \) denotes the condition
\( "[a,b] \wedge [c,d] \wedge [e,f]" \).

We then immediately have the following property.

**PROPERTY 4.1.** For all sets \( U \) and \( V \) of associations
\[
(\bar{T}(U) \wedge \bar{T}(V)) \Rightarrow \bar{T}(U \cap V).
\]

Target associations of a closure statement are associations that can be ob-
tained by substituting names for the unknowns in a target association format.

The intended effect of the execution of the closure statement
\( x: E(x) \Rightarrow \bar{T}(x) \) will be that as few target associations as possible are cre-
ated, in order to establish the truth of the implication
\[
(\forall x: E(x) \Rightarrow \bar{T}(x)). \tag{4}
\]

**EXAMPLE 4.1.** The execution of the closure statement
\( S \)
\[
x: [v,x] \Rightarrow (w,z)
\]
can cause target associations of the format \( (w,z) \) to be created. Which asso-
ciations \( (w,z) \) are created depends on the solution set of the equation
\( x: [v,x] \) in the state in which \( S \) is executed.
Let that state be \(\{(x,a),(x,b),(x,c),(x,a)\}\). Then

\(\forall x: [w,x] \Rightarrow [v,x]\)

holds. The effect of the execution of \(S\) will then be that the associations

\(\{v,b\}\) and \(\{v,c\}\)

are created, causing

\(\forall x: [v,x] \Rightarrow [w,x]\),

and consequently

\(\forall x: [v,x] \Leftrightarrow [w,x]\)

to hold.

(End of example.)

From the knowledge that \(x\) is greater than \(y\), and that \(y\) is greater than \(z\), we may (if "greater than" is transitive) conclude that \(x\) is greater than \(z\). The closure statement

\(S\)

\(x,y,z: \text{[greater,} x,y\text{]} \land \text{[greater,} y,z\text{]} \Rightarrow \text{[greater,} x,z\text{]}\)

would record such a conclusion. It establishes the truth of the implication

\(\forall x,y,z: ([\text{greater,} x,y]\land [\text{greater,} y,z] \Rightarrow [\text{greater,} x,z])\),

which is equivalent to

\(\forall x,z: \exists y: [\text{greater,} x,y]\land [\text{greater,} y,z] \Rightarrow [\text{greater,} x,z]\). (5)

Due to the positive match of the target association \(\text{[greater,} x,y\text{]}\) and the equation of \(S\), (5) will (in general) not be established by creating for all initial solutions \((x,y,z)\) of the left-hand side of \(S\) the absent associations \(\text{[greater,} x,z\text{]}\). For these creations can enlarge the solution set of the left-hand side of \(S\) again, causing that new target associations have to be created, etc. We call such a closure statement cascading. Or, denoting by a "constant" a "member that is not an unknown or a question-mark”,

a closure statement is said to be cascading if and only if it contains a presence condition and a target association format of the same length, and these do not have different constants at corresponding positions.

(Whether a closure statement is a cascading closure statement can hence be established statically.)
EXAMPLE 4.2. Transitive closure. Given a finite set \( W \) on which a binary relation \( S \) is defined, then the transitive closure of \( S \), notation \( \bar{S} \), is defined by

1) \( (\forall x,y \in W: S(x,y) \Rightarrow \bar{S}(x,y)) \),
2) \( (\forall x,z \in W: (\exists y \in W: \bar{S}(x,y) \land \bar{S}(y,z)) \Rightarrow \bar{S}(x,z)) \),
3) \( \bar{S} \) is only true for those arguments for which it is true on account of 1) and 2).

Let a finite set \( W \) and a binary relation \( S \) on \( W \) be given by

\[
(\forall x: [x,x] \in W \equiv "x is the name of an element of W") \land \\
(\forall x,y: [s,x,y] \equiv ([w,x] \land [w,y] \land S(x,y))) .
\]

REMARK 4.1. The expression "\( S(x,y) \)" in the above formula should be read as " \( S \) applied to the arguments of which the names are \( x \) and \( y \)". We shall apply this abuse of language more often.

(End of remark.)

Let, furthermore, be given that \( \neg[t,?,?] \) holds, and let it be requested to write a program that establishes the truth of the relation \( R \)

\[
(\forall x,y: [t,x,y] \equiv ([w,x] \land [w,y] \land \bar{S}(x,y))) .
\]

The definition of the transitive closure suggests the program

\[
x,y : [x,y] \Rightarrow [x,y] ;
\]
\[
x,y,z : [t,x,y] \land [t,y,z] \Rightarrow (t,x,z) .
\]

It establishes the implication of the left-hand side of \( R \) by the right-hand side of \( R \), under invariance of the initially holding implication in the other direction.

(End of example.)

As it is now, the execution of a closure statement cannot only enlarge the solution set of its left-hand side, but the creation of target associations could also reduce it. This is a very unattractive situation.

Let e.g. the state be \( \{(),(r,a,b),(r,b,a)\} \), and let it be requested to establish the truth of

\[
(\forall y: (\exists x: [x,x,y] \land \neg[S,x]) \Rightarrow [x,y]) ,
\]

which is equivalent to

\[
(\forall y: (\exists x: [x,x,y] \land \neg[S,x]) \Rightarrow [x,y]) .
\]
\((\forall x, y: \{x, y\} \land \neg[s, x] = [s, y]\).\)

Suppose we would allow the following statement to establish this implication:
\(x, y: \{x, y\} \land \neg[s, x] = (s, y)\).

The equation in the left-hand side has two solutions: "x = a, y = b" and "x = b, y = a". For both solutions the corresponding target associations \((s, y)\) are absent. This does not necessarily imply that both \((s, a)\) and \((s, b)\) should be created, as \((6)\) should be established by creating as few associations \((s, ?)\) as possible. As the creation of one of them already establishes \((6)\), either \((s, a)\) or \((s, b)\) should be created, but not both. This phenomenon would severely complicate the concurrent creation of target associations, and would as such unsettle our whole design.

We, therefore, wish to avoid that the creation of target associations can reduce the solution set on the left-hand side. From Property 3.1 (p. 12) we know that, if the created target associations do not match the equation in the left-hand side negatively, then its solution set can never shrink.

Hence we forbid this negative match:

If a closure statement contains a presence condition and a target associn format of the same length, and these do not have different constants at corresponding positions, then the presence condition must be a positive one.

(Whether a closure statement obeys this rule can be established statically.)

By prohibiting this negative match the closure statement becomes a deterministic construct. As a consequence its effect cannot depend on the amount of concurrency in the implementation of its activity. In order to establish the truth of \((4)\) the implementation could, as long as
\((\exists x: (\exists a: \alpha \in \tau(x) \land \pi(x) \land \neg\bar{s}(x)))\),
create such an association \(\alpha\). How much of this is done concurrently, is up to the implementation.

We know that information destruction is essential for all nontrivial computing, as a computation would otherwise merely be a reversible transformation of the initial state. How is information destroyed if the closure statement is our only basic statement? It may sound contradictory -- to the novice at least --, but the execution of a closure statement will in general
indeed cause information to be destroyed. The reason for this is that the
mapping from initial state to final state is in general not one-to-one. If
the execution of a closure statement $S$ in state $U_0$ transforms the state
into $U_1$, then the execution of $S$ in state $U_0$ would also lead to the
final state $U_1$.

Still, one might wish—in order to attach transient meanings to asso-
ciations—to destroy associations. We accomplish this by allowing as a statement
a block, which is a statement list surrounded by the delimiters "loc" and
"col". This statement list is the scope of all associations of the forms
listed after the delimiter "loc":

\[
\langle \text{block}\rangle := \text{loc} \langle \text{association format}\rangle \{,\langle \text{association format}\rangle\};
\]

\[
\langle \text{statement list}\rangle := \text{col}
\]

\[
\langle \text{association format}\rangle := \{\{\langle \text{np}\rangle,\} \langle \text{name}\rangle \{,\langle \text{np}\rangle\} \}
\]

\[
\langle \text{np}\rangle := \langle \text{name}\rangle | ?
\]

Upon "block entry" we have for all associations $A$ of the specified for-
mant $\overline{T}^n$. Upon "block exit" all local associations (all associations of the speci-
fied format that have been created during the execution of the statement
list) are destroyed. The logical need of local associations will not arise un-
til the introduction of the repetitive construct.

**Remark 4.2.** The closure statement can easily be generalized into a concur-
rent closure statement. Such a concurrent closure statement is a set of clo-
sure statements in which no target association can negatively match any of the
equations. The concurrent closure statement

\[
\{x_0 : E_0(x_0) \Rightarrow T_0(x_0), \ldots , x_{n-1} : E_{n-1}(x_{n-1}) \Rightarrow T_{n-1}(x_{n-1})\}
\]

establishes—by creating target associations of $T_0, \ldots , T_{n-1}$—the truth of

\[
(\forall x_0 : E_0(x_0) \Rightarrow T_0(x_0)) \land \ldots \land (\forall x_{n-1} : E_{n-1}(x_{n-1}) \Rightarrow T_{n-1}(x_{n-1})) .
\]

In order to achieve this, the different members of the concurrent closure
statement may be executed in any order— even concurrently— and should be
executed until all of them have finished, i.e. until for all $i$ ($0 \leq i < n$)

\[
(\forall x_i : E_i(x_i) \Rightarrow T_i(x_i))
\]

holds.
Any closure statement can be implemented by a concurrent closure statement of which all members have a conjunction of two factors --not containing any parentheses "(" and ")"-- as a left-hand side. We shall illustrate this with an example.

An out-tree is a directed rooted tree, directed in such a way that each vertex is reachable from the root. (When dealing with graphs we mainly employ the terminology as defined in [5].) If an out-tree \( T \) is given by

\[
(\forall x : \{v,x\} \ni x \text{ is the name of a vertex of } T \land \\
(\forall x,y : \{s,x,y\} \ni \{[v,x] \land [v,y] \land \text{ } T \text{ has an arc from } x \text{ to } y \})
\]

then the closure statement

\[
x,y,z : \{a,x,y\} \land \{b,y,z\} \\
\land ((\{v,x\} \land \{a,b,x\} \land \{ev,x\}) \Rightarrow (ev,z))
\]  

(7)

would record all "even" vertices of \( T \), i.e., all vertices for which the path from the root comprises an even number (zero excluded) of arcs. This statement could be implemented by the following concurrent closure statement, that we have written as a block.

\[
\begin{align*}
\text{labels } \{01,02,03\} : \\
\{x,y,z : \{a,x,y\} \land \{b,y,z\} : (x1,x,z), \\
x : \{v,x\} \land \{a,b,x\} : (02,x), \\
x,z : \{a1,x,z\} \land \{b2,x\} \Rightarrow (ev,z), \\
x,z : \{b1,x,z\} \land \{ev,x\} \Rightarrow (ev,z)
\end{align*}
\]

(We could have coded statement (7) more elegantly in two or three statements. We have not done so, because we wished to illustrate the systematic translation.)

The fact that any closure statement can be written as a set of "simple" closure statements, should give us confidence in the implementability of arbitrarily complex closure statements. (It is the analogue of the phenomenon that any arithmetic expression can be written as a succession of binary operations.)

(End of remark.)
CHAPTER 5

CLOSURE OF A SET OF ASSOCIATIONS

In Chapter 4 we have described the closure statement in a mechanistic fashion. In Chapter 6 we shall give the formal definition of the effect of the execution of a closure statement. In this chapter we lay the foundation for the formal definition by studying sets of associations and their relations. In particular we shall study the concept of a closure of a set of associations with respect to a closure statement. This treatment will not depend on the mechanistic appreciation of the closure statement we acquired in Chapter 4. From Chapter 4 we shall only use the knowledge which texts constitute legitimate closure statements.

We apply the notational convention that, if \( S \) denotes the closure statement

\[
x: E(x) \Rightarrow T(x),
\]

then \( \hat{S} \) denotes the condition

\[
(\forall x: E(x) \Rightarrow \hat{T}(x)).
\]

(If, for instance, \( S \) denotes the statement \( x: [v,x] \Rightarrow (v,x) \), then \( \hat{S} \) denotes the condition \( (\forall x: [v,x] \Rightarrow (v,x)) \).

**Property 5.1.** If \( S \) denotes the closure statement \( x: E(x) \Rightarrow T(x) \), and \( U \) some set of associations, then

\[
\hat{S}|U \equiv (\forall x: x \in E(U) \Rightarrow \hat{T}(x))|U.
\]

**Proof.**

\[
\hat{S}|U \equiv (\forall x: E(x) \Rightarrow \hat{T}(x))|U.
\]

According to Remark 3.1 (p. 7) we may distribute the \( U \) over the implication, which yields

\[
\hat{S}|U \equiv (\forall x: [E(x)]|U \Rightarrow \hat{T}(x)|U).
\]

According to the definition of the solution set

\[
E(U)|U \equiv x \in E(U),
\]

which proves the property to hold.

(End of proof.)
THEOREM 5.1. If $S$ denotes an arbitrary closure statement, and $U$ and $V$ denote sets of associations, such that $U \cup V$ does not contain associations that negatively match the left-hand side of $S$, then

$$((\tilde{S}|U) \land (\tilde{S}|V)) \Rightarrow \tilde{S}|(U \cap V).$$

PROOF. Let $S$ denote the closure statement $x : E(x) := T(x)$. We assume

$$(\tilde{S}|U) \land (\tilde{S}|V), \tag{1}$$

and we derive $\tilde{S}|(U \cap V)$. From (1) and Property 5.1 (p. 23) we conclude

$$(\forall x : x \in Z(E,U) : \tilde{T}(x) | U) \land (\forall x : x \in Z(E,V) : \tilde{T}(x) | V),$$

which implies

$$(\forall x : x \in (Z(E,U) \cap Z(E,V))) : (\tilde{T}(x) | U) \land (\tilde{T}(x) | V),$$

or (apply Property 4.1 (p. 17))

$$(\forall x : x \in (Z(E,U) \cap Z(E,V))) : \tilde{T}(x) | (U \cap V), \tag{2}$$

As $U \cup V$ does not contain associations that negatively match $E$, we may apply Theorem 3.2 (p. 13), yielding

$$Z(E,U \cap V) \subseteq (Z(E,U) \cap Z(E,V)). \tag{3}$$

From (2) and (3) we conclude

$$(\forall x : x \in Z(E,U \cap V)) : \tilde{T}(x) | (U \cap V),$$

or (apply Property 5.1 (p. 23))

$$\tilde{S}|(U \cap V).$$

(End of proof.)

The following lemma is a consequence of the above theorem.

LEMMA 5.1. If $U$ denotes some finite set of associations, and $S$ an arbitrary closure statement, and if $W_1$ and $W_2$ are sets $W$ of associations satisfying

1) $U \subseteq W$,
2) $W \setminus U$ does not contain associations that negatively match the left-hand side of $S$,
3) $\tilde{S}|W$,

then $W_1 \cap W_2$ is also a set $W$ of associations satisfying these properties.
PROOF.

1) \((U \in W_1) \land (U \in W_2)\) implies \(U \in (W_1 \cap W_2)\).

2) \((W_1 \cap W_2) \setminus U = (W_1 \setminus U) \cap (W_2 \setminus U)\).

3) As

\[W_1 + W_2 \subset (W_1 \setminus U) \cup (W_2 \setminus U),\]

the set \(W_1 + W_2\) does not contain associations that negatively match the
left-hand side of \(S\). This allows us to apply Theorem 5.1 (p. 23),
yielding
\[(\tilde{S}[W_1] \land \tilde{S}[W_2]) = \tilde{S}[W_1 \cap W_2].\]

(End of proof.)

LEMMA 5.2. If \(U\) denotes some finite set of associations, and \(S\) an arbitrary
closure statement, then there exists a finite set \(W_0\), satisfying properties 1), 2), and 3) of Lemma 5.1 (p. 24).

PROOF. Let \(V\) denote the set of all names occurring in associations of \(U\) or
occurring (as constants) in the target association formats of \(S\). As \(U\) is
finite, \(V\) will be finite as well.

\(W_0\) is defined as the union of \(U\) and the set of all target associations
of \(S\) that can be obtained by substituting names of \(V\) for the unknowns in
the target association formats. Then \(W_0\) is finite and it satisfies property
1). As target associations do not negatively match the left-hand side, \(W_0\)
also satisfies property 2).

We still have to prove \(\tilde{S}[W_0]\), or, with \(S\) denoting the closure statement
\(X: E(x) \Rightarrow T(x)\), (apply Property 5.1 (p. 23))
\[(\forall x: x \in Z(S,W_0): T(x) \subseteq W_0).\]

If \(x \in Z(S,W_0)\), then ---cf. the proof of Theorem 3.1 (p. 10)--- \(x\) contains
only names occurring in associations of \(W_0\), i.e. names in \(V\). But then
\(T(x) \subseteq W_0\).

(End of proof.)

A consequence of the above two lemmata is that for any closure statement \(S\) there exists, for any finite set of associations \(U\), a unique
smallest set ---i.e. with the smallest cardinality--- \(W\) of associations satis-
ifying properties 1), 2), and 3). This unique smallest set is called the
Theorem 5.2. If \( U \) denotes some finite set of associations, and \( S \) an arbitrary closure statement, then \( C(S,U) \) is finite.

Property 5.2. If \( U \) denotes some finite set of associations, \( S \) an arbitrary closure statement, and \( p \) some presence condition, then

\[ p|U = p|C(S,U) \]

Proof. \( U \subset C(S,U) \).

(End of proof.)

Property 5.3. If \( U \) denotes some finite set of associations, and \( S \) the closure statement \( x; R(x) \Rightarrow T(x) \), then for all \( x \)

\[ E(x)|U = E(x)|C(S,U) \]

Proof. \( U \subset C(S,U) \). As \( C(S,U) \setminus U \) does not contain associations that negatively match \( S \), we may apply Property 3.1 (p. 12), yielding

\[ Z(R,U) \subset Z(R,C(S,U)) \]

According to the definition of the solution set, this is equivalent to the property to be proved.

(End of proof.)

Property 5.4. If \( U \) denotes some finite set of associations, and \( S \) an arbitrary closure statement, then

\[ S|C(S,U) \]

Proof. Consequence of the definition of \( C(S,U) \).

(End of proof.)

Property 5.5. If \( S \) denotes some closure statement, and \( U \) and \( V \) denote sets of associations, such that \( U = V \setminus U \) does not contain associations that negatively match the left-hand side of \( S \), then

\[ C(S,U) \subset C(S,V) \]

Proof. We prove that \( C(S,V) \) satisfies properties 1), 2), and 3) of Lemma 5.1 (p. 24). This lemma then learning us that the set \( C(S,U) \cap C(S,V) \) satisfies them too. With \( C(S,U) \) being the smallest such set, we then have
\( C(S, U) \subseteq C(S, V) \).

1) \( (U \subseteq V) \land (V \subseteq C(S, V)) \) implies \( U \subseteq C(S, V) \).
2) \( C(S, V) \setminus U = (C(S, V) \setminus V) \cup (V \setminus U) \).

By definition, \( C(S, V) \setminus V \) does not contain assocations that negatively match the left-hand side of \( S \). It is given that \( V \setminus U \) does not contain them either. Therefore, \( C(S, V) \setminus U \) does not contain assocations that negatively match the left-hand side of \( S \).

3) According to Property 5.4 (p. 26), \( \bar{S}[C(S, V)] \).

(End of proof.)

**Property 5.6.** If \( U \) denotes some finite set of assocations, and \( S \) some closure statement, then

\( \bar{S}[U] \subseteq C(S, U) = U \).

**Proof.** By definition, \( U \subseteq C(S, U) \). \( \bar{S}[U] \) implies that \( U \) satisfies properties 1), 2), and 3) of Lemma 5.1 (p. 24). As \( C(S, U) \) is the smallest such set, we have \( U = C(S, U) \).

(End of proof.)

**Property 5.7.** If \( U \) denotes some finite set of assocations, and \( S \) an arbitrary closure statement, then

\( C(S, C(S, U)) = C(S, U) \).

**Proof.** Consequence of Properties 5.4 (p. 26) and 5.6.

(End of proof.)

**Property 5.8.** If \( U \) denotes some finite set of assocations, and \( S \) the closure statement \( x: E(x) \Rightarrow T(x) \), then

\( C(S, U) \subseteq U \cup \{A: (\exists x: x \in E(S, C(S, U)) \land A \subseteq T(x))\} \).

**Proof.** Let \( V \) denote the set

\( \{A: (\exists x: x \in E(S, C(S, U)) \land A \subseteq T(x))\} \).

We first prove \( C(S, U) \subseteq (U \cup V) \) by a reductio ad absurdum. Suppose there exists an assocation \( A \), such that \( A \in C(S, U) \land A \notin U \land A \notin V \).

Let \( W \) denote the set \( C(S, U) \setminus \{A\} \). If we can show that \( W \) satisfies properties 1), 2), and 3) of Lemma 5.1 (p. 24), then we have found a smaller set than \( C(S, U) \) satisfying these properties, and we have derived a contra-
1) From \( U \subseteq C(S,U) \) follows \( U \setminus \{A\} \subseteq C(S,U) \setminus \{A\} \), or \( U \in W \).

2) As \( C(S,U) \setminus U \) does not contain associons that negatively match \( E \), and 
\( (W \setminus U) \subseteq C(S,U) \setminus U \), the set \( W \setminus U \) does not contain them either.

3) From \( A \notin V \) and the definition of \( V \) we conclude
\[
(Vx: x \in E(S,C(S,U))): A \notin T(x).
\] From \( \hat{S}|C(S,U) \) we may, according to Property 5.1 (p. 23), conclude
\[
(Vx: x \in E(S,C(S,U))): T(x) \subseteq C(S,U).
\]
From (4), (5), and the definition of \( W \) we conclude
\[
(Vx: x \in E(S,C(S,U))): T(x) \subseteq W.
\]
The set \( C(S,U) \setminus W \) contains the associon \( A \) only. As \( A \subseteq C(S,U) \setminus U \), \( A \) does not negatively match \( E \). This allows us to apply Property 3.1 (p. 12) on \( W \) and \( C(S,U) \), yielding
\[
2(E,W) \subseteq 2(E,C(S,U)).
\]
From (6) and (7) we conclude
\[
(Vx: x \in E(W): T(x) \subseteq W),
\]
or (apply Property 5.1 (p. 23)) \( \hat{S}|W \).

Next we prove \( (U \cup V) \subseteq C(S,U) \) by a reductio ad absurdum as well.

Suppose there exists an associon \( A \), such that
\[
A \in V \land A \notin C(S,U).
\]
From the definition of \( V \) we conclude the existence of an \( x \in 2(E,C(S,U)) \), such that
\[
A \subseteq T(x).
\]
From \( \hat{S}|C(S,U) \) and \( x \in E(S,C(S,U)) \) we deduct, by applying Property 5.1 (p. 23),
\[
T(x) \subseteq C(S,U).
\]
From (8) en (9) follows \( A \subseteq C(S,U) \), which contradicts our assumption.

Hence, we have proved
\[
V = C(S,U),
\]
which, combined with \( U \subseteq C(S,U) \), yields
\[(U \cup V) \subseteq C(S, U) \]

(End of proof.)

The above property shows that \( C(S, U) \setminus U \) contains target associations only. The following property is a direct consequence.

**PROPERTY 5.9.** If \( U \) denotes some finite set of associations, \( p \) some presence condition, and \( S \) a closure statement of which no target association fits \( p \), then

\[ p \mid C(S, U) \equiv p \mid U. \]

**PROPERTY 5.10.** If \( U \) denotes some finite set of associations, and \( S \) the noncascading closure statement \( \forall x \in \mathbb{S}(x) \implies T(x) \), then

\[ C(S, U) = U \cup \{ A \mid (\exists x : x \in \mathbb{S}(x) : A \in T(x)) \}. \]

**PROOF.** In a noncascading closure statement target associations do not match \( x \). Hence, we have

\[ \mathbb{Z}(S, U) = \mathbb{Z}(K, C(S, U)). \]

The result now follows directly from Property 5.8 (p. 27).

(End of proof.)

The following property will turn out to be important in Chapter 6. It will then give rise to the invariance theorem for closure statements.

**PROPERTY 5.11.** Let \( S \) denote the closure statement \( \forall x \in \mathbb{S}(x) \implies T(x) \), \( A \) an arbitrary association, and \( U \) some finite set of associations. If

\[ A \in U \iff (\exists x : x \in \mathbb{S}(x) : A \in T(x)) \]

(10)

then

\[ A \in C(S, U) \iff (\exists x : x \in \mathbb{S}(C(S, U)) : A \in T(x)) \].

(11)

**PROOF.** We prove that if (11) is not satisfied, then (10) is not satisfied either, i.e., we assume

\[ A \in C(S, U) \land \neg (\exists x : x \in \mathbb{S}(C(S, U)) : A \in T(x)) \]

(12)

and we derive

\[ A \in U \land \neg (\exists x : x \in \mathbb{S}(U) : A \in T(x)) \].

(13)
According to Property 5.8 (p. 27) (12) implies
\[ A \in U \land \exists x \in Z(E,C(S.U)): A \in T(x) \quad (14) \]
According to Property 5.3 (p. 26) \( A \in Z(E.U) \) implies \( x \in Z(E,C(S.U)) \).
Consequently, (14) implies (13).
(End of proof.)

REMARK 5.1. For non cascading closure statements we have (10) if and only if (11).
(End of remark.)

Property 5.10 (p. 29) gives a constructive characterization of \( C(S,U) \) for non cascading closure statements. We shall generalize this into a constructive characterization of \( C(S,U) \) for arbitrary closure statements. The equivalence of this characterization and our earlier definition will be proved in Theorem 5.3 (p. 31).

We first define, for a finite set \( U \) of assoccs, and for an arbitrary closure statement \( x: E(x) = T(x) \), sets \( F_1 \) of assoccs \((i \geq 0)\) as follows.
\[
\begin{align*}
F_0 & = U, \\
F_{i+1} & = F_i \cup \{A: \exists x \in Z(E,F_i): A \in T(x)\}.
\end{align*}
\]

LEMMA 5.3. If \( U \) denotes some finite set of assoccs, and \( S \) the closure statement \( x: E(x) = T(x) \), then for all \( i \geq 0 \)
\[ F_i \subseteq C(S,U). \]

PROOF. We prove \( F_i \subseteq C(S,U) \) by mathematical induction. Obviously, \( F_0 \subseteq C(S,U) \). Suppose \( F_k \subseteq C(S,U) \) \((k \geq 0)\). We prove \( F_{k+1} = C(S,U) \).

By definition,
\[ F_{k+1} = F_k \cup \{A: \exists x \in Z(E,F_k): A \in T(x)\} \quad (15) \]

From Property 5.3 (p. 26) we know
\[ Z(S,F_k) \subseteq Z(E,C(S,F_k)) \quad (16) \]

From (15) and (16) we conclude
\[ F_{k+1} \subseteq \{F_k \cup \{A: \exists x \in Z(E,C(S,F_k)): A \in T(x)\}\}. \]
or (apply Property 5.8 (p. 27))

\[ F_{k+1} \subseteq C(S, F_k) \]  \hspace{1cm} (17)

As \( U \subseteq F_k \), we have

\[ (C(S, U) \setminus F_k) \subseteq (C(S, U) \setminus U) \]  

C(S, U) \setminus F_k, consequently, does not contain assiosions that negatively match \( E \). This allows us to apply Property 5.3 (p. 26), yielding

\[ C(S, F_k) \subseteq C(S, C(S, U)) \]  

or (apply Property 5.7 (p. 27))

\[ C(S, F_k) \subseteq C(S, U) \]  \hspace{1cm} (18)

From (17) and (18) we conclude

\[ F_{k+1} \subseteq C(S, U) \]  

(End of proof.)

From \( F_i \subseteq F_{i+1} \) and \( F_i \subseteq C(S, U) \) we conclude, as \( C(S, U) \) is finite, that \( \lim_{i \to \infty} F_i \) exists and is finite. The following theorem expresses that this limit is exactly \( C(S, U) \).

THEOREM 5.1. If \( U \) denotes some finite set of assiosions, and \( S \) an arbitrary closure statement, then

\[ C(S, U) = \lim_{i \to \infty} F_i \]

PROOF. Let \( S \) denote the closure statement \( \forall x : E(x) \Rightarrow \tau(x) \), and let \( j \) be such that \( \lim_{i \to \infty} F_i = F_j \). Then \( F_j = F_{j+1} \). We prove that \( F_j \) satisfies properties 1), 2), and 3) of Lemma 5.1 (p. 24). Obviously, \( F_j \) satisfies properties 1) and 2). From \( F_j = F_{j+1} \) and the definition of \( F_{j+1} \), i.e.,

\[ F_{j+1} = F_j \cup \{ A : (\exists x : x \in E(F_j) : A \in \tau(x)) \} \]

we conclude

\[ \{ A : (\exists x : x \in E(F_j) : A \in \tau(x)) \} \subseteq F_j \]  

or

\[ \{ \forall x : x \in E(F_j) : \tau(x) \in F_j \} \]
or (apply Property 5.1 (p. 23)) \( \overline{z_j} \). Hence, \( F_j \) satisfies property 3 as well. From Lemma 5.1 (p. 30) we know \( F_j \subseteq C(S, U) \). As \( C(S, U) \) is the smallest set satisfying properties 1), 2), and 3), we have \( C(S, U) = F_j \).

(End of proof.)

PROPERTY 5.12. If \( U \) denotes some finite set of associences, and \( S_1 \) and \( S_2 \) denote arbitrary closure statements, then

\[
C(S_2, C(S_1, U)) = C(S_2, U)
\]

if and only if

\[
C(S_1, U) \subseteq C(S_2, U).
\]

PROOF. The fact that (19) implies (20) is an immediate consequence of the definition of \( C(S, U) \). Next we assume (20) and derive (19). As

\[
(C(S_2, U) \setminus C(S_1, U)) \subseteq (C(S_2, U) \setminus U),
\]

\( C(S_2, U) \setminus C(S_1, U) \) does not contain associences that negatively match the left-hand side of \( S_2 \). This allows us to apply Property 5.5 (p. 26) on \( C(S_1, U) \) and \( C(S_2, U) \), yielding

\[
C(S_2, C(S_1, U)) = C(S_2, C(S_2, U)),
\]

or (apply Property 5.7 (p. 27))

\[
C(S_2, C(S_1, U)) \subseteq C(S_2, U).
\]

As

\[
(C(S_1, U) \setminus U) \subseteq (C(S_2, U) \setminus U),
\]

we may apply Property 5.5 (p. 26) on \( U \) and \( C(S_1, U) \) as well, yielding

\[
C(S_2, U) \subseteq C(S_2, C(S_1, U)).
\]

From (21) and (22) follows (19).

(End of proof.)

One might wonder whether the \( C(S, U) \) is a closure operator as defined in [12]. This is not the case. The \( C(S, U) \) does not satisfy the fourth Kuratowski closure axiom, requiring that the closure of the union of two sets equals the union of their closures. (Take, e.g., \( U = \{u\} \), \( V = \{v\} \), and \( S = [u] \leadsto (w) \). Then \( C(S, U) \cup C(S, V) = C(S, U \cup V) \).

It is more "powerful" than the closure of a finite set under a binary relation —for definition see below—, which does satisfy the Kuratowski clo-
sure axioms. The closure of a set under a binary relation can be expressed in terms of \( C(S,U) \) —see the following example—, the converse is not true.

**EXAMPLE 5.1.** Let \( V \) denote a finite set on which a binary relation \( R \) is defined, and let \( D \) denote some subset of \( V \). Let, furthermore, sets \( C_i \) \((i \geq 0)\) be given by the recurrence relation

\[
C_0 = D, \\
C_{i+1} = C_i \cup \{ x; x \in V; (\exists y \in C_i; R(y,x)) \}.
\]

Then the closure of \( D \) under \( R \), notation \( D^R \), is defined by

\[
D^R = \lim_{i \to \infty} C_i.
\]

(The above limit exists as for all \( i \ (i \geq 0) \) \( C_i \subseteq C_{i+1} \subseteq V \).)

Let \( V \), \( D \), and \( R \) be given. Let the state \( U \) satisfy

\[
\forall x, y \in V; (x, y) \in U \\
\forall x \in V; (x, x) \in U \\
\forall x, y, z \in V; (x, y) \in U \wedge (y, z) \in U \Rightarrow (x, z) \in U.
\]

Let, furthermore, \( S \) denote the closure statement

\[
x, y; [x, y] \in [x, y, z] \iff (x, z).
\]

We shall prove

\[
\forall x, y \in V; (x, y) \not\in U \Rightarrow (x, y) \not\in D^R.
\]

We prove, by mathematical induction, that for all \( x \) and \( y \)

\[
([x, y], [x, y, z]) \subseteq [x, y] \cup (x, y) \cup R(y, x)
\]

(24)

Then, as \( \lim_{i \to \infty} F_i = C(S,U) \) and \( \lim_{i \to \infty} C_i = D^R \), (23) follows immediately.

Relation (24) is satisfied for \( i = 0 \). Suppose, for all \( x \) and \( y \)

\[
([x, y], [x, y, z]) \subseteq [x, y] \cup (x, y) \cup R(y, x)
\]

(25)

\((k \geq 0)\). We derive

\[
([x, y], [x, y, z]) \subseteq [x, y] \cup (x, y) \cup R(y, x)
\]

(26)

By definition,
\[ x \in G_{k+1} = (x \in G_k \lor (\exists y: y \in G_k \land R(y, x))) \tag{27} \]

By applying the definition of \( F_k \) we obtain

\[
[d, x]_{F_{k+1}} \equiv (d, x)_{F_k} \lor (\exists y: (d, y) \land [x, y, x]_{F_k}) \land [x, y, x]_{F_{k+1}} \equiv [x, y, x]_{F_k} \tag{28}
\]

From (25), (27), and (28) follows (26).

(End of example.)

**Remark 5.2.** The transitive closure, as defined in Example 4.2 (p. 19), can be expressed as a closure of a set under a binary relation. Let \( S \) be a binary relation on a finite set \( W \). Define a binary relation \( R \) on \( WW \) as

\[ R((x_0, x_1), (y_0, y_1)) \equiv (x_0 = y_0 \land S(x_1, y_1)) \]

and define the set \( D \) to be \( \{(x, y) : x, y \in W : S(x, y)\} \). \( D \) is then a subset of \( WW \), and we state without proof

\[ \bar{S}(x, y) \equiv (x, y) \in D^R \]

(End of remark.)
CHAPTER 6

FORMAL DEFINITION OF THE CLOSURE STATEMENT

In the preceding chapter we have laid the foundation for the formal
definition of the semantics of the closure statement. It is our intention
to have the closure statement $S$ transform the state $U$ into the state
$C(S,U)$. What in effect should happen is the assignment "$U := C(S,U)". We,
therefore, define, in analogy to the Axiom of Assignment (see Example 1.1
(p. 4)), the weakest pre-condition of the closure statement as follows.

If $S$ denotes an arbitrary closure statement, and $P$ some condition
on the state, then for all states $U$

$$\wp(S,P) \cup U \equiv P \cap C(S,U)$$

(Like the assignment statement, it satisfies Properties 1.1, 1.2, and 1.3
(p. 3) for predicate transformers.)

PROPERTY 6.1. If $S$ denotes an arbitrary closure statement, and $P$ some
condition on the state, then

$$\neg \wp(S,P) \equiv \wp(S,\neg P)$$

PROOF. We use Remark 3.1 (p. 7). For any state $U$

$$(\neg \wp(S,P)) \cup U \equiv \neg \wp(S,P) \cup U$$
$$\equiv \neg (P \cap C(S,U))$$
$$\equiv \neg (U) \cap C(S,U)$$
$$\equiv \wp(S,\neg P) \cup U$$

(End of proof.)

By substituting "false" --or "true"-- for "P" in the above property, we
obtain

$$\wp(S,\text{true}) \equiv \text{true}$$

which is interpreted as the guaranteed termination of closure statements.

THEOREM 6.1. The finiteness of the set of assertions that characterizes the
state, is an invariant of closure statements.

PROOF. Consequence of Theorem 5.2 (p. 26).

(End of proof.)
THEOREM 6.2. The closure statement is a deterministic statement.

PROOF. According to the definition in Chapter 1, a statement $S$ is deterministic if and only if for any two conditions $P$ and $Q$, and for all states $U$

\[ \wp(S, P \lor Q) | U = (\wp(S, P) \lor \wp(S, Q)) | U. \]

Let $S$ denote an arbitrary closure statement. We use Remark 3.1 (p. 7) and the definition of the closure statement to obtain

\[ \wp(S, P \lor Q) | U = \left( \wp(S, P) | U \lor \wp(S, Q) | U \right) \]

\[ = \left( \wp(S, P) | U \lor \wp(S, Q) | U \right) \]

\[ = \left( \wp(S, P) \lor \wp(S, Q) \right) | U. \]

(End of proof.)

From the properties of closures, as proved in Chapter 5, we can derive equivalent properties of closure statements.

PROPERTY 6.2. If $S$ denotes an arbitrary closure statement, and $p$ some presence condition, then

\[ p \equiv \wp(S, p). \]

PROOF. Consequence of Property 5.2 (p. 26).

(End of proof.)

PROPERTY 6.3. If $S$ denotes an arbitrary closure statement, and $p$ a presence condition, such that no target association of $S$ fits $p$, then

\[ p \equiv \wp(S, p). \]

PROOF. Consequence of Property 5.9 (p. 29).

(End of proof.)

Property 6.2 expresses that the execution of a closure statement does not destroy associations, Property 6.3 expresses that only target associations are created.

PROPERTY 6.4. If $S$ denotes the closure statement $x : E(x) \Rightarrow T(x)$, then for all $x$

\[ E(x) \Rightarrow \wp(S, E(x)). \]
PROOF. Consequence of Property 5.3 (p. 26).
(End of proof.)

PROPERTY 6.5. If \( S \) denotes an arbitrary closure statement, then

\[
wp(S,\hat{S}) \cong true.
\]

PROOF. Consequence of Property 5.4 (p. 26).
(End of proof.)

PROPERTY 6.6. If \( S \) denotes an arbitrary closure statement and \( P \) some
condition on the state, then

\[
\hat{S} \text{ implies } wp(S,P) \cong P.
\]

PROOF. Let \( U \) denote an arbitrary state. We assume \( \hat{S} \upharpoonright U \). Then, according to Property 5.6 (p. 27),

\[
C(S,U) = U.
\]

By definition,

\[
wp(S,P)\upharpoonright U \cong P\upharpoonright C(S,U).
\]

From (1) and (2) we conclude

\[
wp(S,P)\upharpoonright U \cong P\upharpoonright U.
\]

(End of proof.)

Property 6.5 expresses that a closure statement \( S \) establishes the
truth of \( \hat{S} \). Property 6.6 expresses that, if \( \hat{S} \) was true to start with,
then \( S \) is equivalent to the statement "skip".

PROPERTY 6.7. If \( S \) denotes an arbitrary closure statement, and \( P \) some
condition on the state, then

\[
wp(S,S',P) \cong wp(S,P).
\]

PROOF. By definition,

\[
wp(S,S',P) \cong wp(S,wp(S,P)).
\]

Therefore, for any state \( U \),

\[
wp(S,S',P)\upharpoonright U \cong wp(S,wp(S,P))\upharpoonright U
\]

\[
\cong wp(S,P)\upharpoonright C(S,U)
\]

\[
\cong P\upharpoonright C(S,C(S,U))
\]
According to Property 5.7 (p. 27) \( C(S,C(S,U)) = C(S,U) \), consequently
\[
wp(S;S,P)U \equiv P|C(S,U) \\
\equiv wp(S,P)U.
\]
(End of proof.)

PROPERTY 6.8. If \( A \) denotes an arbitrary associon, and \( S \) the noncascading closure statement \( x: \exists(x): T(x) \), then
\[
wp(S,A) \equiv (\exists(x): \exists(x): A \in T(x)).
\]
PROOF. Consequence of Property 5.10 (p. 29).
(End of proof.)

EXAMPLE 6.1. If \( S \) denotes the closure statement
\[
x: [v,x] \Rightarrow (w,x),
\]
then, for any name \( u \),
\[
wp(S,[v,u]) \equiv [v,u], \quad \text{(Apply Property 6.3 (p. 36),)}
\]
and
\[
wp(S,[w,u]) \equiv ([w,u] \cup [v,u]).
\]
(According to Property 6.8.
\[
wp(S,[w,u]) \equiv ([w,u] \cup \exists(x): [v,x]: (w,u) \in \{(w,x)\}) \\
\equiv ([w,u] \cup [v,u]).
\]
(End of example.)

PROPERTY 6.9. If \( S1 \) and \( S2 \) denote arbitrary closure statements, and \( P \) denotes some condition on the state, then
\[
wp(S1,wp(S2,P)) \equiv wp(S2,P)
\]
if and only if for all associons \( A \)
\[
wp(S1,A) \Rightarrow wp(S2,A).
\]
PROOF. Consequence of Property 5.12 (p. 32).
(End of proof.)

From Property 6.3 (p. 36) and Property 6.5 (p. 37) we know that the execution of a closure statement \( S \) establishes the truth of \( A \) by creating target associons of \( S \). The following theorem expresses that not too
many associations are created. More precisely: if we call a target association \( A \) of the statement \( x : E(x) \Rightarrow T(x) \) "wrong" in state \( U \) when
\[ \neg (\exists x : E(x) : A \subseteq T(x)) \],
then the theorem expresses -- besides the fact that only target associations are created -- that if prior to the execution of \( S \) there is no wrong target association present, then after the execution of \( S \) there will be no wrong target association present either.

**Theorem 6.3.** Invariance theorem for closure statements.

Let \( S \) denote the closure statement \( x : E(x) \Rightarrow T(x) \), \( A \) an arbitrary association, and \( F \) the implication
\[ A \Rightarrow (\exists x : E(x) : A \subseteq T(x)) \],
then
\[ F \Rightarrow wp(S,F) \].

**Proof.** Consequence of Property 5.11 (p. 29).
(End of proof.)

**Example 6.2.** We choose for \( S \) the closure statement
\[ x : [v,x] \Rightarrow (w,x) \].

As a first application of the invariance theorem we choose for \( A \) an association that is not a target association of \( S \). Let \( u \) denote an arbitrary name.

We choose for \( A \) the association \([v,u]\). As for all names \( x \) only the association \([w,x]\) is contained in \( T(x) \), there exists no \( x \) such that \( A \subseteq T(x) \). \( F \), consequently, is equivalent to \( \neg[v,u] \). The invariance theorem then expresses
\[ \neg[v,u] \Rightarrow \neg \neg [v,u] \Rightarrow wp(S,[v,u]) \],

i.e. \( \neg[v,u] \) is an invariant of \( S \).

We could also have chosen for \( A \) an arbitrary target association \([w,u]\).

Then there is indeed a value of \( x \) such that \( A \subseteq T(x) \), viz. the value \( u \). \( F \) then becomes
\[ [w,u] \Rightarrow E(u) \],
or
\[ [w,u] \Rightarrow [v,u] \].

As \( u \) denotes an arbitrary name, we have derived the invariance of the im-
According to Property 6.5 (p. 37), the execution of \( S \) establishes the truth of the implication

\[
(\forall x: (v, x) \Rightarrow (v, x))
\]

If (3) is satisfied to start with, it will remain satisfied, and the execution of \( S \) establishes the truth of the equivalence

\[
(\forall x: (w, x) \Leftrightarrow (v, x))
\]

(End of example.)

Suppose we have to write a program that establishes the truth of a relation \( R \). We then have to find an implication \( \hat{S} \) and a relation \( P \), such that initially \( P \) holds, \( P \Rightarrow wp(S, P) \), and \( (P \land \hat{S}) \Rightarrow R \). Often one can write \( R \) as an equivalence, that initially holds in one direction. We then try to establish, under invariance of that initially satisfied implication, the implication in the other direction as well. An instance of this we encountered in the second part of the above example.

The invariance theorem has been formulated in such a fashion that it may provide a practicable tool for proving the invariance of implications. We shall demonstrate this application of the invariance theorem in two more examples. In the above example we obtained the invariant "inverse" implication by simply reversing the implication sign in \( \hat{S} \). We have chosen the following two examples to demonstrate two cases in which more care is required.

**EXAMPLE 6.3.** This is an example in which not all of the unknowns occur in each target association format. The closure statement

\[
S \quad x, y: [a, y] \land [r, y, x] \Rightarrow (d, x)
\]

establishes the truth of the implication

\[
\hat{S} \quad (\forall x, y: ([a, y] \land [r, y, x]) \Rightarrow (d, x))
\]

If we reverse the implication sign in \( \hat{S} \), we obtain a relation that is never satisfied. Although it is an invariant of \( S \), it is of little use. We can write \( \hat{S} \) as

\[
(\forall x: (b, y) \land [r, y, x] \Rightarrow [d, x])
\]

(4)
We shall prove that the "inverse" of (4), i.e.
\[(\forall x: [d, x] \Rightarrow (\exists y: [d, y] \land [x, y, x]))\]
(5)
is kept invariant by \(S\).

We choose for \(A\) the association \((d, u)\), in which \(u\) denotes an arbitrary name. The invariance theorem then expresses the invariance of
\[(d, u) \Rightarrow (\exists x, y: [d, y] \land [x, y, x] \land (d, u) \in ((d, x)))\]
or
\[(d, u) \Rightarrow (\exists y: [d, y] \land [x, y, u])\]
As \(u\) denotes an arbitrary name, this is exactly (5). If initially (5) is satisfied, then \(S\) establishes the truth of the equivalence
\[(\forall x: [d, x] \Rightarrow (\exists y: [d, y] \land [x, y, x]))\]
(End of example.)

**EXAMPLE 6.4.** This is an example in which for some target association \(A\) there exists more than one \(x\) such that \(A \in T(x)\). Let a set \(V\) of integers, and the greater-than relation \(>\) on these integers, be given by
\[(\forall x: [v, x] \Rightarrow "x is the name of an element of \(V\)"
\land
(\forall x, y: [gr, x, y] \Rightarrow ([v, x] \land [v, y] \land x > y))\].

Suppose, furthermore, that we wish to record in associations \((d, ?, ?)\) the names of mutually distinct integers in \(V\), i.e. we wish to establish the truth of
\[(\forall x, y: [d, x, y] \Rightarrow ([gr, x, y] \lor [gr, y, x]))\]
(6)
The closure statement
\[
S \quad x, y: [gr, x, y] \Rightarrow (d, x, y), (d, y, x)
\]
establishes the truth of the implication
\[
\hat{S} \quad (d, x, y) \Rightarrow ([d, x, y] \land [d, y, x])
\]
If we simply reverse the implication sign in \(\hat{S}\), we obtain a relation that is surely not an invariant of \(S\). The relation \(\hat{S}\) is equivalent to
\[(\forall x, y: ([gr, x, y] \lor [gr, y, x]) \Rightarrow [d, x, y])\]
(7)
We shall prove that the "inverse" of (7), i.e.
\[(\forall x, y: [d, x, y] \Rightarrow ([gr, x, y] \lor [gr, y, x]))\]
(8)
is kept invariant by $S$. If (8) is satisfied to start with, e.g. because
$v[d, r, ?]$, then $S$ establishes the truth of (6).

We choose for $A$ the association $(d, u, v)$, in which $u$ and $v$ denote arbitrary names. The invariance theorem then expresses the invariance of

$$[d, u, v] = (\exists x, y: [gr, x, y]: (d, u, v) \in ((d, x, y), (d, y, x)))$$

The condition $(d, u, v) \in ((d, x, y), (d, y, x))$ has two solutions for $(x, y)$, viz. $(u, v)$ and $(v, u)$. We thus obtain

$$[d, u, v] = ([gr, u, v] \cup [gr, v, u])$$

which is exactly (8).

(End of example.)
CHAPTER 7
SOME SMALL EXAMPLES

EXAMPLE 7.1. Convex hull.

A finite collection of (at least three) points in the plane, with no
three points on the same line, is given by

\[(\forall x: \{p, x\} \equiv "x is the name of a point") \land \\
(\forall x, y, z: \{t, x, y, z\} \equiv \{\{p, x\} \land \{p, y\} \land \{p, z\} \land "xyz is a 
\text{clockwise triangle"}\})\,.

It is, furthermore, given that \(r\{h, x, y\}\) holds, and it is requested to
write a program that establishes the truth of the relation

\[R \equiv (\forall x, y: \{h, x, y\} \equiv "xy is an edge of the clockwise convex hull")\]

The condition "xy is an edge of the clockwise convex hull" is equi-
valent to "there exists a point v such that xyv is a clockwise triangle,
and there does not exist a point w such that yvw is a clockwise
triangle". Hence, we can write R as

\[(\forall x, y: \{h, x, y\} \equiv \{\{t, x, y, z\} \land \neg\{t, y, x, z\}\})\,.

Consider the program

\[S \equiv x, y: \{t, x, y, z\} \land \neg\{t, y, x, z\} \Rightarrow \{h, x, y\}\,.

We prove wp(S, R).

We split R into two conditions:

\[P \equiv (\forall x, y: \{h, x, y\} \equiv \{\{t, x, y, z\} \land \neg\{t, y, x, z\}\})\,.
\]

\[S \equiv (\forall x, y: \{t, x, y, z\} \land \neg\{t, y, x, z\} \Rightarrow \{h, x, y\}\,.

Then

\[R \equiv (P \land S)\,.

(1)
\]

From Property 6.5 (p. 37) we know

\[wp(S, \tilde{S})\,.

(2)
\]

From \(\neg\{h, x, y\}\) we conclude \(P\). From the invariance theorem (p. 39) we
conclude

\[P \Rightarrow wp(S, P)\,.

(3)\]
From (3) we conclude

\[ \text{wp}(S,b) . \]  \hspace{1cm} (4)

From (2), (4), and Property 1.2 (p. 3) we conclude \( \text{wp}(S,b \land P) \), or --with (1)-- \( \text{wp}(S,b) \).

**Example 7.2. Library administration.**

A library is a collection of books. Each book has an author, and each author has at least one book in the library. Books are either on loan or in stock. Each book on loan has a borrower. The library is given by

\[
\begin{align*}
& (\forall x,y: [b,x,y] \equiv " \ x \ is \ the \ name \ of \ a \ book \ and \ y \ is \ the \ name \ of \ the \ author \ of \ that \ book") \land \\
& (\forall x,y: [l, x, y] \equiv " \ x \ is \ the \ name \ of \ a \ book \ that \ is \ on \ loan \ and \ y \ is \ the \ name \ of \ the \ borrower \ of \ that \ book")
\end{align*}
\]

It is, furthermore, given that \( \neg [x,?] \land \neg [s,?] \) holds, and it is requested to write two programs. One program should record in associations \( (x,?) \) the names of authors of whom there is a book on loan to another author. The other program should record in associations \( (s,?) \) the names of authors of whom all books are on loan. To be more precise: the one program should establish the truth of the relation

\[ R_1 \quad (\forall x: [r,x] \equiv (\exists y,z: [b,y,x] \land [l,y,z] \land [b,?,z] \land x \neq z)) , \]

and the other program should establish the truth of

\[ R_2 \quad (\forall x: [s,x] \equiv ([b,?,x] \land (\forall y: [b,y,x] \equiv [l,?,y]))) . \]

Initially the left-hand side of \( R_1 \) implies the right-hand side. The program

\[ x, y, z: [b, y, x] \land [l, y, z] \land [b, ?, z] \land x \neq z \implies (r, x) \]

establishes also the implication in the other direction, and hence \( R_1 \).

The relation \( R_2 \) can be written as

\[ (\forall x: [s,x] \equiv ([b,?,x] \land \neg \exists y: [b,y,x] \land \neg [l,?,y])) . \]

We introduce associations \( (h,?) \), in which we record the solution set of the existentially quantified condition, i.e., we establish the truth of the relation

\[ (\forall x: [h, x] \equiv (\exists y: [b,y,x] \land \neg [l,?,y])) . \]  \hspace{1cm} (5)
This can be accomplished by the closure statement

\[ x: [b,y,x] \land \neg[1,y,?] \Rightarrow (h,x) . \]

(In associos \((h,?)\) we then have recorded the names of the authors of whom there is a book in stock.) If (5) holds, the relation \( R2 \) is equivalent to

\[ (\forall x: [s,x] \equiv ([b,?,x] \land \neg[h,x])) , \]

the truth of which can be established by the closure statement

\[ x: [b,?,x] \land \neg[h,x] \Rightarrow (s,x) . \]

As the execution of this statement does not violate the truth of (5) -- apply property 6.3 (p. 36) -- we have also established the truth of \( R2 \).

Program:

\[
\begin{align*}
\text{loc} (h,?) : \\
x, y : [b,y,x] \land \neg[1,y,?] \Rightarrow (h,x) ; \\
x : [b,?,x] \land \neg[h,x] \Rightarrow (s,x) \\
\text{col}
\end{align*}
\]

The occurrence of the universal quantifier in the right-hand side of \( R2 \) necessitated the introduction of the associos \((h,?)\). We shall encounter this theme more often.

**Example 7.3. Missionaries and cannibals.**

Three missionaries and three cannibals wish to cross a river from the left bank to the right bank. For this purpose a boat is situated at the left bank. The boat can sail from one bank to another with either one or two passengers. On either bank the missionaries will be eaten by the cannibals if there are more cannibals than missionaries. We are requested to write a program that creates the (initially absent) association \((yes)\) if and only if the crossing is possible.

We only consider the moments at which the boat is at either bank. If at those moments the situation is legitimate (in the sense that no missionary will be eaten), then the situation during the intermediate time intervals will be legitimate as well. We represent each situation by a pair of quaternary digits \((x,y)\), with \(x\) and \(y\) being respectively the number of missionaries on the right bank and the number of cannibals on the right bank.
(For the time being we do not represent the position of the boat.) The initial situation is then \((\text{zero, zero})\), and the desired final situation is \((\text{three, three})\). The quaternary digits and their successor relation can be recorded by the closure statement

\[ [] := (q, \text{zero}), (q, \text{one}), (q, \text{two}), (q, \text{three}), \]

\[ (\text{suc}, \text{zero}, \text{one}), (\text{suc}, \text{one}, \text{two}), (\text{suc}, \text{two}, \text{three}) \] .

We wish to record in assocns \((r, ?, ?, ?, ?)\) the possible transformations (between legitimate situations) that a crossing of the boat can effect, i.e. we wish to establish the truth of the relation

\[(\text{x}1, y1, x2, y2) := [r, x1, y1, x2, y2] \implies \text{"situation } (x1, y1) \text{ can by a crossing from left to right be transformed into } (x2, y2) \text{"} \] .

We then also have characterised the crossings from right to left, as \((6)\) implies

\[(\text{x}1, y1, x2, y2) := [r, x1, y1, x2, y2] \implies \text{"situation } (x2, y2) \text{ can by a crossing from right to left be transformed into } (x1, y1) \text{"} \] .

And hence we have

\[(\text{x}1, y1, x3, y3) := (3x2, y2) := [r, x1, y1, x2, y2] \land [r, x3, y3, x2, y2] \implies \text{"situation } (x1, y1) \text{ with the boat at the left can be transformed into } (x3, y3) \text{ with the boat again at the left"} \] .

Recording the reachable situations in assocns \((d, ?, ?)\) — cf. Example 5.1 (p. 33) — the program will get the following structure.

\[
\text{loc} (q, ?) := (\text{suc}, ?, ?), (d, ?, ?), (z, ?, ?, ?, ?) ::
\]

\[ [] := (q, \text{zero}), (q, \text{one}), (q, \text{two}), (q, \text{three}), \]

\[ (\text{suc}, \text{zero}, \text{one}), (\text{suc}, \text{one}, \text{two}), (\text{suc}, \text{two}, \text{three}) ; \]

"establish \((6)\)";

\[ [] := (d, \text{zero}, \text{zero}) ; \]

\[ x1, y1, x2, y2, x3, y3 := [d, x1, y1] \land [r, x1, y1, x2, y2] \land [r, x3, y3, x2, y2] := (d, x3, y3) ; \]

\[ x, y := [d, x, y] \land [r, x, y, \text{three}, \text{three}] := \text{(yes)} \]

\[ \text{col} \]

In the refinement of "establish \((6)\)" we first (temporarily) forget the fact that some pairs of quaternary digits represent illegitimate situations,
i.e. we first establish the truth of

\[(x_1,y_1,x_2,y_2) \iff \text{the (possibly illegitimate) situation } (x_1,y_1) \text{ can by a crossing from left to right be transformed into the (possibly illegitimate situation } (x_2,y_2)).\]

This can be accomplished by

\[x_1,x_2,y_1: ([x_1,x_2] \land [y_1] \implies (h_1,x_1,y_2), (h,y_1,x_2), (h,h_1,y_2,x_1), (h_1,y_1,x_2));\]

\[x_1,y_1,x_2,y_2,x_3,y_3: ([x_1,y_1,x_2,y_2] \land [x_2,y_2,x_3,y_3]) \implies (h,h_1,y_1,x_3,y_3).\]

(The associations \((h,?,?,?,?)\) record the transformations under a one-man crossing.)

Next we want to restrict (7) to legitimate situations only. Legitimate situations are those in which (on either bank) the number of missionaries equals the number of cannibals, and those in which all missionaries are on one bank. We record the legitimate situations in associations \((1,?,?)\).

"establish (6)"

\[\text{log} (h,?,?,?,?), (h,?,?,?,?), (1,?,?);\]

\[x_1,x_2,y_1: ([x_1,x_2] \land [y_1] \implies (h,x_1,y_2), (h,y_1,x_2), (h,h_1,y_2,x_1), (h_1,y_1,x_2));\]

\[x_1,y_1,x_2,y_2,x_3,y_3: ([x_1,y_1,x_2,y_2] \land [h_2,y_2,x_3,y_3]) \implies (h,h_1,y_1,x_3,y_3);\]

\[x: [y_1] \iff (1,x), (1,\text{three},x), (1,\text{zero},x);\]

\[x_1,y_1,x_2,y_2: (h_1,h_1,y_1,x_2,y_2) \land [1,1,y_1] \land [1,1,y_2] \iff (x,x_1,y_1,x_2,y_2)\]

col

Instead of applying fancy search techniques, we have programmed a tree search by (concurrently?) generating the whole tree.

EXAMPLE 7.4. Trap of a directed graph.

We first give some nomenclature on directed graphs. A walk (from \(v_0\) to \(v_n\)) in a directed graph is an alternating sequence of vertices and arcs \(v_0 \to v_1 \to v_2 \to \cdots \to v_n \to v_0 \land (n > 0)\) in which each arc \(a_{i-1} \to v_{i-1}\) \(1 \leq i \leq n\). The length of such a walk is \(n\). A nontrivial walk is a walk that contains at least one arc. A path is a walk in which all vertices are
distinct. A cycle is a nontrivial walk in which all vertices except the last
are mutually distinct, and in which the last vertex equals the first. An
acyclic graph is a graph that does not contain a cycle. A vertex \( v \) is
reachable from a vertex \( u \) if and only if there exists a path from \( u \) to
\( v \). A terminal vertex is a vertex that does not have an outgoing arc.

It is requested to write a program that determines the trap of a given
finite directed graph \( G \). A vertex belongs to the trap if and only if there
does not exist a walk from that vertex to a vertex in a cycle. (The trap of
an acyclic graph, consequently, equals the set of all vertices.)

Of the graph \( G \) is given

\[
(Vx: \text{ [v,x] } \equiv \text{ " x is the name of a vertex of } G \text{" }) \land \\
(Vx,y: \text{ [s,x,y] } \equiv \text{ " G has an arc from vertex } x \text{ to vertex } y \text{" }).
\]

Let, furthermore, be given that \( \neg [t,?] \) holds, and let it be requested to
establish the truth of the relation

\[
(Vx: [t,x] \equiv \text{ "vertex } x \text{ in the trap of } G \text{" }).
\]

We first establish the truth of

\[
(Vx,y: [w,x,y] \equiv \text{ "there exists a nontrivial walk from vertex } x \text{ to vertex } y \text{" }).
\]

(Which is nothing else than the generation of the transitive closure of the
relation "connected by an arc", vide Example 4.2 (p. 19).) As we then also have

\[
(Vx: [w,x] \equiv \text{ "vertex } x \text{ in a cycle" }),
\]

the relation \( R \) can be written as

\[
(Vx: [t,x] \equiv \{[v,x] \land \neg \exists y: [w,x,y] \land [w,y,y]\}).
\]

We introduce associations \( [h,?] \) in which we record the solution set of the
existentially quantified condition, i.e. we establish the truth of the relation

\[
(Vx: [h,x] \equiv \exists y: [w,x,y] \land [w,y,y])).
\]

(The associations \( [h,?] \) record the names of the vertices for which there
exists a walk to a vertex in a cycle.) The relation \( R \) is then equivalent to

\[
(Vx: [t,x] \equiv \{[v,x] \land \neg [h,x]\}).
\]
Program:

\[
\begin{align*}
\text{log} & \ (w,?,?) \ (h,?) : \\
& \quad x,y : \ [x,x,y] \implies (w,x,y) ; \\
& \quad x,y,z : \ [x,x,y] \land [w,y,z] \implies (w,x,z) ; \\
& \quad x,y : \ [w,x,y] \land [w,y,y] \implies (h,x) ; \\
& \quad x : \ [v,x] \land \neg [h,x] \implies (t,x) .
\end{align*}
\]

col

EXAMPLE 7.5. Kernel of a directed graph.

The kernel of a directed graph is defined as the set of all vertices from which no terminal vertex can be reached. (The kernel and the trap are, consequently, disjoint.)

It is requested to write a program that determines the kernel of a finite directed graph \( G \), given as in Example 7.4 (p. 47), while initially \( \neg [k,?] \) holds. The program should do so by establishing the truth of the relation

\[ (\forall x : [k,x] \equiv \text{"vertex } x \text{ in the kernel of } G \) , \]

which is equivalent to

\[ (\forall x : [k,x] \equiv ([v,x] \land \neg \text{"from vertex } x \text{ a terminal vertex can be reached") } . \]

If the truth of

\[ (\forall x : [h,x] \equiv \text{"from vertex } x \text{ a terminal vertex can be reached") } \]

has been established, then \( R \) can be written as

\[ (\forall x : [k,x] \equiv ([v,x] \land \neg [h,x])) . \]

Program:

\[
\begin{align*}
\text{log} & \ (h,?) : \\
& \quad x : \ [v,x] \land \neg [h,x] \implies (h,x) ; \\
& \quad x,y : \ [x,x,y] \land \neg [x,y] \implies (h,x) ; \\
& \quad x : \ [v,x] \land \neg [h,x] \implies (k,x) . 
\end{align*}
\]

col
EXAMPLE 7.6. Determining whether a directed graph is acyclic.

Let a finite directed graph \( G \) be given as in Example 7.4 (p. 47). Let, furthermore, be given that \( \neg[c] \) holds, and let it be requested to write a program that establishes the truth of the relation \( R \ [c] \iff \ "G \ contains \ a \ cycle" \).

After establishing the truth of
\[
(\forall x,y: [w,x,y] \iff \ "there \ exists \ a \ nontrivial \ walk \\
from \ vertex \ x \ to \ vertex \ y") ,
\]
(cf. Example 7.4 (p. 47)) the relation \( R \) is equivalent to
\[
[c] \iff (\exists x: [w,x,x]) .
\]

Program:

\[
\text{loop } (w,?,?):
\]
\[
x,y: [x,x,y] \leftarrow (w,x,y);
\]
\[
x,y,z: [w,x,y] \land [w,y,z] \leftarrow (w,x,z);
\]
\[
x: [w,x,x] \leftarrow [c] \]
\[
\text{col}
\]

EXAMPLE 7.7. Determining whether an undirected graph is acyclic.

We first give some nomenclature on undirected graphs. **Adjacent vertices** are vertices that are joined by an edge. A **loop** is an edge joining a vertex to itself. If two vertices are joined by more than one edge, then these edges are called **multiple edges**. A **walk** and a **path** are defined as for directed graphs. An undirected graph is **connected** if and only if every two vertices are joined by a path. A **cycle** in an undirected graph without loops or multiple edges is a walk of at least three edges in which all vertices except the last are mutually distinct, and in which the last vertex equals the first.

A finite undirected graph \( G \), without loops or multiple edges, is given by
\[
(\forall x: [v,x] \iff \ "x \ is \ the \ name \ of \ a \ vertex \ of \ G") \land
\]
\[
(\forall x,y: [x,y,x,y] \iff \ "vertices \ x \ and \ y \ are \ adjacent") .
\]

(\text{Every \ edge \ gives \ rise \ to \ the \ presence \ of \ two \ associations \ (x,?,?) \ .}) \ Let, furthermore, be given that \( \neg[c] \) holds, and let it be requested to write
a program that establishes the truth of the relation

\[ [c] \equiv \text{"G contains a cycle"}. \]

In order to find a cycle in \( G \), we look for a "proper triangle of vertices", i.e., three mutually distinct vertices \( x \), \( y \), and \( z \), such that there exists a walk between \( x \) and \( y \) (not through \( z \)), a walk between \( y \) and \( z \) (not through \( x \)), and a walk between \( z \) and \( x \) (not through \( y \)). We, therefore, establish the truth of

\[ (\forall x,y,z: [x,y,z] \equiv \text{there exists a walk between vertex } y \text{ and vertex } z \text{ (} y \neq z \text{) that does not contain vertex } x). \]

The relation \( R \) is then equivalent to

\[ [c] \equiv (\exists x,y,z: [w,z,x] \land [w,x,y] \land [w,y,z]) \land [w,y,z,x] \].

Program:

\[
\begin{align*}
\text{let} & \quad (w,z,x,y) := \\text{true}; \\
\text{x,y,z} & \quad : (a,x,y) \land (v,z) \land x \neq z \land y \neq z \implies (w,x,y,z); \\
\text{x,y,z} & \quad : (w,u,y,z) \land x \neq z \implies (w,u,x,z); \\
\text{x,y,z} & \quad : (w,z,x,y) \land (w,x,y,z) \land (w,y,z,x) \implies (c)
\end{align*}
\]

\text{col}

\text{Example 7.8. \textit{Safely connected vertices}}.

Let \( G \) be a finite undirected graph, possibly containing loops and multiple edges. Two paths in \( G \) that have no edge in common are called \textit{edge-disjoint}. Two vertices are \textit{safely connected} if and only if they are connected by (at least) two edge-disjoint paths.

The graph \( G \) is given by

\[
(\forall x: [v,x] \equiv \text{"} x \text{ is the name of a vertex of } G \text{"}) \land \\
(\forall x: [e,x] \equiv \text{"} x \text{ is the name of an edge of } G \text{"}) \land \\
(\forall x,y,z: [j,x,y,z] \equiv \text{"} edge } x \text{ joins vertices } y \text{ and } z \text{"}).
\]

Let, furthermore, be given that \( [c,?,?] \) holds, and let it be requested to write a program that establishes the truth of the relation

\[ [v,x:y: [c,x,y] \equiv \text{"} vertices } x \text{ and } y \text{ are safely connected"}. \]

One property of safely connected vertices is that if vertices \( u \) and \( v \) are safely connected, then any two vertices on the edge-disjoint paths
between $u$ and $v$ are safely connected as well.

Another property is that safe connectedness is an equivalence relation. It is easy to see that the relation is symmetric and reflexive (for reflexivity: choose twice the path consisting of the vertex only). The transitivity requires an additional exploration.

Safely connected vertices are those vertices that cannot be separated by the removal of one edge from the graph. (It is obvious that two safely connected vertices cannot be separated by the removal of one edge; the inverse assertion, i.e. that two vertices that cannot be separated by the removal of one edge are safely connected, can be proved by induction on the length of their shortest connecting path.) Suppose now that the vertices $x$ and $y$ are safely connected, and that the vertices $y$ and $z$ are safely connected. If we remove one edge from the graph, then $x$ and $y$ remain connected, and $y$ and $z$ remain connected, and therefore, $x$ and $z$ remain connected too. We conclude that $x$ and $z$ are also safely connected.

From the above two properties we conclude that we can find all safely connected vertices by first looking for adjacent vertices that are safely connected, and then generating the transitive (and reflexive) closure of that relation.

We first establish the truth of

$$\forall x,y,z: [x,y,z] \iff \text{there exists a nontrivial walk between vertex } y \text{ and vertex } z \text{ that does not contain edge } x \text{.}$$

Next we establish the truth of

$$\forall x,y: [x,y] \iff \text{the adjacent vertices } x \text{ and } y \text{ are safely connected}\text{.}$$

after which the truth of $R$ is established by extending the relation of (8) to its transitive and reflexive closure.

Program:
\textit{loc} (w,x,y,z):
\begin{align*}
x,y,z,u & \ni [j,x,y,z] \land [s,u] \land x = u \Rightarrow (w,u,y,z); \\
x,y,z,u & \ni [w,x,y,z] \land [w,x,z,u] \Rightarrow (w,x,y,u); \\
x,y,z & \ni [j,w,y,z] \land [w,x,y,z] \Rightarrow (c,n,z); \\
x,y,z & \ni [c,x,y] \land [c,y,z] \Rightarrow (c,x,z); \\
x & \ni [v,x] \Rightarrow (c,x,z)
\end{align*}

\textit{col}

\textbf{Example 7.9. Minimal transition pair.}

If a directed graph has an arc from vertex $w$ to vertex $v$, then $w$ is called a predecessor of $v$, and $v$ is called a successor of $u$. Let $K$ be some vertex of a directed graph $G$. The minimal transition pair with $K$ as initial vertex, notation "$(M(K), N(K))$", is the unique smallest pair -- i.e. the pair for which the sum of the cardinalities of the components is minimal -- $(M,N)$ such that

1) $M$ and $N$ are sets of vertices of $G$,
2) $K \in M$,
3) all successors of vertices in $M$ are in $N$,
4) all predecessors of vertices in $N$ are in $M$.

Such a unique smallest pair exists, because -- if $V$ denotes the set of vertices of $G$ -- $(V,V)$ satisfies the properties 1), 2), 3), and 4), and if the pairs $(M_0,N_0)$ and $(M_1,N_1)$ satisfy them, then the pair $(M_0 \cap M_1,N_0 \cap N_1)$ satisfies them too.

Let a finite directed graph $G$ be given as in Example 7.4 (p. 47) and let $k$ be the name of the initial vertex $K$. Let, furthermore, be given that $\neg([m,k] \land [n,k])$ holds, and let it be requested to write a program that establishes the truth of the relation $R$

$$\forall x: ([m,x] \land x \in N([k]) \land \forall y: ([n,x] \land x \in N([k])) .$$

We can write the properties 1), 2), 3), and 4) as follows.

\begin{align*}
R_1 & \Rightarrow \forall x: ([m,x] \lor [n,x]) = [v,x] , \\
R_2 & \Rightarrow [m,k] , \\
R_3 & \Rightarrow \forall x, y: ([s,x,y] \land [m,x]) \Rightarrow [n,y] , \\
R_4 & \Rightarrow \forall x, y: ([s,x,y] \land [n,y]) \Rightarrow [m,x] .
\end{align*}
We are requested to write a program that establishes \( R_1 \land R_2 \land R_3 \land R_4 \) by creating as few associations \( (m,n) \) and \( (n,m) \) as possible. \( R_3 \land R_4 \) is equivalent to

\[
(\forall x,y: ([s,x,y] \land ([m,x] \lor [n,y])) \implies ([m,x] \land [n,y])) .
\]

Program:

\[
[] := (m,k);
\]

\[
x,y: [s,x,y] \land ([m,x] \lor [n,y]) \quad \Rightarrow \quad (m,x), (n,y)
\]

Upon termination \( R_1 \) (still) holds. \( R_2 \) is established by the first statement. The second statement establishes \( R_3 \land R_4 \). From the definition of \( C(s,u) \) we conclude that the minimal pair is generated.
CHAPTER 8

AN APPRECIATION OF THE REPETITIVE CONSTRUCT

In Chapter 4 we stated that it is the purpose of a computation to make equivalences true. We were, for instance, requested to establish the truth of the equivalence

\[ (\forall x: \text{odd}(x) \equiv (\text{int}(x) \land \neg \text{even}(x))) \]  

by creating assertions \( \text{odd}(?x) \) while initially the implication

\[ (\forall x: \text{odd}(x) \equiv (\text{int}(x) \land \neg \text{even}(x))) \]  

was satisfied. A closure statement

\[ x: \text{int}(x) \land \neg \text{even}(x) \implies \text{odd}(x) \]

establishes the implication

\[ (\forall x: (\text{int}(x) \land \neg \text{even}(x)) \implies \text{odd}(x)) \]

under invariance of its "inverse" implication (2), thus establishing the truth of the equivalence (1). This is the general pattern of associating computations: the equivalence to be established holds initially in one direction; under invariance of that implication the computation establishes the implication in the other direction as well, thus ensuring the desired equivalence to hold. Theorem 6.3 (p. 39) implies that the closure statement does indeed establish implications under invariance of their inverse implications.

And as such we have used the closure statement in our examples to achieve the desired equivalences. A problem arises, however, as soon as the antecedent of the implication to be established is more complicated than we can express in the left-hand side of a closure statement. In particular may we expect problems in the case that the antecedent contains a universal (or a negated existential) quantifier.

As an example we shall focus our attention on the generation of an arbitrary clique of an undirected graph.

We first give some more nomenclature on undirected graphs. A subgraph of a graph \( G \) is a graph having all its vertices and edges in \( G \). \( G \) is a supergraph of \( G' \) if and only if \( G' \) is a subgraph of \( G \). A spanning subgraph of \( G \) is a subgraph containing all the vertices of \( G \). A complete graph is a graph (without loops or multiple edges) in which all vertices are
adjacent. A clique of a graph is a complete subgraph that is not a proper subgraph of a complete subgraph.

Let an undirected graph \( G \) (possibly containing loops) be given as in Example 7.7 (p. 50). Let, furthermore, be given that \( \forall(c,y) \) holds, and let it be requested to write a program that establishes —by creating associations \( (c,?) \)— the truth of the relation

\[
\forall x: (c,x) \in [(v,x) \land (\forall y: y \neq x: [a,y] \Rightarrow [a,x,y]]).
\]

(We generate an arbitrary clique, recording the vertices of the clique in associations \( (c,?) \). As a clique is a complete graph, it can be characterised by its vertex set.)

\textbf{Remark 8.1.} The generation of an arbitrary clique is a nondeterministic affair. If we want our program to be such that it may establish any of the permissible final states—which seems a laudable goal—, then it should not surprise us that we shall need other constructs besides the (deterministic) closure statement.

\textit{(End of remark.)}

The implication in the one direction

\begin{equation}
\forall x: ([v,x] \land (\forall y: y \neq x: [a,y] \Rightarrow [a,x,y])) \Rightarrow (c,x)
\end{equation}

holds initially. \( P_1 \) expresses that no "wrong" associations are present. Pronouncing \([c,x]\) as "\( x \) in the clique", it expresses that for all \( x \) in the clique, \( x \) is the name of a vertex that is adjacent to all other vertices in the clique. Under invariance of \( P_1 \) the truth of the implication in the other direction, i.e.

\begin{equation}
(\forall x: ([v,x] \land (\forall y: y \neq x: [c,y] \Rightarrow [a,x,y])) \Rightarrow [c,x])
\end{equation}

has to be established. Relation (3) expresses that "sufficiently many" associations have been created. It expresses that all vertices \( x \) that are adjacent to all other vertices in the clique, are in the clique as well. Relation (3) is equivalent to

\begin{equation}
(\forall x: ([v,x] \land \forall (y: y \neq x: [c,y] \land [a,x,y]) \Rightarrow [c,x])
\end{equation}

We are faced with the situation that we have to establish the truth of an implication in which the antecedent contains a negated existential quantifier.
If all quantified variables of the negated existential quantifier --in (4) the "y" only-- had occurred only once in the quantified condition, then the question mark "?" would have provided a simple solution. We have met this case in Example 7.1 (Convex Hull (p. 43)), in which the implication was
\[ (\forall x, y: \{t, x, y, z\} \land \neg (\exists w: \{t, y, x, w\}) \Rightarrow \{h, x, y\} ) , \]
the truth of which could be established by the program
\[ x, y, z: \{t, x, y, z\} \land \neg \{t, y, x, ?\} \Rightarrow \{h, x, y\} . \]
(The "z" may be replaced by a question-mark as well, but that transformation is less essential.)

If some quantified variable of the negated existential quantifier occurs more than once in the quantified condition --as the "y" in (4)--, then we introduce new associations in which we record the solution set of the existentially quantified condition.

We have done so, for instance, in Example 7.2 (Library Administration (p. 44)), in which the (second) implication to be established was
\[ (\forall x: \{h, y, x\} \land \neg (\exists y: \{h, y, x\} \land \neg \{y, y, y\}) \Rightarrow \{s, x\} ) . \]  \(5\)

We introduced new associations \(\{h, y\}\) and established the truth of
\[ (\forall x: \{h, y\} \land \neg (\exists y: \{h, y, x\} \land \neg \{1, y, y\}) ) . \]  \(6\)

As (6) is not matched by the target associations \(\{s, ?\}\), it is an invariant of the closure statement
\[ x, y, z: \{b, y, x\} \land \neg \{b, x, y\} :\{s, x\} . \]
The execution of this statement, consequently, establishes the truth of (5).

We call this technique "recording the quantified condition". In the case of the arbitrary clique, this technique gives rise to the introduction of the extra invariant
\[ P2 \]
\[ (\forall x: \{h, x\} \land (\exists y: y \neq \{0, y\} \land \neg \{0, x, y\}) ) . \]
If \(P2\) is satisfied, \(P1\) can be written as
\[ P1' \]
\[ (\forall x: \{c, x\} \Rightarrow (\{v, x\} \land \neg \{h, x\}) ) , \]
and (4) can be written as
\[ (\forall x: (\{v, x\} \land \neg \{h, x\}) \Rightarrow \{c, x\} ) . \]  \(7\)
We are requested to establish the truth of (7) under invariance of 
P1' A P2.

So far we have followed the same pattern as in the example of the Library Administration. There is, however, one important difference: on account of the match of the target association (c,?) and P2, the creation of target associations can violate the truth of P2. As this match is a positive one, the creation of associations (c,?) will in general require new associations (h,?) to be created. According to P1', the presence of an association (h,x) prohibits the association (c,x) to be created. We, therefore, have to reestablish P2 -- i.e. create the missing associations (h,?) -- after every single creation of a target association (c,?). Consequently, the program should (besides the initialization) be a repetition of

"create for one arbitrary solution of
x: [v,x] & ¬[h,x] & ¬[c,x] the association (c,x) " ;

"establish P2."

As ¬[c,?] & ¬[h,?] holds initially, the invariant P1' A P2 is satisfied to start with, and the initialization of the repetition is empty.

Program (8) should be repeated until the solution set of the equation
x: [v,x] & ¬[h,x] & ¬[c,x]
is empty, thus ensuring (7), the implication that should be established, to hold.

The nonemptiness of some solution set is the criterion we introduce as "guarding boolean expression" of the repetitive construct. The way in which guarded commands constitute a repetitive construct is the same as in [8]:

<repetitive construct> ::= do <guarded command>

[[<guarded command>]] end

<guarded command> ::= <guard> + <guarded list>

<guard> ::= <equation>

<guarded list> ::= <statement list>

The repetitive construct terminates in a state in which the solution sets of all guards are empty. When initially, or upon the completed execution of a selected guarded list, the solution sets of one or more guards are nonempty, one of these guards is selected, and its guarded list is executed.
In program (8) we wish to create for one (arbitrary) solution \( x \) of the (nonempty) guarding equation the associ\( (c,x) \). In order to be able to achieve this, we postulate -- in analogy to [11] -- that the selection of a guarded list for execution, has as a side effect that an arbitrary element of the -- then nonempty -- solution set of the guarding equation is assigned to its unknowns. These unknowns -- that do now possess values -- can be used as "initialized constants" in the guarded list.

**Remark 8.2.** In the indented paragraphs on pages 16 and 20 the word "constant" should be read as "constant that is not an initialized constant". According to the indented paragraph on page 20 the following program is not allowed.

\[
\begin{align*}
\text{do } & x, z, \Gamma [x, y] \land \neg [z, x] \Rightarrow \\
& [x, y, z] \land \neg [x, y] \Rightarrow (z, x) \\
\text{od}
\end{align*}
\]

Allowing it would bring the very problems of Chapter 4 back. (Take, e.g., \((z, (r, a, b), (r, b, a), (a, z, x)), (c, z)\) as an initial state.)

(End of remark.)

The program that records an arbitrary clique now becomes (cf. program (8))

\[
\begin{align*}
\text{let } & (h, x) : \\
\text{do } & x, z \Rightarrow \Gamma [x, y] \land \neg [z, x] \Rightarrow \\
& [x, y, z] : \Rightarrow (c, z) ; \\
& x, y, z \Rightarrow x \not= y \land \neg [y, z] \land \neg [z, x, y] \Rightarrow (h, x) \\
\text{od}
\end{align*}
\]

Only permissible final states are possible, and each permissible final state is possible. As per execution of the guarded list only one association \((c, x)\), viz. \((c, z)\), is created, we can simplify its second statement -- "establish \( y \)" -- into

\[
\begin{align*}
& x, z \not= x \land \neg [z, x, y] \Rightarrow (h, x) \\
\end{align*}
\]

The program terminates, as per execution of the guarded list the integral expression

\[
(\forall x: [x, x] \land \neg [x, x])
\]

decreases. The fact that this expression is nonnegative and finite, guaran-
tions in a finite number of steps. We call the expression
\( \forall x: [v,x] \land \neg [c,x] \) a variant function of the repetitive construct.

According to the invariant \( P_2 \) the associons \( (h,?) \) record those
vertices in the complement of the clique, for which there is a vertex in
the clique to which they are not adjacent. We can simplify the program by
recording in the associons \( (h,?) \) the vertices that are not candidates for
admittance to the clique, either because they are already recorded to be in
the clique, or because there is a nonadjacent vertex in the clique, in the
formula:
\[
P_2' \quad \forall x: [h,x] \land \neg [c,x] \lor ([v,x] \land \exists y: y \neq x: [c,y] \land \neg [a,x,y]))
\]
As for all \( x \)
\[
([v,x] \land [c,x] \land \neg [a,x,x]) \Rightarrow [c,x],
\]
\( P_2' \) is equivalent to
\[
\forall x: [h,x] \lor ([c,x] \lor ([v,x] \land \exists y: [c,y] \land \neg [a,x,y]))\)
Thus we obtain the program

\[
\text{loc} (h,?):
\]
\[
do z: [v,z] \land \neg [h,z] +
\]
\[
[ ] := (c,z), (h,z);
\]
\[
x: [v,x] \land \neg [a,x,z] \Rightarrow (h,x)
\]
\[
\text{od}
\]
\[
\text{col}.
\]
(The correctness of this program is formally proved in Example 9.1 (p. 73).)

Remark 8.3. If it is given that the graph \( G \) does not contain loops, then
\( \neg [v,z] \) implies \( \neg [a,z,z] \) -- the program may be changed into

\[
\text{loc} (h,?):
\]
\[
do z: [v,z] \land \neg [h,z] +
\]
\[
[ ] := (c,z);
\]
\[
x: [v,x] \land \neg [a,x,z] := (h,x)
\]
\[
\text{od}
\]
\[
\text{col}.
\]
(End of remark.)
We shall now treat two examples that can be expressed as the generation of an arbitrary clique. They, consequently, result in analogous programs.

**Example 8.1. Recording of an arbitrary unilateral component of a directed graph.** We first give some more nomenclature on directed graphs. A directed graph is strongly connected, or strong, if and only if every two vertices are mutually reachable. A directed graph is unilaterally connected, or unilateral, if and only if for any two vertices at least one is reachable from the other. A strong component of a directed graph is a strong subgraph that is not a proper subgraph of a strong subgraph. A unilateral component is a unilateral subgraph that is not a proper subgraph of a unilateral subgraph. Each vertex is in exactly one strong component and in at least one unilateral component.

Let a finite directed graph \( G \) be given as in Example 7.4 (p. 47). Let, furthermore, be given that \( \forall i \in \{1, \ldots, p\} \) holds, and let it be requested to record \(-\ln \) associations \((u, v)\) — the vertices of an arbitrary unilateral component of \( G \).

We define a new graph \( G' \) with the same vertex set as \( G \). The graph \( G' \) is undirected. Two vertices \( u \) and \( v \) are adjacent in \( G' \) if and only if there exists in \( G \) a nontrivial walk from \( u \) to \( v \) or vice versa. (The graph \( G' \) may, consequently, contain loops.) The cliques of \( G' \) are then precisely the unilateral components of \( G \).

We generate the edges of \( G' \) by first establishing the truth of

\[
(\forall x, y: \left[ x, y \right] \ni \forall z \ni (y, z) \Rightarrow \left[ y, z \right] \text{ "there exists a nontrivial walk from vertex } x \text{ to vertex } y \text{ in } G' \"},
\]

as we have done in Example 7.4 (p. 47). We then establish the truth of

\[
(\forall x, y: \left[ x, y \right] \ni \forall z \ni (y, z) \Rightarrow \left[ y, z \right] \text{ "vertices } x \text{ and } y \text{ are adjacent in } G' \"},
\]

which is equivalent to

\[
(\forall x, y: \left[ x, y \right] \ni \forall z \ni (y, z) \Rightarrow \left[ y, z \right]) .
\]

The recording of an arbitrary unilateral component of \( G \) is now the same as the recording of an arbitrary clique of \( G' \).
Program:

\[ \text{loc} \{w, y, z\}, \{x, y, z\} : \]
\[ x, y, z : [x, y, z] := (w, y, z); \]
\[ \text{do} \{w, x, y\} \land [w, y, z] := (w, x, z); \]
\[ x, y : [x, y] := (x, y), (y, x); \]
\[ \text{endloc}(h, z); \]
\[ \text{do} \{v, x\} \land \neg [h, z] + \]
\[ [] \Rightarrow (v, x), (h, z); \]
\[ x : [v, x] \land \neg [a, x, z] \Rightarrow (h, x) \]
\[ \text{enddo} \]
\[ \text{end} \]

(End of example.)

**EXAMPLE 8.2.** One characteristic elements per equivalence class. Let a finite set \( V \) and an equivalence relation \( E \) on the elements of \( V \) be given by

\[ (\forall x : [v, x] \in " x \) is the name of an element of \( V \)"
\[ (\forall x, y : [v, x] \land [v, y] \land E(x, y)) \]

We are requested to write a program that selects exactly one element out of every equivalence class of \( E \), i.e., we are requested to establish (if initially \( \forall [c, z] \) holds) the truth of

\[ (\forall x : [c, x] \in ([v, x] \land (\forall y : y \neq x : [c, y] \Rightarrow \neg [c, x, z])) \]

We can associate with \( V \) and \( E \) an undirected graph \( G \). Each element of \( V \) corresponds (in a one-to-one fashion) to a vertex of \( G \). Two vertices of \( G \) are adjacent if and only if their corresponding elements in \( V \) do not satisfy the relation \( E \). (The graph \( G \), consequently, does not contain loops.) The recording of one characteristic element per equivalence class of \( E \) is now the same as the recording of an arbitrary clique of \( G \).

Program:

\[ \text{loc} \{h, z\}; \]
\[ \text{do} \{v, z\} \land \neg [h, z] \Rightarrow \]
\[ [] \Rightarrow (c, z); \]
\[ x : [e, x, z] := (h, x) \]
\[ \text{enddo} \]

(End of example.)
In the example of the arbitrary clique we have demonstrated the need of repetition. We shall now treat an example in which the choice of the invariant forces us to introduce associates that are local to the guarded list. We shall design a program for the recording of balanced vertices in an acyclic directed graph.

For each vertex \( v \) of an acyclic directed graph there exists a (not necessarily unique) longest walk starting in \( v \). We shall write \( L(v) \) for the length of such a walk. Two vertices \( u \) and \( v \) with \( L(u) = L(v) \) are called balanced.

Let a finite acyclic directed graph \( G \) be given as in Example 7.4 (p. 47). Let, furthermore, be given that \( C[v,v,7] \) holds, and let it be requested to write a program that establishes the truth of

\[
\forall x, y : y \in [v, x, v] \Rightarrow \text{"vertices } x \text{ and } y \text{ are balanced"}.
\]

If a finite (nonempty) directed graph \( D \) is acyclic, then it contains at least one terminal vertex. (For assume that all vertices of \( D \) have at least one successor. Then we can make an arbitrary long walk in \( D \). As the number of vertices in \( D \) is finite, a long enough walk must contain some vertex more than once. But then \( D \) contains a cycle.) Obviously, all terminal vertices are mutually balanced, and no terminal vertex is balanced with a nonterminal one. We can, consequently, record the balanced vertices of our graph \( G \) by first recording all terminal vertices to be mutually balanced, and then recording the balanced vertices in the remainder of \( G \), which is a smaller acyclic directed graph. (A maximal path in the remainder always leads to a vertex with a terminal successor. The balanced vertices in the remainder are, consequently, exactly the nonterminal balanced vertices of \( G \).)

We thus have the following invariant (writing \( V(G) \) for the vertex set of a graph \( G \)).

\[
P : \exists \subseteq V(G) \wedge ^{\text{"all balanced vertices in } \exists \text{ have been recorded"}} \wedge ^{\forall u, v : u, v \in V(G) : (u \in \exists \wedge v \notin \exists) \Rightarrow \text{"} u \text{ and } v \text{ not balanced"}}
\]
The choice of \( P \) leads to a program of the following structure.

\[ B := \emptyset; \]

\[ \text{do } B \leftarrow V(G) \quad \text{"record all terminal vertices in } G \setminus B \text{ to be mutually balanced, and extend } B \text{ with these vertices"} \]

\[ \text{od} \]

As the graph \( G \setminus B \) is acyclic, it must contain a terminal vertex. Hence the cardinality of \( V(G) \setminus B \) decreases per execution of the guarded list.

As the balanced vertices are recorded in associons \( (b, \bar{y}, \bar{?}) \), and each vertex is at least balanced with itself, we have:

\[
(\forall x: [b, x, ?] \equiv \text{"vertex } x \text{ in } B \text{"}) \quad \text{and} \\
(\forall x: ([v, x] \land \neg [b, x, ?]) \equiv \text{"vertex } x \text{ in } V(G) \setminus B \text{"}).
\]

Our program will thus have the following structure.

\[ \text{do } z: [v, x] \land \neg [b, z, ?] \equiv \text{"create } (b, x, y) \text{ for all terminal vertices } x \text{ and } y \text{ in the graph } G \setminus B \" \]

\[ \text{od} \]

The terminal vertices in the graph \( G \setminus B \) constitute a subset of \( V(G) \setminus B \), viz. those vertices for which all successors in \( G \) are in \( B \). We record these vertices in associons \( (c, ?) \), i.e. we establish the truth of

\[
(\forall x: [c, x] \equiv ([v, x] \land \neg [b, x, ?] \land (\forall y: [s, x, y] \equiv [b, y, ?]))). \quad (9)
\]

This equivalence can also be written as

\[
(\forall x: [c, x] \equiv ([v, x] \land \neg [b, x, ?] \land (\forall y: [s, x, y] \land \neg [b, y, ?])))
\]

we record the quantified condition in associons \( (h, ?) \), i.e. we establish the truth of

\[
(\forall x: [h, x] \equiv (\forall y: [s, x, y] \land \neg [b, y, ?])). \quad (10)
\]

Assuming \( \neg [h, ?] \) to hold initially, the truth of (10) can be established by the closure statement

\[ x, y: [s, x, y] \land \neg [b, y, ?] \Rightarrow (h, x), \]

after which (9) can be made true by the closure statement (assuming \( \neg [c, ?] \) to hold initially).
x: \{v, x\} \land \neg\{b, x, ?\} \land \neg\{h, x\} \Rightarrow \{c, x\}.

Program:
\begin{align*}
\textbf{do} & \ x: [v, x] \land \neg\{b, x, ?\} + \\
& \quad \log (h, ?), (c, ?); \\
& \quad x, y: [s, x, y] \land \neg\{b, y, ?\} \Rightarrow (h, x); \\
& \quad x: [v, x] \land \neg\{b, x, ?\} \land \neg\{h, x\} \Rightarrow (c, x); \\
& \quad x, y: [c, x] \land \neg\{c, y\} \Rightarrow (b, x, y)
\end{align*}
\textbf{od}

The integer expression \( Nz: [v, x] \land \neg\{b, x, ?\} \) is a variant function of the repetition.

As the repetitive construct contains only one guard, and as its initialized constant \( x \) does not occur in the guarded list, the above program is deterministic. In Chapter 11 we shall show that it is also possible to write a nonrepetitive version.

In (9) and (10) the target associations \( (b, ?, ?) \) negatively match the right-hand side. Given this choice of (9) and (10), we had to declare the associations \( (c, ?) \) and \( (h, ?) \) local to the guarded list. Logically, the introduction of local associations is not necessary. The associations \( (c, ?) \) were introduced to avoid "negative matching" in the statement catering for the creation of the associations \( (b, ?, ?) \). Maintaining as an invariant
\[(Nz: [b, x] \equiv [b, x, ?])\]
their introduction can be avoided.

The relation "balanced" is an equivalence relation. Per execution of the guarded list one equivalence class is recorded. If we extend the associations \( (h, ?) \) with an extra member, representing a characteristic element of the "current" equivalence class, then the (resulting) associations \( (h, ?, ?) \) can be declared global to the repetitive construct. In order to be able to determine this characteristic element, we maintain as an invariant
\[(Nz: [b, x] \equiv ([bb, x] \lor \{v, x\} \land \{vy: [s, x, y] \equiv [bb, y]\}))\).

(In words: \([b, x]\) holds if and only if vertex \( x \) is in \( B \) or vertex \( x \) is a terminal vertex of \( G \setminus B \).)
loc (bb,?, (bt,?), (h,?,?) :-
   x: [v,x] ∧ ¬[x,x,?] ⇒ (bt,x); do z: [bt,z] ∧ ¬[bb,z] ⇒
   x,y: [bt,x] ∧ ¬[bb,x] ∧ [bt,y] ∧ ¬[bb,y] ⇒ (h,x,y);
   x: [bt,x] := (bb,x);
   x,y: [s,x,y] ∧ ¬[bb,y] ⇒ (h,x,s);
   x: [v,x] ∧ ¬[h,x,x] ⇒ (bt,x)
   od
   col

For strategic reasons this second program is inferior to the first.
The extra information that is kept -- the (h,x,y) with y ∈ S -- is left unused.
CHAPTER 9

FORMAL DEFINITION OF THE REPEATED CONSTRUCT

The syntax of the repetitive construct has been defined in the preceding chapter. Its semantics are defined as follows.

Let $S$ denote the repetitive construct

$$E_0 \rightarrow S_0 \hspace{1em} \ldots \hspace{1em} E_{n-1} \rightarrow S_{n-1} \hspace{1em} E_n \,.$$  

Let $BB$ denote the predicate $(3 i: 0 \leq i < n; \{ n \, E_i \} > 0)$, and let $E_i$ denote the equation $E_i$, of which the list of unknowns has been dropped. ($E_i$ expresses the relation which the initialized constants satisfy.)

Then, for any predicate $R$, by definition

$$wp(S, R) \equiv (\exists k: k \geq 0; H_k(R)),$$

in which the predicates $H_k(R)$ are defined by

$$H_0(R) \equiv R \land \neg BB,$$

$$H_{k+1}(R) \equiv H_k(R) \lor (BB \land \forall i: 0 \leq i < n; E_i \implies wp(S_i, H_k(R))).$$

The predicate $H_k(R)$ expresses the condition that the construct terminates after at most $k$ executions of a guarded list, in a state satisfying $R$. The definition of $wp(S, R)$ expresses that there must exist a nonnegative integer $k$, such that the construct terminates after at most $k$ executions of a guarded list, in a state satisfying $R$.

The following theorem expresses that any closure statement can be written as a repetitive construct, in which each guarded list consist of one closure statement with $[]$ as its left-hand side and one target associn (format) as its right-hand side. It shows that an implementation of the closure statement may create the missing associn of the closure one by one, locking (in an arbitrary order) for solutions of the left-hand side for which a target associn is absent.

**Theorem 9.1.** Let $S$ denote the closure statement

$$x: E(x) := A_0(x), \ldots, A_{n-1}(x),$$
Let $S_i$ (0 ≤ i ≤ k) denote the closure statement

$$
i := A_i(x),$$

and let \( S' \) denote the repetitive construct

$$\text{do } x : E(x) \land \neg A_i(x) \Rightarrow S_0
\quad \mathbb{E} \quad \ldots
\quad \mathbb{E} \quad x : E(x) \land \neg A_{i-1}(x) \Rightarrow S_{i-1}
\quad \text{od}.$$

Then, for all \( R \), \( \wp(S, R) \subseteq \wp(S', R) \).

**Proof.** Let NC denote

$$\langle N A : \wp(S, A) \land \neg A \rangle,$$

i.e. the number of assoc. A satisfying \( \wp(S, A) \land \neg A \). According to Theorem 6.1 (p. 35) NC is finite for all states. Obviously, we also have NC ≥ 0.

We first prove the following properties.

1) NC = 0 if and only if \( \text{wp} \).

2) \( E(x) \) implies for all R

$$\wp(S_i, \wp(S, R)) \subseteq \wp(S, R),$$

3) \( E(x) \land \neg A_i(x) \) implies for all k (k ≥ 0)

$$\wp(S, \text{NC = k}) = \{ \text{NC = k+1} \}.$$

Property 2) expresses that the (permitted) execution of the statement \( S_i \) leaves the condition \( \wp(S, R) \) invariant. Property 3) expresses that the (permitted) execution of \( S_i \) decreases the function NC by one.

Proof of 1):

$$\text{wp} \equiv \bigwedge (i : 0 \leq i < k : (\forall x : E(x) \land \neg A_i(x) > 0)
\quad \equiv \bigwedge (i : 0 \leq i < k : (\forall x : E(x) \land \neg A_i(x) = 0)
\quad \equiv \bigwedge (i : 0 \leq i < k : (\forall x : E(x) \Rightarrow A_i(x))) \subseteq S.$$

We prove NC = 0 if and only if \( \text{wp} \).

We first assume NC = 0 and derive \( \text{wp} \). Let i be such that 0 ≤ i < k, and let \( x \) be such that E(x) is satisfied. We prove \( A_i(x) \).

From E(x) we conclude, by applying Property 6.4 (p. 36),
\[ \text{wp}(S, R(x)) \] 

From \( \text{wp}(S, \tilde{A}) \) -- Property 6.5 (p. 37) -- and the definition of \( \tilde{S} \) we conclude

\[ \text{wp}(\tilde{S}, \tilde{A}(x)) = \tilde{A}_1(x) \] 

(2)

From (1) and (2) follows (apply Property 1.2 (p. 3))

\[ \text{wp}(S, \tilde{A}_1(x)) \] 

(3)

As \( N = 0 \) we have for all 
associations \( A \)

\[ \text{wp}(S, \tilde{A}) = \tilde{A} \]

In particular do we have

\[ \text{wp}(S, \tilde{A}_1(x)) = \tilde{A}_1(x) \]

(4)

From (3) and (4) we conclude \( \tilde{A}_1(x) \).

The fact that \( \tilde{S} \) implies \( N = 0 \) is a direct consequence of Property 6.6 (p. 37).

Proof of (2):

We assume \( R(x) \). Then (3) holds. Let \( A \) denote an arbitrary association. As \( S_1 \) is a noncascading closure statement, we may apply Property 6.9 (p. 38), yielding

\[ \text{wp}(S_1, \tilde{A}) = (\tilde{A} \lor A = \tilde{A}_1(x)) \] 

(5)

From Property 6.2 (p. 36) we know

\[ \tilde{A} = \text{wp}(S, \tilde{A}) \]

(6)

From (3) and (6) we conclude

\[ (\tilde{A} \lor A = \tilde{A}_1(x)) \rightarrow \text{wp}(S, \tilde{A}) \]

(7)

From (5) and (7) follows

\[ \text{wp}(S_1, \tilde{A}) = \text{wp}(S, \tilde{A}) \]

From Property 6.9 (p. 38) we then know

\[ \text{wp}(S_1, \text{wp}(S, R)) = \text{wp}(S, R) \]
Proof of 3):

We assume \( E(x) \land \neg A_1(x) \land x \geq 0 \). Then, on account of \( E(x) \), (3) holds. By definition,

\[
wp(S_1, NC = k) \equiv \wp(S_1, (N A \land wp(S_1, \neg A) = k)) ,
\]

or

\[
wp(S_1, NC = k) \equiv ((N A \land wp(S_1, wp(S_1, \neg A)) = k) ,
\]

or (apply Properties 1.2 (p. 3) and 6.1 (p. 36))

\[
wp(S_1, NC = k) \equiv ((N A \land wp(S_1, wp(S_1, \neg A)) = k) .
\]

As \( E(x) \) holds, we may apply Property 2) yielding

\[
wp(S_1, NC = k) \equiv ((N A \land wp(S_1, \neg wp(S_1, \neg A)) = k) .
\]

(8)

From (5) and (8) we conclude

\[
wp(S_1, NC = k) \equiv ((N A \land wp(S_1, \neg A) \land \neg A \land A \land A_1(x)) = k) ,
\]

or

\[
wp(S_1, NC = k) \equiv ((N A \land wp(S_1, \neg A) \land \neg A)
\]

\[
- (N A \land wp(S_1, \neg A) \land \neg A \land A \land A_1(x)) = k) .
\]

(9)

From (3) and \( \neg A_1(x) \) follows

\[
(N A \land wp(S_1, \neg A) \land \neg A \land A \land A_1(x)) = 1 .
\]

(10)

From (9), (10), and the definition of \( NC \) we conclude

\[
wp(S_1, NC = k) \equiv (NC = k + 1) .
\]

(End of proof of Properties 1), 2), and 3).)

Next we shall prove for arbitrary \( R \) and \( k (k \geq 0) \)

4) \( P_k(R) \equiv (wp(S, R) \land NC \leq k) \).

We then have

\[
wp(S, R) = P_{NC}(R)
\]

(11)

and

\[
P_k(R) = wp(S, R) .
\]

(12)
The combination of (11) and (12) allows us to conclude
\[ \text{wp}(S,R) \equiv (\exists k \geq 0: H_k(R)) \equiv \text{wp}(S',R) . \]

Proof of 4):
We prove 4) by mathematical induction. By definition,
\[ H_0(R) \equiv (R \wedge \neg \text{BB}) . \]
\( S \) and \( \neg \text{BB} \) are equivalent. According to Property 6.6 (p. 37) \( S \) implies \( \text{wp}(S,R) \equiv R \). From Property 1) and \( NC \geq 0 \) we conclude \( \neg \text{BB} \equiv (NC \leq 0) \).

We, consequently, have
\[ H_0(R) \equiv (\text{wp}(S,R) \wedge NC \leq 0) . \quad (13) \]

We assume as induction hypothesis
\[ H_k(R) \equiv (\text{wp}(S,R) \wedge NC \leq k) \quad (14) \]

and derive
\[ H_{k+1}(R) \equiv (\text{wp}(S,R) \wedge NC \leq k + 1) . \]

By definition,
\[ H_{k+1}(R) \equiv H_0(R) \lor (\text{BB} \wedge (\forall i: 0 \leq i < k; B_i \equiv \text{wp}(S_i, H_k(R)))) . \quad (15) \]

By substituting (14) in the universally quantified term, this term becomes
\[ (\forall i: 0 \leq i < k; B_i \equiv \text{wp}(S_i, \text{wp}(S,R) \wedge NC \leq k)) , \]

or (apply Property 1.2 (p. 3) and the definition of \( B_i \))
\[ (\forall i: 0 \leq i < k; (E(x) \wedge \neg \text{BB}(x)) \equiv (\text{wp}(S_i, \text{wp}(S,R)) \wedge \text{wp}(S_i, NC \leq k))) . \]

By applying Properties 2) and 3) this becomes
\[ (\forall i: 0 \leq i < k; B_i \equiv (\text{wp}(S,R) \wedge NC \leq k + 1)) , \]

or
\[ \text{BB} \equiv (\text{wp}(S,R) \wedge NC \leq k + 1) . \quad (16) \]

Resubstituting (16) in (15) we obtain
\[ H_{k+1}(R) \equiv (E_0(R) \lor (\text{BB} \wedge \text{wp}(S,R) \wedge NC \leq k + 1)) , \]

or (use (13))
\[ H_{k+1}(R) \equiv ((wp(S,R) \land NC = 0) \lor (BB \land wp(S,R) \land NC \leq k + 1)) \]

\[ \equiv ((wp(S,R) \land NC \leq k + 1) \land (BB \lor NC = 0)) \]

or (see Property 1)

\[ H_{k+1}(R) \equiv ((wp(S,R) \land NC \leq k + 1) \land (BB \lor \neg BB)) \]

\[ \equiv ((wp(S,R) \land NC \leq k + 1). \]

(End of proof.)

In the proof of the above theorem we have shown that the execution of a guarded list \( S \) decreases the nonnegative integer function \( NC \) under invariance of \( wp(S,R) \). The final state satisfies \( \neg BB \), or \( \bar{S} \). If the initial state satisfies \( wp(S,R) \), then the final state satisfies \( wp(S,R) \wedge \bar{S} \), and hence (Property 6.6 (p. 371)) \( R \). This is the general pattern for the construction of repetitive programs: we look for both an invariant relation \( P \), such that \( P \land \neg BB \) implies the relation to be established, and a (non-negative) integer variant function \( t \) of the state, whose value decreases per execution of a guarded list.

In order to characterize all initial states such that a mechanism will decrease the variant function, we employ the notion \( wdec \), which is defined as follows. (Although this definition differs from the one given in [6], it defines the same concept.)

Let \( S \) denote some mechanism, and \( t \) an integer function of the state, then by definition:

\[ wdec(S,t) \equiv wp(S,t < t_0) \]

(For notation \( p^\omega \), see Example 1.1 (p. 4).) The \( wdec(S,t) \) characterizes those states for which the execution of \( S \) establishes \( t < t_0 \), with \( t_0 \) being the initial value of \( t \).

The following theorem is due to Hoare [10], who called it the "Rule of Iteration."

**Theorem 9.2. Invariance theorem for repetitive constructs.**

With \( S \), \( R \), and \( BB \) as defined in the beginning of this chapter, there holds that, if for all \( i \) \((0 \leq i < n)\)

\[ [P \land R^i] \Rightarrow (wp(S_i,P) \land wdec(S_i,t) \land t \leq 0) \]
then

\[ P = wp(S, P \land \neg \neg B) \].

The above formulation has been taken from [3], which also contains a
proof of the theorem. The theorem expresses that if the (permitted) execution of any guarded list leaves the truth of \( P \) invariant and decreases the nonnegative integer function \( \tau \), then the execution of the repetitive construct, in a state satisfying \( P \), will lead to termination in a state satisfying \( P \land \neg \neg B \).

EXAMPLE 9.1. In this example we give a formal proof of the correctness of the program, presented in Chapter 8, for the recording of an arbitrary clique:

\[
\text{loc} (h, ?) := \begin{align*}
\text{do} & \ z : [v, z] \land \neg [h, z] := \\
& \quad [] \leftarrow (c, z), (h, z); \\
& \quad x : [v, x] \land \neg [a, x, z] := (h, x) \\
\text{od} \\
\text{call}
\end{align*}
\]

Let \( A(u, w) \) denote

\[
(\forall y : y \neq [c, y] \Rightarrow [a, w, y]) .
\]

Then \( R \), \( P_1 \), and \( P_2 \) can be written as follows. (Compare \( R \), \( P_1 \) and
\( P_2 \) on pages 56 and 60.)

\[
R \quad (\forall x : [c, x] \Rightarrow ([v, x] \land A(x, x)))
\]
\[
P_1 \quad (\forall x : [c, x] \Rightarrow ([v, x] \land A(x, x)))
\]
\[
P_2 \quad (\forall x : [h, x] \Rightarrow ([c, x] \lor ([v, x] \land \neg A(x, x))))
\]

Let \( S \) denote the repetitive construct of the program. We prove

\[
([c, ?] \land \neg [h, ?]) = wp(S, R) .
\]

With the following notational conventions

\[
S_1 \quad [] \leftarrow (c, []), (h, z), \\
S_2 \quad x : [v, x] \land \neg [a, x, z] := (h, x), \\
S \quad [v, z] \land \neg [h, z], \\
P_1 \quad (\forall x : [h, x] \Rightarrow ([c, x] \lor ([v, x] \land \neg A(x, x))))
\]

we first prove
(P₁ ∧ P₂ ∧ B) = wp(S₁, P₁ ∧ P₂ ∧ [c, z]) ,
   \tag{17}

and

(P₁ ∧ P₃ ∧ [c, z]) = wp(S₂, P₁ ∧ P₂) .
   \tag{18}

Then (apply Property 1.4 (p. 3) and the definition of the semicolon)

(P₁ ∧ P₂ ∧ B) = wp("S₁; S₂", P₁ ∧ P₂) .

(₃₁ is used to characterize the state at the semicolon.) We first formulate
some relations between A(z, x) and A(x, x) . For all x

(∀A(z, x) ∧ ¬(c, x)) → ¬A(x, x) ,
   \tag{19}

(∀A(x, x) ∧ ¬(c, x)) → ¬A(z, x) ,
   \tag{20}

(A(z, x) ∧ [z, z] ∧ [a, x, z]) = A(x, x) .
   \tag{21}

For all x (x = z)

([c, z] ∧ ¬(a, x, z)) → ¬A(x, x) .
   \tag{22}

Proof of (17): I:

By applying Property 6.8 (p. 38) we obtain for all x

wp(S₁, A(z, x)) = A(z, x) ,
   \tag{23}

and for all x (x = z)

wp(S₁, A(x, x)) = A(x, x) ∧ [a, x, z] .
   \tag{24}

We assume P₁ ∧ P₂ ∧ B and derive wp(S₁, P₁), wp(S₁, P₃), and

wp(S₁, [c, z]) . From P₂ ∧ B we conclude

[v, z] ∧ ¬[c, z] ∧ A(z, z) .
   \tag{25}

We write P₁ as follows.

(∀x: x = z; [c, x] = ([v, x] ∧ A(x, x)) ∧ [c, z] = ([v, z] ∧ A(z, z))

By applying Property 6.8 (p. 38), (23), and (24) we obtain

wp(S₁, P₁) = (∀x: x = z; [c, x] = ([v, x] ∧ A(z, z) ∧ [a, x, z]) ∧ [v, z] ∧ A(z, z) ,

which, on account of (25), is implied by P₁ . We write P₂ as follows.

(∀x: x = z; [b, x] = ([c, x] ∨ ([v, x] ∧ ¬A(z, z)))
∧ [b, x] = ([c, z] ∨ ([v, z] ∧ ¬A(z, z)))
By applying Property 6.6 (p. 39) and (33) we obtain

\[ \text{wp}(S_1, P_1) \equiv (\forall x \; x \neq z; \; [h, x] \equiv [(c, x) \vee ((v, x) \wedge \neg \Lambda(y, x))]), \]

which, on account of (19) and (20), is implied by \( P_2 \wedge \neg [c, z] \). We, furthermore, obtain \( \text{wp}(S_1, [c, z]) \equiv \bot \).

Proof of (18):

We assume \( P_1 \wedge P_2 \wedge [c, z] \) and derive \( \text{wp}(S_2, P_1) \) and \( \text{wp}(S_2, P_2) \). From \( P_1 \wedge [c, z] \) we conclude \([h, z]\). From Property 6.2 (p. 36) we know \( \text{wp}(S_2, P_1) \equiv P_1 \), which leaves only \( \text{wp}(S_2, P_2) \) to be proved.

By applying Property 6.6 (p. 36) we obtain for \( \text{wp}(S_2, P_2) \)

\[ \forall x: ([h, x] \vee ([v, x] \wedge \neg [a, x, z])) \equiv ([c, x] \vee ([v, x] \wedge \neg \Lambda(y, x))). \]

For \( x = z \) (26) is implied by \([h, z] \wedge [c, z]\). Suppose \( x \neq z \). From (19) and \( P_1 \) we then conclude

\[ [h, x] \equiv ([v, x] \wedge \neg \Lambda(x, z)). \]

From (22) and \([c, z]\) we conclude

\[ ([v, x] \wedge \neg [a, x, z]) \equiv ([v, x] \wedge \neg \Lambda(x, z)). \]

From \( P_2 \) we conclude

\[ [c, x] \equiv [h, x]. \]

From (21) and \([c, z]\) we conclude

\[ ([v, x] \wedge \neg \Lambda(x, z)) \equiv ([v, x] \wedge \neg \Lambda(y, z) \vee \neg [a, x, z])). \]

and, hence, (use \( P_1 \))

\[ ([v, x] \wedge \neg \Lambda(x, z)) \equiv ([h, x] \vee ([v, x] \wedge \neg [a, x, z])). \]

From (27), (28), (29), and (30) follows (26).

As the variant function \( t \) we choose \((N_k: [v, x] \wedge \neg [c, z])\). Then, obviously, \( t \equiv 0 \). Using

\[ t = (N_k: x \neq z; [v, x] \wedge \neg [c, x]) + (N_k: [v, x] \wedge \neg [c, z]), \]

we derive

\[ \text{wp}('S_1; S_2', t) \equiv \wp('S_1; S_2', t < t_0) \]

\[ \equiv (N_k: x \neq z; [v, x] \wedge \neg [c, x]) < t_0 \]

\[ \equiv (N_k: x \neq z; [v, x] \wedge \neg [c, x]) < t \]
\( \equiv \{v, z\} \land \neg \{c, z\} \),

which is implied by (25).

We conclude that \( P_1 \land P_2 \land B \) implies

\[
wp(\{S_1, S_2\}, P_1 \land P_2) \land \text{wdeq}(\{S_1, S_2\}, t) \land t \neq 0.
\]

From Theorem 2.2 (p. 72) we then know

\[
(P_1 \land P_2) \Rightarrow wp(S, P_1 \land P_2 \land \neg BB).
\]  
(31)

\( \neg BB \)

\[
(\forall x: \{v, x\} \Rightarrow \{h, x\})
\]

We assume \( \neg \{c, x\} \land \neg \{h, x\} \) and derive \( wp(S, x) \). From \( \neg \{c, x\} \land \neg \{h, x\} \) we conclude \( P_1 \land P_2 \), and, on account of (31), \( wp(S, P_1 \land P_2 \land \neg BB) \). As

\[
(P_2 \land \neg BB) \Rightarrow (\forall x: \{[v, x] \land A(x, x)\} \Rightarrow \{c, x\}),
\]

we then have (use P1 and R) \( wp(S, R) \).

(End of example.)
CHAPTER 10  

SOME EXAMPLES

EXAMPLE 10.1. Selection of distinct subsets.

A finite set \( V \) and a finite "family" \( W \) of (not necessarily mutually distinct) subsets of \( V \) are given (Distinctly named subsets of \( V \) may be otherwise equal.). We are requested to write a program that makes a selection out of the subsets in \( W \), such that for each subset \( S \) in \( W \) there is exactly one selected subset that contains the same elements as \( S \). The initial state satisfies

\[
(Vx: \{v, x\} \in \ "x \ is \ the \ name \ of \ an \ element \ of \ V \" ) \wedge \\
(Vx: \{w, x\} \in \ "x \ is \ the \ name \ of \ an \ element \ of \ W \" ) \wedge \\
(Vx, y: \{m, x, y\} \in \{w, x\} \wedge \{v, y\} \wedge "element \ y \ in \ subset \ x \" ).
\]

Let, furthermore, be given that \( \neg[C, o, r] \) holds. We are requested to write a program that establishes the truth of

\[
(R: \{c, x\} \in \{w, x\} \wedge (Vy: y \neq x; [c, y] = \"subsets \ x \ and \ y \ differ\")) .
\]

We record in assoc \( \{d, ?, ?, \} \) names of mutually distinct subsets in \( W \), or ---to be more precise--- we establish the truth of

\[
(Vx, y: \{d, x, y\} \in \{w, x\} \wedge \{v, y\} \wedge \neg[C, m, x, z] \vee \\
(3z: \{m, y, z\} \wedge \{w, x\} \wedge \neg[C, m, x, z])) .
\]  

(1)

The relation \( R \) can then be written as

\[
(Vx: \{c, x\} \in \{w, x\} \wedge (Vy: y \neq x; [c, y] = \{d, x, y\})).
\]

But that is exactly the relation that was to be established in the example of the arbitrary clique (Chapter 8). As we also have for all \( x \in \{d, x, z\} \), we can employ the program of Remark 8.3 (p. 60):

loc (h, ?, (d, ?, ?);  
"establish (1)";

do x: \{w, x\} \wedge \neg[h, x] = 
[ ] \Rightarrow (c, x);

x: \{w, x\} \wedge \neg[d, x, z] \Rightarrow (h, x)
do

col
We still have to implement "establish (1)". The closure statement
\[ S \quad x, y, z : [m, x, z] \land [w, y] \land \lnot[3, y, z] \Rightarrow (d, x, y), (d, y, z) \]
establishes the truth of
\[ S \quad \forall x, y, z : ([m, x, z] \land [w, y] \land \lnot[3, y, z]) \Rightarrow ([d, x, y] \land [d, y, z]) \]
\[ S \quad \text{can be written as} \]
\[ \forall x, y : ([m, x, z] \land [w, y] \land \lnot[3, y, z]) \Rightarrow ([d, x, y] \land [d, y, z]) \]
which is equivalent to
\[ \forall x, y : ([m, x, z] \land [w, y] \land \lnot[3, y, z]) \lor \]
\[ ([m, x, z] \land [w, y] \land \lnot[3, x, z]) \Rightarrow [d, x, y] \]
\[ (2) \]
By applying Theorem 6.2 (p. 40) we obtain that \( S \) maintains the invariance of
\[ \forall x, y : ([d, x, y] \Rightarrow ([m, x, z] \land [w, y] \land \lnot[3, y, z]) \lor \]
\[ ([m, y, z] \land [w, x] \land \lnot[3, x, z]) \]
\[ (3) \]
On account of \( \lnot[3, d, 7, 7] \) (3) is satisfied to start with. \( S \), therefore, establishes a state satisfying (2) and (3), and thus (1).

**EXAMPLE 10.2.** Adjusting equivalence classes.

Let a finite set \( V \) and an equivalence relation \( \equiv \) on the elements of \( V \) be given as in Example 8.2 (p. 62). Let, furthermore, be given that \( \lnot[3, c, 7] \land \lnot[3, c, 7, 7] \) holds. We are requested to write a program that records the equivalence classes of \( z \). It should do so by selecting one characteristic element per equivalence class (as in Example 8.2 (p. 62)), and recording for each element of \( V \) the characteristic element of its equivalence class. More precisely: it should, by creating associations \( (c, 7) \) and \( (m, 7, 7) \), establish the truth of
\[ R_1 \quad \forall x : [c, x] \Rightarrow ([v, x] \land \forall x : [c, y] \Rightarrow \lnot[e, x, y]) \]
\[ R_2 \quad (v, x) \Rightarrow ([c, x] \land [e, x, y]) \]
Program (cf. Example 8.2 (p. 62)):

```plaintext
do z : [v, z] \land \lnot[m, 7, z] =
  [1] := (c, z);
  x : [s, x, z] := (m, z, x)
do
```
REMARK 10.1. If we take for $V$ the set of vertices of a finite directed graph, and for $E$ the equivalence relation "mutually reachable", then we obtain a program for the recording of the strong components of a directed graph. For a finite directed graph that is given as in Example 7.4 (p. 47), we then obtain the following program.

\[
\text{loc} (e, v, z) ;
\text{log} (w, v, z) ;
\begin{align*}
x, y & : [x, y] \equiv (w, v, z) ; \\
x, y, z & : [x, y, z] \land [w, y, z] \equiv (w, v, z) ; \\
x, y & : [v, x, y] \land [w, y, x] \equiv (e, v, y) ; \\
x & : [v, x] \equiv (v, x, x) \\
\text{col} ; \\
d & : [v, z] \land \neg [e, v, z] \Rightarrow \\
[1] & \equiv (v, z) ; \\
x & : [e, x, z] \equiv (e, z, x) \\
\text{col} \\
\text{(End of remark.)}
\]

EXAMPLE 10.3. Arbitrary spanning tree.

A finite (directed) graph without arcs is given by

\[
(V x : \{v, x\} \equiv " x \ is \ the \ name \ of \ a \ vertex") .
\]

We are requested to write a program that generates arcs between the vertices, such that the resulting directed graph constitutes a spanning out-tree. We establish the truth of

\[
(V x, y : \{q, x, y\} \equiv " \text{the out-tree has an arc from vertex } x \text{ to vertex } y") ,
\]

while initially \(\neg [q, v, z]\) holds.

We first select an arbitrary vertex as root. Then we extend the out-tree until it spans the whole graph. In order to be able to characterize the vertices in the tree, we maintain

\[
(V x : \{h, x\} \equiv "\text{vertex } x \text{ in the tree}") ,
\]
or

\[
(V x : \{h, x\} \equiv "\ x \text{ is the name of the root" } \lor \{q, x, z\}) .
\]
We accomplish the selection of the root by means of a repetitive construct, whose guarded list is executed as long as $\gamma[h,?]$ holds, i.e. exactly once. As guard we choose
\[
x: [v, x] \land \gamma[h,?] .
\]
Upon termination we then know that
\[
(\forall x: [v, x] = [h,]) ,
\]
or
\[
(\exists x: [v, x] = [h,]) .
\]
\[
do x: [v, x] \land \gamma[h,?] + [ ] := (h, x) \ od
\]

REMARK 10.2. The above repetitive construct resembles the pseudo-repetition that occurs in some programs in [6]. There the alternative construct
\[
if B = S \land \gamma B = \text{skip} \ fi
\]
is in the case that $B = \wp(S, \gamma B)$ replaced by the pseudo-loop
\[
do B = S \ od .
\]

(End of remark.)

An arc from vertex $x$ to vertex $y$ may be laid if $x$ is in the tree, and $y$ is not.

Program:
\[
\lambda v: (h,?):
\]
\[
do x: [v, x] \land \gamma[h,?] + [ ] := (h, x) \ od;
\]
\[
do x, y: [v, x] \land [v, y] \land \gamma[h, y] + [ ] := (q, x, y), (h, y) \ od;
\]

The integer expression \((\exists x: [v, x] \land \gamma[h, x])\) is a variant function for the second repetitive construct. Only permissible final states are possible, and each permissible final state is possible.

REMARK 10.3. The generation of an arbitrary sequence, in which $[q, x, y]$ denotes $y$ to be the successor of $x$ in the sequence, is equivalent to the generation of an arbitrary spanning tree in which each vertex has at most one outgoing arc. This implies that by adding the factor $\gamma[0, h,?]$ to the guard of the second repetitive construct, we obtain a program for the generation of an arbitrary sequence.

(End of remark.)
EXAMPLE 10.4. Arbitrary order-preserving sequence.

The vertices of a finite acyclic directed graph can be ordered into a sequence, in such a way that for any arc from a vertex \( u \) to a vertex \( v \), \( u \) has a smaller ordinal number in the sequence than \( v \).

A finite acyclic directed graph \( G \) is given as in Example 7.4 (p. 47). Let, furthermore, be given that \( \neg[q,?,?] \) holds. We are requested to write a program that establishes the truth of

\[
(Vx,y: [q,x,y] \equiv \text{"vertex } y \text{ is the successor of vertex } x \text{ in the sequence"}) .
\]

We construct the sequence in the order of increasing ordinal number. A vertex may be chosen as the next element of the sequence if (it is not already in the sequence, and) all its predecessors (in \( G \)) are in the sequence. As in Example 10.3 (p. 79) we record the names of the vertices in the sequence in assoc \( s \) (h,?) , i.e. we maintain the invariant

\[
(Vx: [h,x] \equiv \{"x \text{ is the name of the first element" } v (q,?,x)\} .
\]

We record in assoc \( s \) (c,?) the name of the vertices for which all predecessors are in the sequence, i.e. we also maintain the invariant

\[
(Vx: [c,x] \equiv ([v,x] \land (Vy: [s,y,x] \Rightarrow [h,y]))) .
\]

The equivalence \( P2 \) can also be written as

\[
(Vx: [c,x] \equiv ([v,x] \land \gamma Vy: [s,y,x] \land \gamma[h,y]))) .
\]

In order to be able to (re)establish the truth of \( P2 \) in the guarded list, we record the quantified condition in assoc \( s \) (d,?) , i.e. we establish the truth of

\[
(Vx: [d,x] \equiv (Vy: [s,y,x] \land \neg[h,y])) .
\]

If (4) holds, \( P2 \) can be written as

\[
(Vx: [c,x] \equiv ([v,x] \land \neg[d,x]));
\]

As in (4) the presence condition of the global assoc \( s \) (h,?) is negated, the assoc \( s \) (d,?) have to be declared local to the guarded list.

Program (comments between braces "(" and ")"):
\[ \text{loc} (c,?) (h,?) : \]
\[ x: [v,x] \land \lnot [z,?,x] : = (c,x) (z2); \]
\[ \text{do} \ x: [c,x] \land \lnot [h,?] \rightarrow [? ] : = (h,x) \ 	ext{od} \ [p1]; \]
\[ \text{do} \ z1,z2: [h,z1] \land \lnot [q,z1,?] \land [c,z2] \land \lnot[h,z2] \rightarrow \]
\[ \text{loc} (d,?) : \]
\[ [?] : = (q,z1,z2), (h,z2) \ [p1]; \]
\[ x,y: [z,?,y] \land \lnot [h,y] : = (d,x) \ [p4]; \]
\[ x: [v,x] \land \lnot[d,x] : = (c,x) \ [p2] \]
\[ \text{col} \]
\[ \text{col} \]

The expression \( \{v,x\} \land \lnot (h,x) \) is a variant function for the second repetitive construct. Only permissible final states are possible, and each permissible final state is possible.

**REMARK 10.4.** For a graph \( G \) that does not have any arcs, we continually have

\[ \lnot [d,?] \land \{v,x\} \equiv [v,x] \] .

The above program then reduces to

\[ \text{loc} (h,?) : \]
\[ \text{do} \ x: [v,x] \land \lnot [h,?] \rightarrow [? ] : = (h,x) \ 	ext{od}; \]
\[ \text{do} \ z1,z2: [h,z1] \land \lnot [q,z1,?] \land [v,z2] \land \lnot[h,z2] \rightarrow \]
\[ [?] : = (q,z1,z2), (h,z2) \]
\[ \text{od} \]
\[ \text{col} \ . \]

This is exactly the program of Remark 10.3 (p. 80) for the generation of an arbitrary sequence.

(End of Remark.)

**EXAMPLE 10.5.** Most distant leaves of an out-tree.

A finite out-tree \( T \) is given by

\[ (\forall x: (v,x) \equiv \" x is the name of a vertex of \( T \") \land \]
\[ (\forall x,y: (x,y) \equiv \" T has an arc from vertex \( x \) to vertex \( y \") . \]

We define an integer function \( P \) on the vertices of \( T \), with \( P(x) \) being the length of the path from the root to vertex \( x \). In terms of the representation of \( T \) :
\( ([v,x] \land \neg [u,x,?]) \Rightarrow P(x) = 0 \),
\[ [s,y,x] \Rightarrow P(x) = P(y) + 1. \]

We have to design a program that records the vertices with maximal path length from the root, i.e., a program that, if initially \( \neg [d,?] \) holds, establishes the truth of
\[ \forall x : [d,x] = ([v,x] \land (\forall y : [v,y] \Rightarrow P(x) \geq P(y))) . \]

We have to record all vertices \( x \) of \( T \) for which
\( \forall y : [v,y] \Rightarrow P(x) \geq P(y) \). We shall record those vertices for a subtree \( U \), and we shall extend \( U \) until it equals \( T \). The vertices of \( U \) are recorded in associations \( (u,?) \). We maintain the invariance of
\[ P1 : (\forall x : [u,x] \Rightarrow [v,x]) . \]

We have to characterize the most distant leaves of \( U \). We do, however, not record the most distant leaves themselves, but instead of that, we record --in associations \( (i,?)-- \) those vertices of \( U \) that are not the most distant ones, as this last property is a monotonic one during the growth of \( U \), i.e., we also maintain as an invariant
\[ P2 : (\forall x : [i,x] \Rightarrow ([u,x] \land (\exists y : [u,y] \land P(x) < P(y)))) . \]

\( U \) initially contains the root only. We proceed to extend \( U \) until
\( (\forall x : [v,x] \Rightarrow [u,x]) . \)

As a variant function we choose \( (\forall x : [v,x] \land \neg [i,x]) . \)

Program:
\[
\text{loc} (u,?), (i,?):
\begin{align*}
& x : [v,x] \land \neg [u,x,?] \Rightarrow (u,x); \\
& \text{do } z : [v,z] \land \neg [u,z] \Rightarrow x : [u,x] \Rightarrow (i,x); \\
& \text{od}; \\
& x : [v,x] \land \neg [i,x] \Rightarrow (d,x)
\end{align*}
\]

Upon termination of the repetitive construct we know, from \( P1 \land \exists \), that
\[ (\forall x : [u,x] \Rightarrow [v,x]) , \]

which combined with \( P2 \), yields
\[(Yx: [i,x] \equiv ([v,x] \land (\forall y: [v,y] \land p(x) < p(y))))\,.

At the end of the program we then have for all \(x\)

\[(d,x) \equiv ([v,x] \land \forall [i,x])\]
\[= ([v,x] \land (\forall [v,x] \lor (\forall y: [v,y] \equiv p(x) \geq p(y))))\]
\[= ([v,x] \land (\forall y: [v,y] \equiv p(x) \geq p(y)))\,.

which exhibits the truth of \(K\) to be established.

**Example 10.6.** Evidential locus of two vertices in an undirected graph.

The evidential locus of two vertices \(u\) and \(v\) in a finite connected undirected graph is the set of all vertices for which the length of the shortest path to \(u\) equals the length of the shortest path to \(v\). Let \(m\) and \(n\) be the names of the vertices \(M\) and \(N\), respectively, of a finite connected undirected graph \(G\), which is given as in Example 7.7 (p. 50). We are requested to design a program that records in assertions \((1,7)\) the evidential locus of \(M\) and \(N\). For that reason we define two integer functions \(p_m\) and \(p_n\) on the vertices of \(G\), with \(p_m(x)\) and \(p_n(x)\) being the length of the shortest path from \(x\) to, respectively, \(M\) and \(N\).

It is given that initially \(\forall [1,7]\) holds. The program should establish the truth of

\[(Yx: [1,x] \equiv ([v,x] \land p_m(x) = p_n(x)))\,.

We generate two sets of vertices, called \(U_M\) and \(U_N\), around \(M\) and \(N\), respectively. Let \(\text{maxp}\) be a ghost variable with initial value zero and let its value be incremented by one per execution of the guarded list. Upon every execution of the guarded list \(U_M\) and \(U_N\) are the sets of vertices \(x\) with \(p_m(x) \leq \text{maxp}\) and \(p_n(x) \leq \text{maxp}\), respectively. The vertices \(x\) in \(U_M\) with \(p_m(x) < \text{maxp}\) are recorded in the set \(I_M\). \(U_N\) likewise contains an "interior" \(I_N\). For any path length we can then distill those vertices \(x\) for which \(p_m(x) = p_n(x)\), as they are in \(U_M \cap U_N\), but not in \(I_M \cup I_N\).

We can extend \(U_M\) and \(U_N\) until all vertices of \(G\) are in \(U_M \cup U_N\).

Recording the sets \(U_M\), \(U_N\), \(I_M\), and \(T_M\) in assertions \((um,7)\), \((un,7)\), \((im,7)\), and \((in,7)\), respectively, we maintain the following invariants:

\[p1\]
\[\forall [u,m,x] \lor [u,n,x] \equiv [v,x]\,.

P2 \quad (\forall x: [\text{in}, x] \equiv ([\text{un}, x] \land (\exists y: ([\text{un}, y] \land P_N(y) > P_M(x))
\lor ([\text{un}, y] \land P_N(y) > P_N(x))))))

P3 \quad (\forall x: [\text{in}, x] \equiv ([\text{un}, x] \land (\exists y: ([\text{un}, y] \land P_N(y) > P_M(x))
\lor ([\text{un}, y] \land P_N(y) > P_N(x))))))

P4 \quad (\forall x: [\text{in}, x] \equiv ([\text{un}, x] \land P_N(x) = P_N(x)))

From P2 and P3 we may conclude
\quad (\forall x: ([\text{un}, x] \land \lnot([\text{in}, x] \land [\text{un}, x] \land \lnot([\text{in}, x])) \Rightarrow P_N(x) = P_N(x))

As for all x
\quad ([\text{un}, x] \land \lnot([\text{un}, x]) \lor ([\text{un}, x] \land \lnot([\text{un}, x])) \Rightarrow P_N(x) \neq P_N(x)

we may write P4 as
\quad (\forall x: [\text{in}, x] \equiv ([\text{un}, x] \lor [\text{un}, x] \land P_N(x) = P_N(x))

As a variant function we choose (\exists x: [v, y] \land \lnot([\text{im}, x] \lor [\text{in}, x])

Program:
\quad \text{loc} (\text{un}, ?, (\text{un}, ?), (\text{im}, ?), (\text{in}, ?)):
\quad [\text{ }] \Rightarrow (\text{un}, x), (\text{un}, y);
\quad \text{do}
\quad x: [\text{un}, x] \land \lnot([\text{un}, x] \lor [\text{un}, x]) \Rightarrow (\text{in}, y);
\quad x: [\text{un}, x] \Rightarrow (\text{in}, x);
\quad x, y: \text{im}, x \land [a, x, y] := (\text{um}, y);
\quad x, y: \text{in}, x \land [a, x, y] := (\text{um}, y)
\quad \text{od}
\quad \text{col}

Upon termination of the repetitive construct we know, from P1 \land \exists x
\quad (\forall x: [v, x] \equiv ([\text{un}, x] \lor [\text{un}, x]))

which, combined with P4', exhibits R to hold.
CHAPTER 11
DYNAMICALLY CREATED NAMES

When programming for associats we establish new relations between names, on account of already existing relations between these names. We may be faced with the situation that the collection of names in the initially existing relations --the names in the associats initially present-- is insufficient to represent the new relations. If the number of necessary new names is statically known --i.e. is independent of the initial state--, then they can be introduced as constants in the program text. If, however, this number is not known until the execution of the program, then new names have to be created dynamically. An instance of this we encounter in the following problem.

Let $V$ be a set of elements. We are requested to write a program that generates all subsets of $V$. The set $V$ is given by

$$(Vx: [v,x] \equiv \text{ "x is the name of an element of V "}).$$

We know, furthermore, of the initial state that $\neg[s,?] \land \neg[m,?,?]$ holds. For any set $X$, $S(X)$ will denote the set of all subsets of $X$. We have to establish the truth of

$$R \equiv (Vx: [s,x] \equiv \text{ "x is the name of an element of } S(V) \text{ "}) \land$$

$$(Vx,y: [m,x,y] \equiv ([s,x] \land [v,y] \land \text{ "element y in subset x "})).$$

We obtain the invariant relation by replacing in $R$ the constant set $V$ by a variable set $W$, which is invariantly contained in $V$. $W$ is initialized empty -- $S(W)$ then contains the empty set only-- and is extended until it equals $V$. The relation

$$S(W \cup \{u\}) = \{S(W) \cup \{x \cup \{u\}: x \in S(W)\}\}$$

suggests to extend $W$ with one element of $V \setminus W$ at a time. Recording the names of the elements of $W$ in associats $(w,?)$, the program will have the following structure.
\text{loc} \{w, ?\}:
\begin{align*}
[] & \assign \{s, \text{empty}\}; \\
do \mathbf{u} \assign [v, u] \land \exists[w, u] \Rightarrow \\
"\text{for all } x : S(w) \text{ generate the set } x \cup \{u\}"; \\
[] & \assign (v, u)
\end{align*}
\text{end}

The refinement of the quoted statement requires the generation of as many new names as the cardinality of \( S(w) \). For any set in \( S(w) \) --whose names are recorded in \text{assoc} \( s, ? \) -- a new set has to be generated. As a consequence we need a new name for each solution of the equation \( x : [s, x] \). For this reason we introduce the possibility to create a new name for each solution of an equation by changing the syntax of "target assoc format" as follows.

\begin{align*}
&\text{<target assoc format>} \assign (\text{<name>}, \text{<ques>}) \\
&\text{<ques>} \assign \text{<name> | <unknown> | !}
\end{align*}

The execution of such a closure statement should affect a final state in which a new name has been created for each solution of the left-hand side. In order to guarantee that this is always possible, we pose the restriction that no target assoc format containing an exclamation point "!" may match the equation in the left-hand side.

The effect of the statement can best be described as if it were executed in two steps. If all target assoc formats contain an exclamation point, then the first step is a "skip", otherwise it is the closure statement that has the original equation as its left-hand side, and the target assoc formats that do not contain the exclamation point as its right-hand side. In the second step the assoc cases with the new names are created, possible cascading being taken care of in the first step. Each element of the solution set of the equation is extended with a unique new name --"new" meaning "distinct from any name occurring in the present assoc cases or as a constant in the program text"--, and for these "extended solutions" the corresponding target assoc cases (with each exclamation point replaced by the new name) are created. If \( x : E(x) \) denotes the equation in the left-hand side, then the number of created new names equals \( |E(x)| \cup x \), in which \( U \) denotes the state after the first step (or, thanks to the "nonmatching", after the second step).
EXAMPLE 11.1. The effect of the closure statement
\[ x: [a, x] \leadsto (b, !), (c, !), (!) \]
is that (Nx: [a, x]) distinct new names are introduced. For each new name
in the associations (b, n) and (c, m, n) are created. The closure statement
\[ x: [a, x] \leadsto (a, x), (b, !), (c, !), (!) \]
has the same effect.

(End of example.)

In the example of the subsets we want to create a new name for every
solution of the equation \( x: [s, x] \). We, therefore, create a one-to-one
function between the solutions of \( x: [s, x] \) and the new names. This func-
tion is temporarily recorded in associations \( (f, ?) \). The closure statement
\[ x: [s, x] \leadsto (f, x, !) \]
establishes the truth of

\[ \forall x, y: [f, x, y] \equiv ([x, x] \land " y = the new name corresponding to \)
\[ x ") . \]

Program:

\[
\text{let } (w, ?) = \]
\[
[] := (s, empty); \quad \text{do } u: [v, u] \land \lnot (w, u) = \]
\[
\text{let } (f, ?, ?) = \]
\[
x: [s, x] := (f, x, !) ; \quad x, y, z: [f, x, y] \land [w, x, z] := (z, y, z) ; \quad y: [f, ?, y] := (z, y, u), (s, y) \]
\[
\text{col}; \quad \text{col} \]
\[
[] := (w, u) \quad \text{do} \]

Now that we can create new names, we could think about creating inte-
gers. The following program should -- according to Peano -- generate the nat-
ural numbers.

\[
[1] := (\text{int}, \text{zero}); \quad \text{do } x: [\text{int}, x] \land \lnot (\text{suc}, x, ?) = [1] := (\text{int}, !), (\text{suc}, x, !) \text{ do} \]

The above program, of course, does not terminate. But this is due to the fact that the set of natural numbers is infinite. We can, however, instead of trying to generate all natural numbers, generate some suitable subset.

Suppose we want to solve Example 10.5 (Most distant leaves of an outcome (p. 82)) by computing for each vertex \( x \) the length \( P(x) \) of the path from the root to \( x \). The subset of the natural numbers ranging from zero through \( (N: [v,x]) \) will then be sufficient to represent these function values.

We record the fact that \( P(x) = y \) by creating the association \( (p,x,y) \). An the most distant leaves are vertices \( x_0 \) for which there exists no vertex \( x_1 \) with \( P(x_1) = P(x_0) + 1 \), the following program will do the job.

\[
\text{loc } (\text{int},?,(\text{suc},?,?)):
\text{"create the natural numbers from zero through \((N: [v,x])\"};
\text{loc } (p,?,?):
\text{x:} [v,x] \land \neg[s,?,x] := (p,x,\text{zero});
\text{x0,x1,y0,y1:} [s,x0,x1] \land [p,x0,y0] \land [\text{suc},y0,y1] := (p,x1,y1);
\text{x0,y0,y1:} [p,x0,y0] \land [\text{suc},y0,y1] \land \neg[p,?,y1] := (d,x0)
\]

The operation "create the natural numbers from zero through \((N: [v,x])\" can be refined into

\[
\text{loc } (h,?):
[] := (\text{int},\text{zero});
\text{do } x,y: [\text{int},x] \land \neg[\text{suc},x,?] \land [v,y] \land \neg[h,y] :=
[] := (\text{int},1),(\text{suc},x,1),(h,y)
\]

\[
\text{rem} \text{ark 11.1.} \text{Apart from the generation of the natural numbers, the program for the recording of the most distant leaves does not involve explicit repetition. Would our programs be executed in an environment in which a sufficient amount of integers were pre-created, we would be able to do away with explicit repetition in many solutions. Nonrepetitive programs give the implementation a larger freedom to choose its own sequencing. In the above}
\]
program for the recording of the most distant leaves, e.g., it is only at
the two semicolons that the implementation has a "synchronization task",
whereas in the repetitive version (Example 10.5 (p. 82)) the implementation
is forced to break up the generation of the associes \((1, 7)\) and \((u, 7)\)
into a number of successive steps that is proportional to the depth of the
tree (which in our new version could be recognized as the "depth of cascading").
(end of remark.)

We shall now solve in the above fashion two other problems that were
previously solved using repetition.

Example 11.2. Balanced vertices. This problem was earlier solved in Chapter
6. We shall now record for any vertex \(x\) the \(--\)in Chapter 8 defined--
function value \(L(x)\), by establishing the truth of
\[
(\forall x, y: [1, x, y] \land ([y, x] \land [int, y] \land L(x) \land y)) \land
(\forall x, y: [n1, x, y] \land ([1, y] \land L(x) \land y))
\]
(In words: associes \((1, 7, 7)\) record lower bounds for the function \(L\),
associes \((n1, 7, 7)\) characterize the recorded lower bounds that are not
attained.)
Then
\[
(\forall x, y: ([1, x, y] \land \neg[n1, x, y]) \rightarrow L(x) = y)
\]

In order to be able to generate the associes \((n1, 7, 7)\), we must first
create the relation "smaller than" on the set of introduced integers. We
shall for that reason establish the truth of
\[
(\forall x, y: [sm, x, y] \equiv ([int, x] \land [int, y] \land x < y))
\]
which can be accomplished by "create the smaller-than relation":
\[
x, y: [sm, x, y] := (sm, x, y);
x, y, z: [sm, x, y] \land [sm, y, z] := (sm, x, z)
\]
Program:
\textit{loc (int,7), (suc,7), (sn,7)}:

"create the natural numbers from zero through \((\text{N}x; [v,x]))":
"create the smaller-than relation":
\textit{loc (1,7), (n1,7)}:
\hspace{1em} x; [v,x] \land \neg [s,x,?] \Rightarrow (1,x,\text{zero});
\hspace{1em} x0,x1,y0,y1; [1,x0,y0] \land [s,x1,x0] \land [\text{suc},y0,y1] \Rightarrow (1,x1,y1);
\hspace{1em} x,y0,y1; [1,x,y0] \land [1,x,y1] \land [\text{suc},y0,y1] \Rightarrow (1,x,y0);
\hspace{1em} x0,x1,y; [1,x0,y] \land \neg [\text{sn},x0,y] \\
\hspace{2em} \land [1,x1,y] \land \neg [\text{sn},x1,y] \Rightarrow (1,x0,x1)"

\textit{col}

\textit{col}

\textbf{REMARK 11.2.} The three statements catering for the creation of the associations (1,7) and (n1,7) can be combined into the following two.
\hspace{1em} x; [v,x] \Rightarrow (1,x,\text{zero});
\hspace{1em} x0,x1,y0,y1,y2; [1,x0,y0] \land [s,x1,x0] \land [1,x1,y1] \land [\text{suc},y0,y2] \\
\hspace{2em} \land \neg [\text{sn},y1,y2] \Rightarrow (1,x1,y2), (n1,x1,y1)"

(End of remark.)

(End of example.)

\textbf{EXAMPLE 11.3. Equidistant issue of two vertices in an undirected graph.}
This is the same problem as Example 10.6 (p. 84). We shall now record for any vertex \(x\) the function values \(P_N(x)\) and \(P_M(x)\), by establishing the truth of (cf. example 11.2 (p. 90))
\hspace{1em} (\forall x,y; [pm,x,y] \Rightarrow ([v,x] \land [int,y] \land P_N(x) \leq y)) \\
\hspace{1em} (\forall x,y; [pn,x,y] \Rightarrow ([v,x] \land [int,y] \land P_M(x) \leq y)) \\
\hspace{1em} (\forall x,y; [gpm,x,y] \equiv ([pm,x,y] \land P_M(x) \leq y)) \\
\hspace{1em} (\forall x,y; [gpn,x,y] \equiv ([pn,x,y] \land P_M(x) \leq y))

Then
\hspace{1em} (\forall x,y; ([pm,x,y] \land \neg [gpm,x,y]) \Rightarrow P_M(x) = y) \\
\hspace{1em} (\forall x,y; ([pn,x,y] \land \neg [gpn,x,y]) \Rightarrow P_M(x) = y)

In order to be able to perform a correct initialization, we must know the maximum of the created natural numbers. We create \textit{max} as the name of the next natural number by means of the program
\[
\begin{align*}
\text{Program:} & \\
\text{loc} (\text{int}, ?, \{\text{suc}, ?, ?\}) := \\
\text{create the natural numbers from zero through } N \{v, x\} & \\
\text{create the smaller-than relation}; & \\
\text{create max as the name of the next natural number}; & \\
\text{loc} (\text{pm}, ?, ?, \{\text{pn}, ?, ?, \{\text{n}, ?, ?, ?\}\}) := \\
\text{x: } [\text{int, } x] & := \{\text{pm}, \text{x, max}\} \\
\text{[ ] := } \{\text{pm, pm, zero}\}, \{\text{pn, zero}, \{\text{pn, pm, max}\}\}, \{\text{pm, pm, max}\}; & \\
\text{x0, x1, y0, y1, x2, x3, y2, y3, x4, x5, y4, y5} & := [\text{a, x0, x1}] \land \{\text{pm, x0, y0}\} \land \{\text{pm, x1, y1}\} \\
\land \{\text{suc, y0, y2}\} \land \{\text{pm, x1, y2}\}; & \\
x0, x1, y0, y1, y2, x2, x3, y2, y3, x4, x5, y4, y5 := \{\text{a, x0, x1}\} \land \{\text{pm, x0, y0}\} \land \{\text{pm, x1, y1}\} \\
\land \{\text{suc, y0, y2}\} \land \{\text{pm, x1, y2}\}; & \\
x, y := \{\text{pm, x, y}\} \land \{\text{pm, x, y}\} \land \{\text{pm, x, y}\} := \{1, x\} \\
\end{align*}
\]

**REMARK 11.3.** Often one is not only interested in the length of a shortest path, but also in the identity of a (the) shortest path(s). Suppose, e.g., that we want to record for each vertex \( x \) in the equidistant locus of \( M \) and \( N \) all shortest paths from \( M \) to that vertex \( x \). We wish to establish the truth of

\[
(\forall x, y, z : [x, y, z] \in \{1, x\} \land \{a, y, z\} \land \{\text{there exists a shortest path from } M \text{ to } x \text{ containing } y + z \}) \\
\]

We know that each "target vertex" \( x \) in the equidistant locus is in all shortest paths from \( M \) to that vertex \( x \). If vertex \( x0 \) is in a shortest path to \( x \), and \( x0 \) is adjacent to a vertex \( x1 \) with \( P_M(x1) = P_M(x0) - 1 \), then \( x1 \) is also in a shortest path to \( x \), and \( x1 + x0 \) is part of a shortest path from \( M \) to \( x \). We record the names of the vertices in the shortest paths in associations \( \{h, ?, ?\} \) by establishing the truth of

\[
(\forall x, y : [h, x, y] \in \{1, x\} \land \{y, x\} \land \{\text{vertex } y \text{ in a shortest path from } M \text{ to } x \})
\]
The truth of (1) is established by inserting at the end of the program (before the first "oo0") the following text.

\[
\text{log (h, x, x)}:
\begin{align*}
x: [m, x] & \Rightarrow (h, x, x) ; \\
x, x_0, x_1, y_0, y_1: [h, x, x_0] & \land [pm, x_0, y_0] \land \neg [gx, x_0, y_0] \\
& \land [suc, y_1, y_0] \land [a, x_0, x_1] \land [pm, x_1, y_1] : (a, x, x_1, x_0), (h, x, x_1) \\
\end{align*}
\]

(End of remark.)

(End of example.)
CHAPTER 12

RECORDING THE CLIQUES OF AN UNDIRECTED GRAPH

In Chapter 8 a program was designed that records an arbitrary clique of an undirected graph. An undirected graph may contain many cliques. The recording of all of them is a more cumbersome affair (an NP-complete problem, in the complexity jargon [1]). This is one reason why we have chosen this problem. The other reason is that it is a problem that with traditional programming tools would probably be solved in a backtracking fashion (cf. [3]).

A finite undirected graph $G$ (possibly containing loops) is given as in Example 7.7 (p. 50). Let, furthermore, be given that $\neg[c,7] \land \neg[a,7]$ holds, and let it be requested to write a program that establishes the truth of

$$
R = \{(x, y) : (x, y) \in G \land (y, x) \in G\} .
$$

As a clique is a maximal complete subgraph, we decide to record all complete subgraphs of $G$. Given the complete subgraphs, the cliques can easily be determined.

Let $V$ denote the set of vertices of $G$. For any subset $X$ of $V$, $CS(X)$ will denote the set of all complete subgraphs of $G$ with vertices from $X$. We record the complete subgraphs of $G$ by establishing the truth of

$$
R1 = \{(x, y) : (x, y) \in CS(V) \land (y, x) \in CS(V)\} .
$$

This problem resembles the problem of the generation of all subsets, for which we designed a program in Chapter 11. We shall try to derive an analogous program for the generation of all complete subgraphs. As in the preceding chapter, we obtain the invariant relation by replacing the constant set $V$ by a variable set $W$.

$$
P1 = \{(x, y) : (x, y) \in CS(W) \land (y, x) \in CS(W)\} = \{(x, y) : (x, y) \in CS(W) \land (y, x) \in CS(W)\} .
$$
Likewise, the program will be of the same structure as the one on page 87. There is, of course, one difference: we do not record all subgraphs, but only the complete ones. More precisely: per execution of the guarded list we generate the extension with vertex \( u \) only for those \( x \in \text{CS}(W) \), that satisfy

\[
(\forall z: [mc, x, z] \Rightarrow [a, u, z]).
\]

This quantified condition can be written as

\[
\neg(\exists z: [mc, x, z] \land \neg[a, u, z]).
\]

We record, as usual, the existentially quantified condition in associns \((h, ?, ?)\), i.e. we maintain as an extra invariant

\[
(P2 \quad (\forall x, u: [h, x, u] \equiv ([v, u] \land (\exists z: [mc, x, z] \land \neg[a, u, z])))\).
\]

(In words: \([h, x, u]\) holds if and only if the complete subgraph \( x \) contains at least one vertex to which \( u \) is not adjacent.) Per execution of the guarded list we shall generate the extension with vertex \( u \) for each \( x \in \text{CS}(W) \) satisfying \( \neg[h, x, u] \).

After the recording of the complete subgraphs of \( G \), we shall determine which complete subgraphs are cliques. By having the scope of the associns \((h, ?, ?)\) range over the complete program, we can exploit the fact that \( P2 \) holds after the generation of \( \text{CS}(V) \). A complete subgraph \( x \) of \( G \) is a clique of \( G \) if and only if for each vertex \( y \) in the complement of \( x \) there exists at least one vertex in \( x \) to which \( y \) is not adjacent; in formula:

\[
(\forall x: [c, x] \equiv ([cs, x] \land (\forall y: ([v, y] \land \neg[mc, x, y] \Rightarrow [h, x, y]))) ,
\]

or

\[
(\forall x: [c, x] \equiv ([cs, x] \land \neg(\exists y: ([v, y] \land \neg[mc, x, y] \land \neg[h, x, y]))) .
\]

We determine the cliques of \( G \) by recording the quantified condition of (1) in associns \((nc, ?)\), i.e. we establish the truth of

\[
(\forall x: [nc, x] \equiv ([cs, x] \land (\forall y: ([v, y] \land \neg[mc, x, y] \land \neg[h, x, y]))) .
\]

Then (1) can be written as

\[
(\forall x: [c, x] \equiv ([cs, x] \land \neg[nc, x])).
\]

The program will, consequently, have the following structure.
\texttt{loc (cs,?), (mc,?,?), (h,?,?)};
\texttt{loc (w,?)};
\hspace{1cm} \{ \} := (cs, \text{empty});
\hspace{1cm} \text{do } u: [v,u] \land \gamma[w,u] \\
\hspace{1cm} \text{"for all } x \in \text{CS}(w) \text{ satisfying } \gamma[h,x,u] \\
\hspace{1cm} \text{generate the extension of } x \text{ with } u, \text{ reestablish } P2"; \hspace{1cm}
\hspace{1cm} \{ \} := (w,u) \hspace{1cm}
\texttt{od};
\texttt{col};
\texttt{loc (mc,?)};
\hspace{1cm} x, y: [cs, x] \land \gamma[v, y] \land \gamma[mc, x, y] \land \gamma[h, x, y] := (mc, x);
\hspace{1cm} x: [cs, x] \land \gamma[mc, x] := (c, x)
\texttt{col};
\hspace{1cm} x, y: [c, x] \land [mc, x, y] := (m, x, y)
\texttt{col}

The refinement of the quoted statement is, except for the condition 
\texttt{\gamma[h,x,u]} and the reestablishing of \texttt{P2}, the same as in Chapter 11 (cf. p. 98);
\hspace{1cm} \text{"for all ..."};
\texttt{loc (f,?,?)};
\hspace{1cm} x: [cs, x] \land \gamma[h, x, u] := (f, x, !);
\hspace{1cm} (2)
\hspace{1cm} x, y, z: [f, x, y] \land [mc, x, z] := (mc, y, z);
\hspace{1cm} y: [f, x, y, u, (cs, y)],
\hspace{1cm} x, y, z: [f, x, y] \land [h, x, z] := (h, y, z);
\hspace{1cm} y, z: [f, x, y] \land [v, z] \land \gamma[a, z, u] := (h, y, z)
\texttt{col}

The last two statements of the above block reestablish \texttt{P2}. A vertex \texttt{x} has a disconnection with the new \texttt{y} if and only if either it has a disconnection with the old \texttt{x}, or it is not adjacent to the added vertex \texttt{u}.

(Note that the second and the fourth statement are "pseudo-cascades", as in the present associations \texttt{(f,x,y) x and y are never equal to each other.}

As each vertex of \texttt{W} is in at least one complete subgraph of \texttt{W}, we have for all \texttt{x}.
\[ [w,x] \in [w,?,x] . \]

We can, consequently, eliminate the associations \((w,?)\).

* * *

We are interested in the recording of cliques. As each clique is a complete subgraph, we decided to record all complete subgraphs of \( G \). The number of complete subgraphs, however, can exceed the number of cliques by far. The complete \( n \)-graph, for instance, has only one clique, but it has \( 2^n \) complete subgraphs. We shall now try to reduce the number of recorded complete subgraphs.

Statement (2) records the names of the new complete subgraphs. We shall accomplish the reduction by changing statement (2) so as to have it create fewer new names.

For execution of the guarded list a new complete subgraph is generated for each \( x \in \mathcal{C}(W) \) that can be extended with vertex \( u \). The old \( x \) remains recorded as well. We investigate whether there are cases in which there is no need to preserve the old complete subgraph. Of course, we can never destroy the old complete subgraph. We can, however, transform it into the new one by simply adding to it the vertex \( u \).

For each recorded complete subgraph \( x \) of \( W \) we call a vertex \( z \) a candidate for \( x \) if and only if \( [w,z] \land [h,x,z] \). When we generate an extension of a recorded complete subgraph \( x \) of \( W \), we do that because ultimately we wish to find all cliques of \( G \) that are supergraphs of that \( x \) and that contain, besides the vertices in \( x \), only candidate vertices for \( x \). If vertex \( u \) is a candidate for \( x \), and \( u \) is adjacent to all other candidates for \( x \), then any such clique will contain vertex \( u \). In that case there is no need to continue the recording of the old complete subgraph \( x \), as it does not (and will not) contain vertex \( u \). Instead of creating a new complete subgraph next to the old one, we shall, in such a case, add vertex \( u \) to the old complete subgraph.

For any recorded complete subgraph \( x \) of \( W \) that has \( u \) as a candidate vertex, we record the candidates in associations \((e,x,?)\), i.e. we establish the truth of
\( \forall x, z \in \{ a, x, z \} \in (\neg \{ a, x \} \land \neg \{ h, x, z \} \land \neg \{ v, z \} \land \neg \{ w, z \} \land \neg \{ h, x, z \}) \).

The old \( x \) has to be kept -- and a new name has to be created -- if and only if

\( \exists z \in a \times u : (v, x, z) \land \neg \{ a, z, u \} \).

Otherwise, we record vertex \( u \) as a member of \( x \); the reestablishment of \( P_2 \) (with respect to \( x \)) may in that case be omitted. We thus arrive at the following test, which should replace statement (2).

\begin{align*}
\text{loc} (u, ?, ?): \\
\neg \{ a, x \} \land \neg \{ h, z, u \} & \\
\land \neg \{ v, z \} \land \neg \{ w, z \} \land \neg \{ h, x, z \} & = (v, x, z); \\
\neg \{ a, x \} \land \neg \{ a, z, u \} & \land a \neq u = (f, x, ?); \\
\neg \{ a, x \} \land \neg \{ f, x, ? \} & = (w, x, u)
\end{align*}

As each vertex of \( W \) is in at least one clique of \( W \), we can eliminate the associations \( (w, ?) \) in this version as well.

If \( G \) is the complete \( n \)-graph (because of its regularity always an easy example for counting purposes), then we have \( [a, z, u] \) for any two vertices \( z \) and \( u \) (\( a \neq u \)). Then the above block never creates a new name, and the program records only one complete subgraph.

**Remark 12.1.** This is another example of a program that traditionally would be solved by backtracking. As in Example 7.3 (Missionaries and Cannibals (p. 45)), we first generate a large enough portion of the "search tree", large enough to ensure that its complement does not contain any "interesting" elements, and then we trace which (or whether any) interesting elements have been generated. It seems that, when programming with associations, backtracking is replaced by this "concurrent generation".

(End of Remark.)

**Remark 12.2.** We have solved an NP-complete problem with a program that contains only one repetitive construct. Its guarded list is executed exactly \( (Wn: (v, x, z)) \) times. All closure statements in the program are either non-cascading or pseudo-cascading. If the complexity theory is also applicable to association programs, this seems to imply that the execution of a non-cascading closure statement can be an NP-complete problem.
(End of remark.)
CHAPTER 13

ON WHAT WE HAVE REJECTED

In the preceding chapters we have presented the ultimate product of our research. We have not tried to give a true design history. Most of the proposals that looked promising at first sight, but that had to be rejected at a later stage, have not been treated. In this chapter we shall deal with some of these proposals, and we shall indicate why we have decided to reject them. We cannot "prove" why we could not accept these proposals. All we shall do is try to communicate our motives for rejection.

It was E.W. Dijkstra who first started to think about programming with associates. His original ideas included, besides the creation of associates, the explicit destruction of associates. The statement

\[ x: [v,x] \Rightarrow [w,x] \]

for instance, would, for any solution of the equation \( x: [v,x] \land [w,x] \), destroy the associate \((w,x)\). Its execution would establish the truth of the implication

\[ (\forall x: [v,x] \Rightarrow \neg[w,x]) \]

The negative match of a target associate and the left-hand side was originally also allowed. The statement

\[ s: [r,x,y] \land [t,x] \Rightarrow (t,y) \]

for instance, would establish the truth of the implication

\[ (\forall x,y: ([r,x,y] \land [t,x]) \Rightarrow [t,y]) \]  (1)

The semantics of our basic statement were described in an operational fashion. We postulated that the effect of the execution of, for instance, the above statement \( s \) is equivalent to the effect of a repetitive implementation that, as long as the equation

\[ x,y: [r,x,y] \land [t,x] \land \neg[t,y] \]

has a nonempty solution set, selects an arbitrary nonempty subset of that solution set, and that creates for all these selected solutions the associates \((t,y)\). (Destruction was described analogously.)
If such a repetitive implementation would select subsets of size one only, then we would have a sequential execution; by only requiring the selected subsets to be nonempty, we leave it to the implementation to choose the degree of concurrency. Note that such an implementation does not necessarily create the minimal number of target associations that is required to establish \( \{ \} \). If \( S \) is, e.g., executed in the state
\[
\{ (t,a), (x,a,b), (x,b,a) \},
\]
then either \( (t,a) \), or \( (t,b) \), or both will be created. It is a nondeterministic construct.

We have tried to write a program for the selection of exactly one (arbitrary) element of a nonempty set \( V \). The names of the elements of \( V \) are recorded in associations \( (v,w) \), the name of the selected element should be recorded in an association \( (s,w) \). We tried the statement
\[
x: [v,x] \Leftarrow (s,x),
\]
but that selects the whole \( V \). We tried the statement
\[
x: [v,x] \land [s,w] \Leftarrow (s,x),
\]
but that selects an arbitrary nonempty subset of \( V \). We tried to select all of them, followed by the destruction of all but one:
\[
x: [v,x] \Leftarrow (s,x);
\]
x:y: [s,x] \land [s,y] \land x \neq y \Leftarrow (s,y).
\]
But then all of them could have been destroyed again.

As we wrote at that time: "a new tool seems indicated." We considered the introduction of a special kind of unknowns — that we during this discussion will denote by adding an exclamation point,— to indicate that the described repetitive implementation should select subsets of size one only, and we arrived at the program
\[
x!: [v,x] \land [s,w] \Leftarrow (s,x).
\]
(The program
\[
x!: [v,x] \Rightarrow (s,x)
\]
would still copy the whole \( V \), at least at a time.)

For "ordinary" unknowns the implementation may select an arbitrary nonempty subset of the solution set, for unknowns "x!" it selects only one so-
olution at a time. What if we have both types in one statement? We declared
the order in which the unknowns are listed to be important, and we designed
a very complicated rule —too complicated to be repeated here—for the per-
mitted selections of solutions. Consider, e.g., the execution of the program

$$\text{if } x, y : \{a, x, y\} \Rightarrow \{b, x, y\}$$  

in a state satisfying \( \neg \{b, x, y\} \). For each \( x \) such that \( \{a, x, y\} \), it would
select one \( y \) satisfying \( \{a, x, y\} \), and for these \( (x, y) \)-pairs, it would
create the associations \( \{b, x, y\} \). The program

$$\text{if } x, y : \{a, x, y\} \Rightarrow \{b, x, y\},$$  

on the other hand, would (if \( \{a, x, y\} \)) select one \( y \) satisfying \( \{a, y, y\} \),
and (with that \( y \)) it would, for all \( x \) satisfying \( \{a, x, y\} \), create the
associations \( \{b, x, y\} \).

Even with such a scheme, repetition was still indispensable. We did,
for instance, not succeed in writing a nonrepetitive program for the selec-
tion of one characteristic element per equivalence class. As it, furthermore,
turned out to be virtually impossible to give a nonoperational description
of its semantics, we dropped this complicated scheme, and we decided to have
only one type of unknown, indicating by ":=" (instead of "⇒") that the im-
plementation should select one solution at a time. The statement (3) could
then be coded as

$$\text{if } x, y : \{a, ?, y\} \land \{1, ?, y\} \Rightarrow \{1, y\};$$

$$\text{if } x, y : \{a, x, y\} \land \{1, y\} \Rightarrow \{b, x, y\},$$

whereas the equivalent of (2) would require repetition. (The single arrow
"⇒" was dropped again when we started to think about repetition.)

The definition of our statement should be such that it allows for the
derivation of practicable theorems that characterize all relevant aspects
of the semantics. When we tried to achieve this, we did not succeed for
statements with a negative match, neither did we succeed in a satisfactory
manner for statements with association destruction. We shall elucidate this.

The property that the execution of the statement

$$\text{if } x, y : \{r, x, y\} \land \neg \{t, x\} \Rightarrow \{t, y\}$$

establishes the truth of the implication

$$\forall x, y : \{r, x, y\} \land \neg \{t, x\} \Rightarrow \{t, y\}$$  

(4)
does not exclude the possibility that too many association \((t,?)\) are created. One could, for instance, in order to establish the truth of \((4)\) create for all "prevailing" names \(x\) the associations \((t,x)\) (cf. the set \(\mathcal{V}\) in the proof of Lemma 5.2 (p. 25)).

We wish to express that if prior to the execution of \(S\) no "wrong" associations \((t,?)\) --- "wrong" in the sense that "their unknowns do not constitute a solution of the left-hand side"--- were present, then they will not be present afterwards either; i.e. we wish to express the invariance of "no wrong associations present". For the statement \(S\) this would be the invariance of the implication

\[
\forall y: \{t,y\} \rightarrow \exists x: \{x,y\} \land \lnot\{t,x\} .
\]

In the case of a negative match --- as in \(S\) --- this invariance, however, is in general not guaranteed, as the created target associations can cause the solution set of the left-hand side to shrink. We decided to disallow the negative match of target associations and the left-hand side.

For a statement with association destruction the prevention of shrinking solution sets gives rise to the disallowance of target associations that fit positive presence conditions of the left-hand side. With such a prohibition we obtain the invariance of "no wrong associations absent". The statement

\[
x,y: \{x,y\} \land \lnot\{s,x\} \Rightarrow \{s,y\} ,
\]

for instance, would establish the truth of the implication

\[
\forall y: \exists x: \{x,y\} \land \lnot\{s,x\} \Rightarrow \lnot\{s,y\}
\]

under invariance of "no wrong associations absent":

\[
\forall y: \lnot\{s,y\} \Rightarrow \exists x: \{x,y\} \land \lnot\{s,x\} .
\]

This invariance, however, is of little use. As the set of names is unlimited, implication \((6)\) will be false for any finite set of associations. Having "automatic destruction" of local associations at block exit, there is little sense in introducing explicit association destruction as well. If one wishes to record that certain associations are not "valid" anymore, one can always record this in additional associations, and write, instead of \((5)\), e.g.,

\[
x,y: \{x,y\} \land \lnot\{s,x\} \Rightarrow \{\text{invalid},x\} \lor \lnot\{s,x\} .
\]

We decided to reject explicit association destruction.
At this stage we hoped to have simplified our basic statement — then called "creation statement"— enough, as to allow for a non-operational definition of its semantics. This turned out to be rather difficult for cascading creation statements. We did, however, not want to disallow cascade, as we felt

$$v,y: [v,x] \land [x,x,y] \Rightarrow (v,y)$$

to be a statement too beautiful to be rejected. We decided to introduce the mathematical concept "closure of a set of association". We postulated that the creation statement generates the closure, and we called it "creation statement" since.

* * *

It was at this stage of the research that we again introduced some ideas that we have not maintained.

We, for instance, wished to distinguish between unknowns that only occur in the left-hand side and the other unknowns of a closure statement, calling them "existential" and "free" variables, respectively. This distinction caused matters to become needlessly intricate, and after we realized that

$$(\forall x: P(x) \Rightarrow Q)$$

is equivalent to

$$(\exists x: P(x)) \Rightarrow Q,$$

we dropped this distinction again.

We also have considered to have the closure statement abort if initially there is a "wrong" association present. The statement

$$x: [v,x] \Rightarrow (w,x),$$

for instance, should abort if initially

$$(\exists x: [w,x] \land \neg [v,x])$$

holds. Although it was felt encouraging to have a construct in the language that could give rise to an abort, we did not pursue this idea. In order to avoid aborting we had to introduce auxiliary associations to record intermediate results. The presence conditions of these auxiliary associations lengthened
the equations of subsequent closure statements.

Another idea that has been abandoned was the treatment of the concurrent closure statement as the basic statement. This complicated our presentation tremendously: we had subscripts—and sometimes even subscripts—hanging around everywhere. Although the concurrent closure statement is more powerful than the nonconcurrent one, we could only find far-fetched cases in which the concurrent closure statement could not be written as a nonconcurrent one.

* + *

The execution of a statement, be it a block or a single closure statement, cannot effect the destruction of associations that are global to that statement. The effect of a statement that does not create global associations, is, consequently, equivalent to that of "skip". When we first embarked upon the introduction of repetition, we observed that there is little sense in continuing to execute a repetitive construct if an execution of the "repeatable statement" has not created any global associations. We considered not to introduce explicit guards in our repetitive construct, but to postulate that the execution of a repetitive construct finishes as soon as a single execution of the repeatable statement has not created any associations global to it. Every repetitive construct that does not involve dynamic generation of new names, would then be a terminating construct.

For such a scheme the formulation of the invariance theorem turned out to be rather difficult. For we had to characterize the states in which no global associations are created.

Without the "single arrow" closure statement we were, furthermore, not able to write a program for the selection of one characteristic element per equivalence class. The single arrow closure statement had in the meantime become suspect. For its effect could only be described nonoperationally by means of the \((N...)-concept\), and that concept usually does not provide a simple interface.

We then decided to combine (from the single arrow closure statement) the idea of selection of one solution at a time, and (from traditional programming) the idea of explicit guards. We introduced equations as guards. A
guarded list is only eligible for execution if its guarding equation has a nonempty solution set. If a guarded list is chosen to be executed, then an arbitrary solution of its guarding equation is selected and may be "used" during the execution of the guarded list.

We decided to allow a set of guarded commands, although we never came across a problem for which the program contained a repetitive construct with more than one guard. It may be that repetitive constructs with only one guard suffice, but we introduced the general guarded command set, because without complicating matters it enabled us to formulate Theorem 9.1 (p. 67) about the arbitrariness of the order in which an implementation of the closure statement may trace (and create) the missing associations.
CHAPTER 14

EPilogue

We have proposed an instruction repertoire that should allow ultraconcurrent execution. New hardware techniques may make such implementations feasible (cf. [14]). One could try to reach such a goal by the introduction of large fancy data-types, upon which numerous powerful operations are defined. We have not done so. We have designed a limited and simple instruction repertoire, consisting of mathematically well-manageable concepts. We did not introduce any concept without discussing its desirability, nor did we introduce any concept without simultaneously providing a discipline for its use.

As a data structure we have chosen a set of relations. On this data structure we have defined only one operation, the closure statement. It creates, as a function of the existing relations, new relations. Its semantics are formally defined by means of its weakest pre-condition. Two sequencing primitives have been introduced: concatenation—the semicolon—and repetition. In order to be able to judge when—and how—to employ repetition, an appropriate design principle—the recording of the quantified condition—has been formulated.

Designing a language for a nonexisting machine, we enjoyed the (lucky) circumstance of being forced to obey Wirth’s following demand (cited from [15], underlining as in the original): "The first criterion that any future programming language must satisfy (...) is a complete definition without reference to compiler or computer. Such a definition will inherently be of a rather mathematical nature. To many hardcore programmers, this demand perhaps sounds academic and (nearly) impossible. I certainly have not claimed that it is easy! I only claim that it is a necessary condition for genuine progress."

We have fairly often encountered programs that could be used for many (at first sight different) problems. This may perhaps confirm the idea that there do not exist too many really different algorithms. It also seems to indicate that, when programming for associations, we arrive at some sort of "canonical" programs.
A need of recursion did not arise. We did introduce a "cascading creation" which is nothing else than the creation of a (generalized) closure. If \( U \) denotes the set of vertices of some out-tree, of which \( R \) represents the predecessor-successor relation, then determining the closure under \( R \) of the subset of \( U \) that contains only the root, is also known as "traversing a tree". Its classical solution is a recursive one. The recursive solution for tree-traversal is often regarded to be the prototype application for recursive subroutines. The generation of closures, however, is not restricted to trees, it may be applied in directed graphs in general. As such cascade may be considered to be a logical generalization of recursion. Recent publications (cf. [5]) seem to indicate that the generation of closures in directed graphs is a frequently recurring theme.

There is another branch of computing science in which the data structure consists of a set of relations, viz. that of "relational data bases" [4], and one may wonder how different our approaches are.

In articles on (relational) data bases the data structure usually models named objects of the "real world". (Examples then deal with "suppliers and parts", "employers and employees", etc.) As a consequence the name of a relation --which is not considered to be the name of a "real" object-- is treated differently from the arguments of a relation. Our association

\[
\text{(larger-than, transitive)},
\]

representing that the larger-than relation is a transitive relation, would then either have to be coded as

\[
\text{larger-than(transitive)},
\]

or as

\[
\text{transitive(larger-than)},
\]

meaning, respectively, "transitivity is one of the properties of the larger-than relation", and "larger-than is one of the transitive relations". This choice is not an irrelevant one, as names of relations may usually not occur as arguments in relations. The symmetry in the association representation allows us to write both statements like

\[
x \in [x, \text{transitive}] \land [x, \text{reflexive}] \land [x, \text{symmetric}] \Rightarrow (x, \text{equivalence}),
\]

and statements like
\( x \in \{ \text{larger than }, x \} \Rightarrow (\text{smaller than }, x) \).

A second major difference seems to be that in relational data bases only some standard operations for the creation of relations are defined. These operations may comprise (in the data base jargon) composition, (cyclic) join, projection, restriction, tie, etc. All of these are special cases of the closure statement. The resulting relation, however, is always a relation with a different name. A cascading creation can, consequently, not be expressed in such repertoires.

Another branch of computing science in which the need of operations on relations—including that of the generation of the transitive closure—has been uttered, is the field of artificial intelligence. To quote a paper on programming languages for artificial intelligence [2]: "In particular, (...) one would like to be able to use content-retrievable ordered triples (or n-tuples) (...) as basic data types." The languages Sail and Planner do have these content-retrievable tuples. They are then called, respectively, associations and assertions. In neither of these languages, however, the creation of closures has been recognized as a primitive operation.

The above observations give a strong indication that the creation of closures is indeed a fundamental concept. The preceding chapters have shown that it is very well possible to construct a programming language around it.
BIBLIOGRAPHY


pp. 379 - 387.


Thesen, Berichte des Institutes für Informatik 17, ETH Zürich, March 1976.
<table>
<thead>
<tr>
<th>INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>abort</td>
</tr>
<tr>
<td>acyclic graph</td>
</tr>
<tr>
<td>adjacent vertices</td>
</tr>
<tr>
<td>assignment statement</td>
</tr>
<tr>
<td>association</td>
</tr>
<tr>
<td>association format</td>
</tr>
<tr>
<td>B0</td>
</tr>
<tr>
<td>B1</td>
</tr>
<tr>
<td>balanced vertices</td>
</tr>
<tr>
<td>block</td>
</tr>
<tr>
<td>BNF</td>
</tr>
<tr>
<td>cascading closure statement</td>
</tr>
<tr>
<td>clique</td>
</tr>
<tr>
<td>closure statement</td>
</tr>
<tr>
<td>closure with respect to a binary relation</td>
</tr>
<tr>
<td>closure with respect to a closure statement ( C(S, U) )</td>
</tr>
<tr>
<td>complete graph</td>
</tr>
<tr>
<td>concurrent closure statement</td>
</tr>
<tr>
<td>condition (on the state)</td>
</tr>
<tr>
<td>connected</td>
</tr>
<tr>
<td>constant</td>
</tr>
<tr>
<td>cycle</td>
</tr>
<tr>
<td>deterministic construct</td>
</tr>
<tr>
<td>edge-disjoint paths</td>
</tr>
<tr>
<td>empty association</td>
</tr>
<tr>
<td>equation</td>
</tr>
<tr>
<td>equidistant locus</td>
</tr>
<tr>
<td>exclamation point</td>
</tr>
<tr>
<td>Term</td>
</tr>
<tr>
<td>----------------------------------------</td>
</tr>
<tr>
<td>$F_i$</td>
</tr>
<tr>
<td>factot</td>
</tr>
<tr>
<td>fit</td>
</tr>
<tr>
<td>guarded</td>
</tr>
<tr>
<td>guarded command</td>
</tr>
<tr>
<td>guarded list</td>
</tr>
<tr>
<td>$R_k(R)$</td>
</tr>
<tr>
<td>initialized constant</td>
</tr>
<tr>
<td>invariance theorem for closure statements</td>
</tr>
<tr>
<td>invariance theorem for repetitive constructs</td>
</tr>
<tr>
<td>kernel</td>
</tr>
<tr>
<td>left-hand side</td>
</tr>
<tr>
<td>length of a walk</td>
</tr>
<tr>
<td>local associons</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>match</td>
</tr>
<tr>
<td>mechanism</td>
</tr>
<tr>
<td>minimal transition pair</td>
</tr>
<tr>
<td>multiple edges</td>
</tr>
<tr>
<td>$(N \ldots)$</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>negative match</td>
</tr>
<tr>
<td>negative presence condition</td>
</tr>
<tr>
<td>nontrivial walk</td>
</tr>
<tr>
<td>out-tree</td>
</tr>
<tr>
<td>path</td>
</tr>
<tr>
<td>positive match</td>
</tr>
<tr>
<td>positive presence condition</td>
</tr>
<tr>
<td>predecessor</td>
</tr>
</tbody>
</table>
predicate \( P \upharpoonright U \)

predicate transformer

presence condition

primary

question-mark

reachable

repetitive construct

\( \& \)

safely connected vertices

semicolon

set difference \( \backslash \)

skip

solution set \( Z(\mathbf{z},U) \)

spanning subgraph

state

statement list

state vector

strong component

strongly connected

subgraph

successor

supergraph

symmetric set difference \( \uplus \)

\( T(x), \bar{T}(x) \)

target association

target association format

target association format set

term

terminal vertex

terminating construct

transitive closure

trap
unilateral component 61
unilaterally connected 61
universal relation 6
unknown 9

variant function 60

walk 47,50
wdoc 72
weakest pre-condition wp(S,R) 2
\exists(s,u) 9
ZIMENVATTING

In het proefschrift worden (programmeertaal)constructies voorgesteld die de implementatie een zeer hoog — niet statisch bepaald — graad van gelijkvrijheid toestaan, zonder deze echter van de implementatie te eisen. Dergelijke implementaties kunnen door nieuwe hardware-technieken actueel worden.

Tien eindig het repertoire van taalconstructies systematisch en beperkt te houden, worden toestanden van berekeningen geregistreerd in relaties — "associations" — met "names" als argumenten. Er wordt slechts één statement — de "closure statement" — voorgesteld, die de toestand van het systeem (de verzameling aanwezige associaties) kan wijzigen. De closure statement creëert als functie van de aanwezige associaties nieuwe associaties. Als sequencing primitivies — taalconstructies waarmee de mogelijke uitvoeringsvolgorde van de closure statements kunnen worden vastgelegd — worden uitsluitend de puntkoma en de repetitie voorgesteld.

Een gevolg van deze beperkte en systematische opzet is dat er voor de programma's een soort "kanonische" vorm ontstaat, waardoor hetzelfde programma vaak van toepassing blijkt op (op het eerste gesicht) verschillende problemen. De associai-programma's hebben de neiging kort te zijn en weinig expliciete sequencing te vereis. In vergelijking met conventionele programma's zijn associai-programma's, ondanks de potentieel hoge graad van gelijkvrijheid van hun uitvoering, en ondanks hun compacte notatie, goed te begrijpen en niet moeilijk te ontwerpen. Een behoefte aan recursie doet zich nergens voelen.

Hoofdstuk 1 is een inleidend hoofdstuk. Hierin wordt o.a. het formele apparaat — de "weakest pre-conditions" van Dijkstra— beschreven waarmee in latere hoofdstukken de semantiek van de taalconstructies zal worden gedefinieerd. In hoofdstuk 2 worden de associaties ingevoerd als argumentlijsten van een universele relatie. Hierdoor ontstaat een uniforme behandeling van namen van relaties en argumenten van relaties. Hoofdstuk 3 behandelt het karakteriseren van toestanden van berekeningen met behulp van oplossingsverzamelingen van vergelijkingen.

In hoofdstuk 4 wordt de closure statement op operationele wijze ingevoerd. Tevens wordt in dit hoofdstuk de syntax van de closure statement ge-
definiërd. Voor elke closure statement S bestaat er een verzameling en- 
voudige (slechts twee factoren in het linker lid) closure statements waarvan 
de "uitvoering" hetzelfde netto resultaat oplevert als de uitvoering van S 
zelf. Dit geeft vertrouwen in de mogelijkheid om willekeurig ingevulde 
closure statements te implementeren. In hoofdstuk 5 worden twee definities 
de afsluiting van een verzameling associaties onder een closure statement 
gegeven. Hun equivalentie wordt bewezen. Bij iedere closure statement hoort 
een relatie en een verzameling "target associaties". De afsluiting van een 
verzameling V van associaties onder een closure statement S is nu de kleinste 
uitbreiding van V met target associaties van S, waarvoor de relatie van S 
geldt. De uitvoering van een closure statement transformeert de toestand U 
(een verzameling associaties) naar de afsluiting van U onder die closure state- 
ment. In hoofdstuk 6 wordt de semantiek van de closure statement formeel ge-
definieerd. Hoofdstuk 7 bevat voorbeelden van associatie-programma's.

In hoofdstuk 8 wordt (aan de hand van voorbeelden) de behoefte aan re-
petitie en locale associaties aanspankelijk gemakzakt, en de repetitieve constructie op operationele wijze ingevoerd. De formele definitie van de repetitieve constructie vindt plaats in hoofdstuk 9. Bewezen wordt dat een implementatie 
de van de closure statement die te genereren afsluiting kan bepalen door (in 
willekeurige volgorde) te zoeken naar argumenten waarvoor de relatie van de 
closure statement niet geldt. Met behulp van de invariantieseteling van 
Boas voor repetitieve constructies wordt de correctheid van een programma 
uit hoofdstuk 9 bewezen. Hoofdstuk 10 bevat voorbeelden van associaties-pro-
gramma's met repetitie. In hoofdstuk 11 wordt een mechanisme geïntroduceerd 
om dynamisch nieuwe namen te creëren. Nadat hiervoor de integers zijn inge-
voerd, wordt een aantal voorbeelden uit het voorgaande hoofdstuk zonder re-
petitie opgelost.

Het genereren van alle kliken van een ongerichte graaf is een van de 
klassieke "niet-polynomische" problemen. In hoofdstuk 12 wordt voor dit pro-
bleem een eenvoudige oplossing zonder backtracking gegeven. In hoofdstuk 13 
wat teruggeblikt op het onderzoek. Van een aantal alternatieve mogelijk-
heten wordt aangegeven waarom deze niet gekozen zijn. Hoofdstuk 14, tenslot- 
te, bevat een aantal slotomkeningen. Onder meer worden hierin associaties 
vergelijken met relatiional data bases en soortgelijke datastructuren in talen 
voor artificial intelligence.
CURRICULUM VITAE

I

Voorberekeningen waarvan het resultaat een booleaan waarde is, kost het con-
troleren van de juistheid van het antwoord evenveel moeite als het genereren ervan.

II

De invariantiestelling voor closure statements [1] geeft geen volledige ka-
rakterisering van de semantiek van de closure statement.


III

Een eengante definitie van de wdec, dan die in [2], is

\[ \text{wdec}(S,t) \equiv \text{wp}(S,t < \tau_0) \]

waarin \( \tau_0 \) het predicat \( \tau \) is, met ieder voorkomen van \( \tau_0 \) vervangen
door \( t \).


IV

De stelling

"Zij \( a_0, a_1, \ldots, a_n \) (\( n > 0 \)) een rij getallen met \( a_0 < a_n \),
en zij \( x \) zodanig dat \( a_0 < x < a_n \), dan geldt

\( \forall i: 0 \leq i < n: a_i \leq x < a_{i+1} \)"

volgt uit de eindigheid van het programma.
i, j := 0, n;
\textbf{do} i + 1 \neq j \rightarrow m := (i + j) \text{ div } 2;
\begin{align*}
&\text{if } a_m \leq x = i := m \\
&\text{else } 0 < a_m + j := m
\end{align*}
\textbf{od}

en de invariantie van

\[ a_i \leq x < a_j \land 0 \leq i < j \leq n. \]

De in [3] gegeven karakterisering van alle vierde orde vier-punts Runge-Kutta methoden voor het oplossen van een eerste orde differentiaalvergelijking \( y' = f(x, y) \) is niet volledig. Het schema

\[ y_{n+1} = y_n + \frac{1}{6}((1 - \theta)k_0 + 4k_1 + \theta k_2 + k_3), \]

met

\begin{align*}
k_0 &= hf(x_n, y_n), \\
k_1 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_0), \\
k_2 &= hf(x_n, y_n + \frac{1}{2}k_0), \\
k_3 &= hf(x_n + h, y_n + \frac{3}{2}k_0 + \theta k_1 + \frac{3}{2}k_2), \\
h &= x_{n+1} - x_n, \\
\theta &\text{ vrij te kiezen } (\theta \neq 0).
\end{align*}

is ook vierde orde exact.


VI

Een mechanisme \( S \) noemen wij \textit{idempotent} dan en slechts dan als voor alle \( P \) geldt

\[ \wp(S, \wp(S, P)) \equiv \wp(S, P). \]
Een mechanisme $S$ noemen wij reversibel dan en slechts dan als voor alle $P_1$ en $P_2$ geldt

$$(wp(S,P_1) \equiv wp(S,P_2)) \implies (P_1 \equiv P_2).$$

Hieruit volgt onmiddellijk dat de (empty) statement $S$, waarvan de semantiek gegeven is door

$$wp(S,P) \equiv P,$$

de enige statement is die zowel idempotent als reversibel is.

VII

Het programma in [4, p. 96] voor het partitioneren van een array

$$av(f: 1),$$

zodanig dat na afloop geldt

$$(\forall k: i \leq k < i; av(k) < a) \land$$
$$(\forall k: i \leq k < i; av(k) > a),$$

is ingewikkeld en fout. Deze observatie onderstrept dat een testgeval, zoals de auteurs in Example 3.3 hebben gemeend te moeten tonen, niet de plaats van een correctheidsbewijs kan innemen.


VIII

Voor sommige zelfstabiliserende systemen [5] kan de bewijsvoering zeer vereenvoudigd worden door deze systemen te transformeren naar sequentiële systemen die hetzelfde eindresultaat bewerkstelligen.


IX

Het gebruik van het symbool "$\Rightarrow$" als implicatietermen is verwarrend.
X

Inleidend programmeeronderwijs is niet gebaat bij het ter beschikking staan van een computer. Het gebruik van terminalfaciliteiten in het ondervijs moet worden ontraden.

XI

Het is onverantwoord een automatiseringsysteem in bedrijf te stellen, zonder zorgvuldig voorbereid te hebben hoe het weer buiten bedrijf moet worden gesteld.

XII

Het wijdverbreide vertrouwen op per computer geproduceerde resultaten is misplaatst, en daardoor gevaarlijk.