

Eindhoven University of Technology
Department of Mathematics and Computing Science

Predicate calculus: concepts and misconceptions

by

Lex Bijlsma and Rob Nederpelt

96/21

ISSN 0926-4515

All rights reserved

editors: prof.dr. R.C. Backhouse
prof.dr. J.C.M. Baeten

Reports are available at:
<http://www.win.tue.nl/win/cs>

Computing Science Report 96/21
Eindhoven, November 1996

Predicate calculus: concepts and misconceptions

Lex Bijlsma and Rob Nederpelt
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands

October 30, 1996

Abstract

The paper focusses on the logical backgrounds of the Dijkstra-Hoare program development style for correct programs. For proving the correctness of a program (i.e. the fact that the program satisfies its specifications), one uses the so-called *predicate calculus* in this style of programming. Predicate calculus can be conceived of as a logically sound and complete manipulation technique for dealing with logical formulas which also contain programming variables.

We relate predicate calculus to the classical logical formalism, by contrasting its syntax, derivation rules and semantics to the classical framework. We also comment on two abstractions of predicate calculus: the set-theoretical and the algebraic approach. In doing so, we give predicate calculus and its abstract variants a firm basis, on a par with the foundations of the well-known first order logic. Such a comparison of predicate calculus and classical logic has not yet been sufficiently elaborated before.

We conclude our paper with a number of examples showing that the, up to now, unsatisfactory presentation of predicate calculus and some of its features (such as the square brackets notation) has led to errors and fallacies in the literature.

Apologia

In this paper we try to serve two masters: both masters are computer scientists, but one is familiar with (elementary) traditional logic, while the other one prefers the logic of predicate calculus in the Dijkstra-style. Since it is impossible to serve two masters satisfactorily, we foresee that our style of presentation will be alternately lengthy and compact for the one, and compact and lengthy for the other. If this is the case, we beg for understanding. All we try to do is to reconcile the two logical styles by bringing them together.

1 Introduction

Program derivation is a semi-formal style of program synthesis originated by Dijkstra [D76] and Hoare [H69]. Its practice, of which an up to date exposition can be found in [K90], involves a view of predicate calculus that is somewhat different from the one traditionally prevalent among logicians. As we shall show by examples taken from the literature, these differences have led to a certain amount of confusion. Although a book-length exposition of Dijkstra's views on predicate calculus is available [DS90], this contains no references to other approaches

and seems to have little success in clarifying the confusion surrounding certain concepts, in particular structures, punctuality, and the ‘everywhere’ operator.

The purpose of this paper is to distinguish and contrast various approaches to Dijkstra’s predicate calculus, to give proper definitions of the concepts involved, and to pinpoint several places in the literature where a failure to separate incompatible approaches has led to actual errors.

This paper is organized as follows.

In Part I we give an overview of the fundamental aspects of predicate calculus as it is currently in use.

We start in section 2 with a description of the *well-formed and well-typed logical formulas* of predicate calculus, relative to a context of typed program variables. This leads to the notion of ‘predicate’. We also mention well-typed programs and Hoare-triples. The deductive system of predicate calculus is described in section 3, together with the format in which deductions are presented in the predicate calculus style. Its standard semantics (the *state semantics*, giving the standard model) is described in section 4.

Section 5 gives two abstract views on predicate calculus: the set-theoretical abstraction and the algebraic abstraction.

Section 6 explains the extension of predicate calculus with the so-called square brackets of Dijkstra en Scholten.

In section 7 we compare the different approaches to predicate calculus (syntactic; semantic; abstract).

In Part II we pinpoint several cases of misunderstandings concerning the different forms of predicate calculus.

In section 8 we show how the different approaches to predicate calculus, as explained in Part I, have been mixed in the literature. In section 9 we demonstrate that the absence of explicit types has caused problems. Finally, in section 10 we give examples in which the separation between language and meta-language has abusively been ignored.

The paper ends with conclusive remarks.

PART I: Concepts

2 The syntax of predicates

We start this section with giving a formal framework of the logical formalism used in the program derivation community. In this and the following section we describe the first order language only on which this method for constructing correct programs is based. In recent developments, the logic used in this program construction community has been extended to higher orders. We give examples in sections 6 and 9. This higher order logic, however, being still in development, is not treated as thoroughly in the present paper as the first order part is.

The syntactic first order framework that we describe can be compared with the approach of [G81], [AO91] or [GS95a], since it uses axioms and inference rules. However, we give a more precise characterization of what logical formulas are, splitting the set of variables into logical variables and programming variables. Moreover, we introduce types and contexts in order

to give an adequate definition of the kind of predicates used in the Dijkstra-Hoare predicate calculus. Finally, we describe how programs and Hoare-triples fit in this setting.

Logical formulas

There are two kinds of formulas involved in the Dijkstra-Hoare style of programming: the *logical* formulas and the *programming* formulas. We concentrate on the logical formulas, since the formulas for programming are sufficiently elaborated in the literature. (See e.g. [M90].)

As is usual in logic, logical formulas have so-called terms as constituents. Terms are used to construct propositional formulas and predicate-logical (quantified) formulas. We discuss these three notions consecutively.

As to the terms:

Terms are constructed from the following alphabet, in the usual manner (see the definition below):

- variables and constants,
- function symbols.

The set of variables, contrary to the usual logical situation, is divided into two disjoint sets: the set \mathcal{V}_l of *logical* variables and the set \mathcal{V}_p of *programming* variables. The idea behind this is that the elements of \mathcal{V}_p are the identifiers from the program text, while the elements of \mathcal{V}_l will be used as bound variables in logical formulas. The set of constants will be denoted by \mathcal{C} and the set of function symbols by \mathcal{F} . A typical element of \mathcal{C} is **true**.

Each symbol in \mathcal{F} has a fixed, positive arity. For example, if $f \in \mathcal{F}$ with arity two, $0 \in \mathcal{C}$, $n \in \mathcal{V}_l$ and $x \in \mathcal{V}_p$, then $f.n.0$ and $f.x.(f.n.n)$ are terms. As these examples show, we use the dot ‘.’ for function application.

Hence, we can give the following definition of terms:

Definition 2.1 *The alphabet consists of the disjoint sets of symbols \mathcal{V}_l , \mathcal{V}_p , \mathcal{C} and \mathcal{F} . There is a fixed arity function $\mathcal{F} \rightarrow \mathbf{N} \setminus \{0\}$.*

Each $n \in \mathcal{V}_l$, $x \in \mathcal{V}_p$, $c \in \mathcal{C}$ is a term.

If $f \in \mathcal{F}$ with arity n and if t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term.

As to the propositional formulas:

In the alphabet for the propositional formulas we have two more sets:

- relation symbols,
- logical connectives.

Again, each relation symbol has a fixed arity. The set \mathcal{R} of relation symbols includes such well-known symbols as $>$ and $=$.

The set of logical connectives contains \neg , \vee , \wedge , \Rightarrow (for formal implication) and \equiv (for equivalence).

The propositional formulas are constructed from terms, relation symbols and logical connectives as usual (see below). For example, propositional formulas are $(f.n.0 > x) \wedge (y = 0)$ and $\neg(b \equiv \mathbf{true})$. (Here b is a variable, e.g. $b \in \mathcal{V}_p$; binary relation symbols and binary connectives are, as usually, written in infix format.)

Definition 2.2 *The alphabet is extended with a new set \mathcal{R} and sets \mathcal{L}_1 and \mathcal{L}_2 of (unary and binary) logical connectives.*

There is a fixed arity function $\mathcal{R} \rightarrow \mathbf{N} \setminus \{0\}$.

If $r \in \mathcal{R}$ with arity n and if t_1, \dots, t_n are terms, then $\mathcal{R}(t_1, \dots, t_n)$ is a propositional formula.

If φ and ψ are propositional formulas and if $\sim \in \mathcal{L}_1$ and $\bullet \in \mathcal{L}_2$, then $\sim \varphi$ and $\varphi \bullet \psi$ are propositional formulas.

As to the predicate-logical formulas:

Finally, the alphabet of the predicate-logical formulas contains one more set:
– quantifiers.

The quantifiers are the usual ones: \forall and \exists . Each quantification has a *domain*, which is a formula denoting a set. We call these formulas *domain formulas* or *types*; they have a syntax of their own (which we shall not describe) and are imported in the predicate-logical formulas. (See below for more comments on types.)

Quantified formulas are written in a special format, e.g.: $(\exists n : n \in \mathbf{N} : n < x)$ for: there exists n in \mathbf{N} such that $n < x$. Here \mathbf{N} is the domain and n a bound logical variable.

One makes a flexible use of this format, which enables a quantified formula like $(\exists n : n > 0 \wedge n < 9 : n < x)$, which can be considered to be a handy abbreviation for $(\exists n : n \in \{k \in \mathbf{Z} | k > 0 \wedge k < 9\} : n < x)$. Here $\{k \in \mathbf{Z} | k > 0 \wedge k < 9\}$ is the domain. Note the convention that domains are subsets of \mathbf{Z} by default.

The predicate-logical formulas are constructed from terms, relation symbols, logical connectives and quantifiers (including types), as usual:

Definition 2.3 *The alphabet is extended with the quantifiers \forall and \exists .*

If τ is a domain formula and, $n \in \mathcal{V}_l$ and t is a propositional formula or a predicate-logical formula, then $(\forall n : n \in \tau : t)$ and $(\exists n : n \in \tau : t)$ are predicate-logical formulas.

We shall speak of formulas as a shorthand for ‘predicate-logical formulas’.

Many of the function symbols and relation symbols used in this formalism have a fixed standard interpretation, e.g. “greater than” for $>$. (See also section 4.) The interpretation of the connectives and quantifiers is as usual.

Variables and types

Not all logical formulas are useful in the Dijkstra-Hoare programming calculus. Of course, the arities of function symbols and relation symbols must be respected. But there are also conditions on variables and types that must be obeyed.

As to the variables:

Definition 2.4 *We say that a logical formula Q is well-formed or wf in this calculus, if the arities are respected and the following variable condition holds: all logical variables in Q are bound by a quantifier in the formula, whereas all programming variables in Q are free in the formula.*

Note that subformulas of wf formulas need not be well-formed themselves.

As to the types:

Both terms and formulas have a type. As we said before, a ‘type’ (or domain formula) is a representation of some set from the mathematical ‘real world’.

First, we shall describe the types of variables.

Each logical variable in a wf formula has a *type* which is recorded behind the quantifier binding that variable. For example, the occurrences of n in $(\exists n : n \in \mathbf{N} : n < x)$ have \mathbf{N} as their type.

The programming variables in a wf formula, although being free, have a type as well, which is a domain formula denoting a non-empty set, e.g. \mathbf{B} for booleans or \mathbf{Z} for integers. (In principle, also domain formulas like $\{k \in \mathbf{Z} | k > 0 \wedge k < 9\}$ could be used as types for programming variables, but this is not usual.) In order to record the types of programming variables, we add a *context* in front of a wf formula. Such a context is a set of pairs, each pair consisting of a programming variable and its type. Such a set of pairs is usually written as a list, e.g.: $p : \mathbf{B}, x : \mathbf{Z}, y : \mathbf{Z}$. It is assumed that all programming variables in the left hand sides of these pairs are different.

A logical formula is *well-typed* if all of its constituents (including the terms occurring in the formula) are so. (Below, we present a formal definition of well-typedness, related to a notion of derivation.) Subtyping is allowed: for example, a term of type \mathbf{N} is also of type \mathbf{Z} .

We use the notation $wt(\Gamma; Q)$ to express that the well-formed logical formula Q is well-typed in the context Γ . We call $wt(\Gamma; Q)$ a *well-typedness statement*. If Γ is empty, we write $wt(Q)$.

Below, we give the derivation rules for the establishment of well-typedness. Therefore we need a notion of derivability, written as $\Gamma \vdash Q : T$. It expresses that Q has type T in the context Γ . If Γ is empty we write $\vdash Q : T$.

We need the notion $\Gamma \vdash Q : T$ as stepping stone for the establishment of $wt(\Gamma; Q)$. Therefore we have to give *types* to terms and formulas. As regards the terms, we need to have types for constants, function symbols and relation symbols, but also for logical variables, since logical variables may be free in a term or formula Q occurring in $\Gamma \vdash Q : T$. (Recall that $wt(\Gamma; Q)$ implies that all logical variables in the well-formed formula Q are bound; the type of a logical variable is then given inside Q by the quantification.)

Hence, we require that each logical variable has a type *given beforehand*, which will finally (at the end of the derivation with the derivability notion \vdash) match with the type given by the quantification. If this given type of $n \in \mathcal{V}_l$ is T , then we write: $n : T$.

Also, the types of constants, function symbols and relation symbols are assumed to be given beforehand. Again, we write $c : T$ if $c \in \mathcal{C}$ has type T , and so on. There must be a connection with the arities:

- The type of a function symbol f with arity m is always $X_1 \times \cdots \times X_m \rightarrow Y$ for some types X_1, \dots, X_m, Y ,
- The type of a relation symbol R with arity m is always $X_1 \times \cdots \times X_m$ for some types X_1, \dots, X_m .

Definition 2.5 *The derivation rules for establishing well-typedness are the following:*

$$\frac{n \in \mathcal{V}_l, n : T}{\vdash n : T}$$

$$\frac{x \in \mathcal{V}_p, x : T}{x : T \vdash x : T.}$$

$$\frac{c \in \mathcal{C}, x : T}{\vdash c : T}$$

$$\begin{array}{c}
\frac{f \in \mathcal{F}, f : X_1 \times \cdots \times X_m \rightarrow Y, \Gamma_i \vdash s_i : X_i \text{ for } 1 \leq i \leq m}{\bigcup \Gamma_i \vdash f.s_1 \cdots s_m : Y} \\
\frac{R \in \mathcal{R}, R : X_1 \times \cdots \times X_m, \Gamma_i \vdash s_i : X_i \text{ for } 1 \leq i \leq m}{\bigcup \Gamma_i \vdash R.s_1 \cdots s_m : \mathbf{B}} \\
\frac{\varphi \text{ is a formula, } \Gamma \vdash \varphi : \mathbf{B}}{\Gamma \vdash \varphi : \mathbf{B} \text{ for } \in \mathcal{L}_1} \\
\frac{\varphi_1 \text{ and } \varphi_2 \text{ are formulas, } \Gamma_1 \vdash \varphi_1 : \mathbf{B}, \Gamma_2 \vdash \varphi_2 : \mathbf{B}}{\Gamma_1 \cup \Gamma_2 \vdash \varphi_1 \bullet \varphi_2 : \mathbf{B} \text{ for } \bullet \in \mathcal{L}_2} \\
\frac{\varphi \text{ is a formula, } \Gamma \vdash \varphi : \mathbf{B}, n \in \mathcal{V}_l, n : T}{\Gamma \vdash (\otimes n : n \in T : \varphi) : \mathbf{B} \text{ for } \otimes \in \{\forall, \exists\}}
\end{array}$$

Note that, if $x : T' \in \Gamma'$ and $x : T'' \in \Gamma''$ for $x \in \mathcal{V}_p$, then $T' = T''$, since each programming variable has a fixed type (given beforehand).

Now we are ready to formulate the requirement for well-typedness:

Definition 2.6 *If φ is a formula such that $\Gamma \vdash \varphi : \mathbf{B}$ for some Γ and all logical variables in φ are bound, then $wt(\Gamma; \varphi)$.*

It follows that, if $wt(\Gamma; Q)$, then Q is well-formed and the type of Q is always \mathbf{B} .

Examples of well-typedness statements are:

- $wt(x : \mathbf{Z} ; (\exists n : n \in \mathbf{N} : n < x))$,
- $wt(x : \mathbf{Z}, y : \mathbf{Z}, b : \mathbf{B} ; \neg b \Rightarrow ((\exists n : n \in \mathbf{N} : f.n < f.(f.x)) \vee (b \equiv \text{true})))$.

A derivation of the first of these statements is the following:

$$\begin{array}{l}
x : \mathbf{Z} \vdash x : \mathbf{Z}, \\
\quad \vdash n : \mathbf{N}, \\
x : \mathbf{Z} \vdash n < x : \mathbf{B}, \\
x : \mathbf{Z} \vdash (\exists n : n \in \mathbf{N} : n < x) : \mathbf{B}, \\
wt(x : \mathbf{Z} ; (\exists n : n \in \mathbf{N} : n < x)).
\end{array}$$

(Note that $<$ is of type (e.g.) $\mathbf{Z} \times \mathbf{Z}$; by subtyping we have that n is of the appropriate type \mathbf{Z} .)

There is still one remark to be made about the *names* of variables. Different programming variables should have different names, since they are free. But for logical variables one could use the same name for different variables, even if those different variables have different types. For example, a formula like $(\forall k : k \in \mathbf{N} : k > x) \wedge (\exists k : k \in \mathbf{Z} : k < x)$ can hardly be misunderstood. Such a formula cannot be constructed with the above derivation rules. However, with a simple renaming of variables one obtains the desired result.

Now we can express what the basic formula in predicate calculus – a *predicate* – is: it is a well-typed logical formula in some context. I.e.:

Definition 2.7 *P is a predicate if there exists a context Γ such that $wt(\Gamma; P)$.*

Programs

A program is a list of programming formulas, separated by ‘;’. We refer to the literature (see e.g. [M90]) for more information about the syntax of programming formulas. Here we

only give an example:

$x := y; \text{ if } x > 0 \rightarrow x := x - 1 \square x \leq 0 \rightarrow x := x + 1 \text{ fi}$ is a program.

Note that a program contains programming variables such as x , y or b , just like predicates do.

Definition 2.8 A program π is well-typed in the context $x_1 : t_1, \dots, x_n : t_n$, which we denote by $wt(x_1 : t_1, \dots, x_n : t_n ; \pi)$, if all programming variables occurring in π are among x_1 to x_n .

Definition 2.9 A Hoare-triple is a triple $\{Q\}\pi\{R\}$, where both Q and R are well-typed logical formulas (predicates) in a common context Γ , and π is a well-typed program in the same context Γ . Such a Γ is called a context for the Hoare-triple.

Q is the *precondition* of π and R is the *postcondition* of π ; the idea is that the program π , if started while Q holds, will terminate, and upon termination R will hold.

For example, the following is a Hoare-triple:

$\{x > 0\} \ x := x + 2; y := 3 \ \{x > 1 \wedge y > 1\}$.

A context for this Hoare-triple is e.g. $x : \mathbf{Z}, y : \mathbf{Z}$.

3 Predicate calculus

Predicate calculus is the name for a calculational style of working with predicate logic, introduced by E.W. Dijkstra and C.A.R. Hoare for program development. The original approach by these authors was *semantic*; see the next section for a description of the predicate calculus as used in e.g. [D76]. In the present section, we follow the *syntactic* approach, summarizing and extending the work that has been done in e.g. [G81], [AO91] and [GS95a]. Basic syntactic entities in predicate calculus are predicates as described in the previous section. Theorems about these predicates are generated by means of inference rules from axioms.

In this section we discuss the axioms and inference rules that give the machinery for making deductions. These axioms and inference rules tend to look different from the ones traditionally employed in first order logic ([N87], [F91]), since Dijkstra and Hoare give more prominence to equivalence at the expense of implication. It can be proved, however, that this logic is equivalent to classical logic. (This is done in [GS95a] for the propositional logic from [GS93].) The connection may be exploited to import the soundness and completeness of first order logic.

In this section, we use P, Q, R, \dots as meta-variables for predicates.

The deductive system of predicate calculus is based on a number of axioms (or axiom schemes) and a limited number of inference rules.

Axioms

Users of predicate calculus do not bother about the size of their axiom set. Usefulness is more important than economy. One of the axioms is **true**. Axioms are often classified in groups under a common name, e.g. *absorption*, consisting of the pair of axiom schemes $P \wedge (P \vee R) \equiv P$ and $P \vee (P \wedge R) \equiv P$. An *instance* of the first absorption rule is:

$(x > 0 \wedge (x > 0 \vee (b \equiv \text{true}))) \equiv x > 0$.

A remarkable axiom in the class *associativity* is $((P \equiv Q) \equiv R) \equiv (P \equiv (Q \equiv R))$. The validity of this formula is not very well known among logicians. It is, however, a sometimes useful tautology for the equational style of reasoning that is employed in predicate calculus. Examples of axiom schemes about quantified formulas are the so-called *trading rules*: $(\forall i : R \wedge S : P) \equiv (\forall i : R : S \Rightarrow P)$ and $(\exists i : R \wedge S : P) \equiv (\exists i : R : S \wedge P)$. An instance of the first trading rule is: $(\forall i : i > -5 \wedge i < 5 : i^2 < 25) \equiv (\forall i : i > -5 : i < 5 \Rightarrow i^2 < 25)$.

Rules

The main deduction rule for predicate calculus (called *Leibniz's rule* or the *compatibility rule for equivalence*) is the following:

Definition 3.1 *Leibniz's rule is:*

$$\frac{P \equiv Q}{\dots P \dots \equiv \dots Q \dots} .$$

Here $\dots P \dots$ is a formula in which P occurs as a subformula, and $\dots Q \dots$ is the same formula with that occurrence of P replaced by Q . Hence, the derivability of the premiss or *rewrite equivalence* $P \equiv Q$ is used as a justification for the derivability of the conclusion, the left hand side $\dots P \dots$ being *rewritten* into the right hand side $\dots Q \dots$.

Note that this deduction rule (just as the following ones) is essentially a rule *scheme*: only by *substitution* of predicates for the meta-variables one can actually apply these rules.

A second deduction rule is what we call the *equational modus ponens*:

Definition 3.2 *The equational modus ponens rule is:*

$$\frac{P \quad P \equiv Q}{Q} .$$

In this rule, the second premiss ($P \equiv Q$) acts as the rewrite equivalence. Since symmetry: $(R \equiv S) \equiv (S \equiv R)$ is an axiom (scheme), we have as a derived rule (using the equational modus ponens twice):

$$\frac{P \quad Q \equiv P}{Q} .$$

It follows that rewriting is a symmetric operation: if R can be rewritten into S , either by Leibniz's rule or by the equational modus ponens, then S can also be rewritten into R .

The rule which we call the *equational transitivity* is now a derived rule:

$$\frac{P \equiv Q \quad Q \equiv R}{P \equiv R} .$$

We show this as follows (As usual, we write $\vdash P$ for: we have a derivation for P): Assume $\vdash P \equiv Q$ and $\vdash Q \equiv R$. The second assumption and Leibniz's rule give $\vdash (P \equiv Q) \equiv (P \equiv R)$. Combine this with the first assumption, then the equational modus ponens gives $\vdash P \equiv R$.

Derivations

It is the intention that theorems in predicate calculus are exactly the theorems in classical predicate logic (where the programming variables are considered to be constants). Semantically spoken, they are the valid formulas. However, one seldom mentions semantics in predicate calculus. The style is to *postulate* new axiom schemes whenever there appears to be a good use for them. Their semantic validity is not verified and left to the possibly incredulous observer.

The usual *format* of a derivation as applied in predicate calculus is the following:

$$\begin{array}{l} S \\ \equiv \{\text{hint 1}\} \\ T \\ \equiv \{\text{hint 2}\} \\ U \end{array}$$

Here *hint 1* points at the rewrite equivalence used to motivate the rewriting of S into T (or vice versa), leading to the result $\vdash S \equiv T$. The used rewrite equivalence is usually a premiss of Leibniz's rule. *Hint 2* does the same for the rewriting of T into U (or vice versa), leading to $\vdash T \equiv U$. An application of the rule of equational transitivity, also only implicitly present in this format, combines $\vdash S \equiv T$ and $\vdash T \equiv U$ into $\vdash S \equiv U$, the *final conclusion* of the above derivation. (See also section 10, 'Associativity', for the meaning of this proof format in terms of the so-called square brackets.)

A hint can simply be a name of a *class* of equivalences, such as 'absorption' or 'associativity'. It is generally left to the reader to choose the appropriate axiom scheme, the appropriate instantiation and the appropriate subformula being replaced, plus its place of occurrence.

The above format can be used for a derivation of $S \equiv U$ in *two steps*. It is exemplary for a derivation in n steps, $n \geq 1$.

A derivation as presented in the above format can also be seen as a form of *equational reasoning*, where *equivalences* of the form $P \equiv Q$ are the main formulas motivating the steps in a derivation. However, there is also a small place for *implications*. For working with implications, there is a third deduction rule, the (ordinary) *modus ponens*:

Definition 3.3 *The modus ponens rule is:*

$$\frac{P \quad P \Rightarrow Q}{Q} .$$

This rule can be used as follows in the derivation format given above:

$$\begin{array}{l} S \\ \Rightarrow \{\text{hint 3}\} \\ T \end{array}$$

Here *hint 3* points at the reason why $\vdash S \Rightarrow T$.

A derivation with one or more steps in this \Rightarrow -format mixed with ordinary \equiv -steps, leads to a final conclusion of the form $\vdash P \Rightarrow Q$. Similarly, the use of steps with \Leftarrow (with hints why $\vdash S \Leftarrow T$) plus steps with \equiv leads to a final conclusion $\vdash P \Leftarrow Q$, i.e. $\vdash Q \Rightarrow P$.

4 State semantics

The standard semantics for predicates and programs uses the notion of *state* for the interpretation of programming variables. In this section we describe this standard semantics for the syntax which we introduced in section 2.

It turns out that the semantics of a predicate is a boolean function on the set of all interpretations (the 'state space').

The standard model

The standard model for the predicate calculus is as expected. The only syntactic objects for which the standard interpretation is not obvious, are the programming variables. (We discuss the interpretation of programming variables below.) The standard model is based on a domain D which is a *disjoint union* of sets, for example $D = \text{'naturals'} \cup \text{'integers'} \cup \text{'booleans'}$, with standard relations like *'is-equal'* and *'greater-than'* on both *'naturals'* and *'integers'* and relations like *'is-equivalent-to'* on *'booleans'*. Of course, the interpretation function I sends \mathbf{N} to the set *'naturals'*, $>$ to the relation *'greater-than'*, etc. Therefore, we shall not distinguish between the symbol \mathbf{N} and the name *'naturals'* for the set of the natural numbers and we write both as \mathbf{N} . In the same vein, we shall use $>$ both for a relation *symbol* in the predicate calculus and for the actual *relation* *'greater-than'* on *'naturals'* or *'integers'*. Similarly, we identify the symbol 0 with natural or integer number 0 , etc.

Since all logical variables in wf formulas are bound, validity of a wf formula is independent of logical variable assignments, i.e. functions assigning values (in the domain) to logical variables. Hence, although assignment functions are needed for the recursive definition of validity of a formula in a model, we will not bother about the details of these assignment functions for logical variables. (For the interpretation of *programming* variables, see below.)

Moreover, we assume that all function symbols and relation symbols which are being used, have a fixed or a standard interpretation.

The *types* have a standard interpretation as well. For example, the type $S \rightarrow T$ of a function symbol f is interpreted as the *set* $S \rightarrow T$. One assumes that interpretations are *type-correct*, i.e. for term t of type U and interpretation I , it always holds that $I.t$ belongs to (the set corresponding with) U .

For the interpretation of programming variables (recall that these variables are *free* in wf formulas) one uses the word *state* instead of interpretation. Hence, states are functions from \mathcal{V}_p , the set of programming variables, to D . As a consequence of the type-correctness of interpretations, state values are restricted to a subset of D . For example, it holds for the second occurrence of x in the statement $wt(x : \mathbf{Z} ; x < 0)$ that $s.x \in \mathbf{Z}$, for *any* state s .

We use meta-variables s, s' etc. for interpretation functions. As we explain below, the (only) interesting part of an interpretation is its behaviour for programming variables. For example, the value of the logical formula $\neg b \Rightarrow x < 0$ in the statement

$$wt(x : \mathbf{Z}, b : \mathbf{B} ; \neg b \Rightarrow x < 0)$$

depends only on $s.x$ and $s.b$. Assume, for example, that $s.x = 3$ and $s.b = \text{true}$ in model $\mathcal{M} = \langle D, s \rangle$. The value of the formula $x < 0$ in \mathcal{M} is $s.x < 0$, which is **false**. The value of $\neg b \Rightarrow x < 0$ in \mathcal{M} is **false** \Rightarrow **false**, which is **true**.

In the calculational style of predicate calculus, one sometimes prefers to see a state s as an operation symbol operating on formulas which *distributes* over all symbols and sub-formulas of the formula. For example, $s.(\neg b \Rightarrow x < 0) = (s.(\neg b))(s. \Rightarrow)(s.(x < 0)) = \dots = (s.\neg)(s.b)(s. \Rightarrow)(s.x)(s.<)(s.0)$, which is $\neg(s.b) \Rightarrow (s.x) < 0$ because of the standard interpretation connected with $\neg, \Rightarrow, <$ and 0 .

In this manner, the value of a logical formula occurring in a statement can be determined, given a model \mathcal{M} . We do not bother to give the precise definitions. We only introduce some notation: $\models_{\mathcal{M}} wt(\Gamma; Q)$ expresses that Q is type-correct and that the value of Q in \mathcal{M} is

true, where the types of the programming variables in Q are given in Γ . We also say in this case that the statement $wt(\Gamma; Q)$ is *valid* in \mathcal{M} . As is usual in logic, we denote the fact that $wt(\Gamma; Q)$ holds for *all* models, by $\models wt(\Gamma; Q)$, hence by omitting the subscript of \models . In this case we say that that Q is *valid*.

Since a model \mathcal{M} is determined by the interpretation function for programming variables only, one can (given the domain D) identify a model with such an interpretation function s *restricted to* \mathcal{S} , the set of all states or *state space*. Hence, one can also consider (the semantics of) a predicate to be a boolean function on the set \mathcal{S} of all states: Let Q be a predicate with context Γ ; then this boolean function has value *true* for precisely those $s \in \mathcal{S}$ for which $\models_s wt(\Gamma; Q)$.

This is the view of predicates that prevails in semantically-oriented approaches like [D76] and [Bh86].

If one likes to emphasize the boolean function character of a predicate, one can write a predicate Q as the meta-language expression $\lambda_{s \in \mathcal{S}}(s.Q)$, using the function binder λ , as in the lambda calculus.

The semantics of programs

A terminating, deterministic program π can be seen as a *state transformer*: it is a function mapping states to states. For example, the program $x := x + 1$ maps each state s to a state $\pi.s$ which is the same as s for all programming variables except x and such that $\pi.s.x = (s.x) + 1$. The precise behaviour of such a program π as a state transformer can be determined (calculated) by a set of rules which follow the syntax of program construction. For these rules, we refer to the literature (see e.g. [M90]).

We lift such a state transformer (program) π to a function on models: for a model \mathcal{M} with domain D and interpretation function (or state) s , we define $\pi.\mathcal{M}$ to be the model \mathcal{M}' with the same domain as \mathcal{M} , but with interpretation function $\pi.s$ for programming variables.

Given a Hoare-triple $\{Q\}\pi\{R\}$ with context Γ , we can now define when this Hoare-triple is *valid*, denoted $\models \{Q\}\pi\{R\}$:

Definition 4.1 $\models \{Q\}\pi\{R\}$ if for all models \mathcal{M} the following holds: $\models_{\mathcal{M}} wt(\Gamma; Q)$ implies $\models_{\pi.\mathcal{M}} wt(\Gamma; R)$

5 Abstractions from the standard model

Derivations were described in section 3. To every (sub)formula occurring in a derivation, a meaning is given by the state semantics described in the previous section. However, if our only aim is to provide a justification for the axioms and rules of predicate calculus, it is not necessary to go to that amount of detail. For the application of axioms and rules, the internal structure of terms plays no role; hence it is possible, and considerably simpler, to investigate these axioms and rules by means of an interpretation that abstracts from this structure. Two such abstractions are in use: one is based on set theory, the other on algebra.

The view on predicates in these abstractions is also more general than in the syntactic approach of section 2. There, predicates were formulas produced according to certain rules from a given alphabet. This brings about that different formulas describe different predicates.

This is to be contrasted with the abstract approaches that will be described in this section: again, formulas are used to describe predicates, but here it is quite possible that different formulas describe the same predicate.

Set-theoretical abstraction

In this abstraction, the meta-variables for predicates as they occur in the axioms and rules of predicate calculus are taken to denote boolean functions on (or, equivalently, subsets of) some fixed set S . In terms of boolean functions, a predicate Q is equal to the meta-language expression $\lambda_{s \in S}(Q.s)$ (note that, in contrast to the situation in the preceding section, we must now write $Q.s$ rather than $s.Q$). In terms of subsets, Q is a subset of S . Similarly, a set $\{Q.i \mid i \in I\}$ of predicates $Q.i$ indexed by I (e.g.: $I = \mathbf{Z}$), is a set of subsets of S .

The logical operators and quantifiers are defined by lifting, i.e.

$$\begin{aligned} (\forall i : Q.i : R.i) &= \lambda_{s \in S} \forall i \in I (Q.i.s \Rightarrow R.i.s) , \\ \neg Q &= \lambda_{s \in S} (\neg Q.s) . \end{aligned}$$

The logical symbols on the left are the symbols from predicate calculus; the ones on the right are meta-symbols applied in the conventional way to ordinary boolean values. In terms of subsets rather than boolean functions this becomes

$$\begin{aligned} (\forall i : Q.i : R.i) &= \bigcap \{(S \setminus Q.i) \cup R.i \mid i \in I\} , \\ \neg Q &= S \setminus Q . \end{aligned}$$

Observe that the set I , the type of bound variable i , is not explicitly mentioned on the left hand side. Usually it is the default type \mathbf{Z} ; however, in section 9 we shall give an example from [DS90] where this had led to serious problems.

Algebraic abstraction

In [vdW91], [ABHVV92] and [Dij94], predicates are the elements of a fixed complete, completely distributive, complemented lattice. Observe that the subsets of a fixed set S do indeed form such a lattice, with infima and suprema provided by intersections and unions respectively. Hence the algebraic view generalizes the set-theoretical one; below (see section 7) we shall show that this involves a proper generalization.

Denote infima and suprema in the lattice by \sqcap and \sqcup respectively, and the complement by \sim . Then an implication operator may be defined by

$$Q \rightarrow R = \sim Q \sqcup R$$

and an equivalence operator by

$$Q \leftrightarrow R = (Q \rightarrow R) \sqcap (R \rightarrow Q) .$$

The usefulness of the algebraic approach is determined by the fact that, with this definition, operator \leftrightarrow is indeed associative. (For a proof see (62) of [Dij94].)

Most of the treatment in [DS90] can best be understood as a variant of the algebraic method, but there are a few difficulties with this interpretation. In the first place, the authors of [DS90] do not choose \sqcup and \sqcap as the fundamental operators from which the others are defined, but \sqcup and \leftrightarrow ; subsequently \sqcap is defined by

$$Q \sqcap R = Q \leftrightarrow R \leftrightarrow Q \sqcup R .$$

This asymmetry between \sqcup and \sqcap blurs the connection with lattice theory, and indeed the latter is not mentioned in [DS90] except for its dismissal in the preface.

In the second place, the operators are not denoted by the symbols we have been using just now, but by the logical-looking $\wedge \vee \neg \equiv \Rightarrow$. As a consequence, the reader has to be very careful to distinguish between the properties of the objects being studied and that of the logic being used to study them; in particular, he must guard against any tendency to assign the logical connectives their usual interpretation prematurely.

6 The square brackets

The extension of predicate calculus as introduced in [DS90], has not always been clearly understood. The square brackets notation has led to confusion on several occasions, as we will show in the following sections.

In this section, we try to explain the meaning and use of the square brackets.

Extended predicate calculus

In [DS90], the language of predicate calculus is extended: each universal closure of a predicate over all occurring *programming* variables becomes a new *logical* formula. Note that, in the previous sections, all programming variables were free and quantification was only allowed over *logical* variables.

For example, not only $x > y$ is a predicate in the extended predicate calculus of [DS90], but also $\forall x \forall y.(x > y)$. Note that the context does not change by this closure: a typical context for $x > y$ is $x : \mathbf{Z}, y : \mathbf{Z}$, but $\forall x \forall y.(x > y)$ needs a similar context, since x and y are *not typed* in the latter expression (albeit that they are bound).

The notation used for this extension in [DS90] is the *square brackets notation*: instead of $\forall x \forall y.(x > y)$, one writes $[x > y]$. In general, we can say that $[Q]$ is an abbreviation for $\forall \bar{x} Q$, where the \forall -quantifier ranges over the various types of the elements of \bar{x} , the set of all programming variables in Q ; the mentioned types must be given in a context.

In the (uninterpreted) predicate calculus, it is not possible to define the square brackets directly by means of a closed formula, unless the language is extended as suggested above by adding the possibility of universal quantifications over all program variables in a predicate. In an unextended language, square brackets have to appear in axioms and inference rules. The details of how this is to be done have been little explored: at the time of writing, the only proposals known to us are [S94] and [GS95b]. Soundness of axioms and inference rules are generally easy to show. However, completeness for higher order systems is not guaranteed. As a way out for completeness, one usually assumes *finiteness* for the set of program variables and for their values, and hence also for the set of states. Soundness and completeness of the syntax with respect to the semantics have been proved for the logic of [S94] in [Ni94]; in [GS95b] these are derived from more general facts about modal logic. See also [Dij96].

One reason why the square brackets have been so little investigated may be the fact that $[Q]$ is valid – and hence a theorem by completeness – if and only if Q is, as we will explain below, so they may seem redundant. However, their real usefulness becomes clear when they appear within formulas, especially in extensions of predicate calculus with higher order functions. In particular, we mention *predicate transformers* (functions on predicates, see

below), *substitutions* (functions on formulas) and *weakest preconditions* (functions on pairs of predicates and programs).

For instance, while a *predicate transformer* is any function f from and to predicates that satisfies

$$[Q \equiv R] \Rightarrow [f.Q \equiv f.R] \quad , \quad (1)$$

the *punctual* predicate transformers (for an example, see below) are characterized by

$$[(Q \equiv R) \Rightarrow (f.Q \equiv f.R)] \quad . \quad (2)$$

The difference can be phrased in words as follows, using the terminology of the semantic approach. Formula (1) expresses: if the predicates Q and R are equivalent (in every state, either both **true** or both **false**), then the transformed predicates $f.Q$ and $f.R$ are equivalent, whereas (2) says: in every state it holds that, if Q and R have the same value, then $f.Q$ and $f.R$ have the same value.

Without the square brackets, capturing the difference is much more difficult.

It will be clear that *punctual* predicate transformers are, indeed, predicate transformers. The two notions of functions on predicates coincide in many cases. However, in extensions of predicate calculus with higher order functions, punctuality becomes a restriction of the notion ‘predicate transformer’.

Example.

If f is a predicate transformer for which $f.P$ is a formula obtained from P and other predicates by means of negation, conjunction and disjunction, then P is punctual.

For example, the predicate transformer f with

$$f.P = (\neg P \wedge x > 0) \vee (P \wedge y = 1)$$

is punctual, as is not hard to show.

However, if we take f to be

$$f.P = P[x := x^2] \quad ,$$

i.e., f is the result of substituting x^2 for x in P , then f is a predicate transformer which is not punctual. We show this as follows.

The fact that this f is a predicate transformer is obvious. However, f is not punctual, since there exist predicates Q and R and a state s such that $Q \equiv R$ in s , but not $f.Q \equiv f.R$. For example, take $Q = (x \leq 2)$, $R = (x = x)$ and s is a state with $x = 2$. Then $(x \leq 2) \equiv (x = x)$ in s , but *not* $(x^2 \leq 2) \equiv (x^2 = x^2)$ in s .

As a consequence, the weakest precondition wp and the weakest liberal precondition wlp are not punctual for a fixed program. It also turns out, that the square brackets themselves are, as a predicate transformer, not punctual.

State semantics

Let us now give the connection of the square brackets with the semantics of predicate calculus as we explained in section 4. Since there are no free programming variables in $[Q]$, we can define $\mathcal{M} \models wt(\Gamma; [Q])$, for a fixed domain \mathcal{D} , as $\models wt(\Gamma; Q)$, independent of \mathcal{M} .

(Recall that the latter notation means validity for all models.) Consequently (and maybe slightly confusing): $\models wt(\Gamma; [Q])$ iff $\models wt(\Gamma; Q)$, i.e., $[Q]$ is valid iff Q is valid.

We call a predicate Q in context Γ for which $[Q]$ is valid with respect to Γ , a *universal predicate*.

One can express $[Q]$ in words as follows: $[Q]$ holds, iff Q holds *in all states*. That is, $[Q]$ holds iff the programming variables in Q can be considered to be arbitrary constants; their value does not play a role in the establishment of the validity of Q .

Examples of universal predicates are all instances of tautologies, in particular all instances of axioms, e.g.: $(x > 0) \Rightarrow ((y > 0) \Rightarrow (x > 0))$. Another example is $x \geq 0$ in the context $x \in \mathbf{N}$. The validity of the latter predicate is based on *mathematical*, not logical evidence. But $x \geq 0$ is *not* a universal predicate in the context $x \in \mathbf{Z}$.

Note the difference between $[Q]$ and $\models wt(\Gamma; Q)$: the former expression is a *formula* (or rather: a predicate) in the extended predicate calculus, it can be valid or not, dependent of the state and the context; the second expression is an expression in the meta-language, saying that Q is valid with respect to Γ in all states. The latter is equivalent with saying that Q is a universal predicate in context Γ , or that $[Q]$ is *valid* in Γ (i.e. for all states).

Set-theoretical abstraction

With a predicate Q viewed as a boolean function on set S , the definition of the square brackets becomes

$$[Q] = \begin{cases} \lambda_{s \in S} \mathbf{true} & \text{if } \forall_{s \in S} (Q.s = \mathbf{true}) , \\ \lambda_{s \in S} \mathbf{false} & \text{if } \exists_{s \in S} (Q.s = \mathbf{false}) . \end{cases}$$

A consequence of this definition is

$$[Q \equiv R] = \begin{cases} \lambda_{s \in S} \mathbf{true} & \text{if } \forall_{s \in S} (Q.s = R.s) , \\ \lambda_{s \in S} \mathbf{false} & \text{if } \exists_{s \in S} (Q.s \neq R.s) . \end{cases}$$

So $[Q \equiv R]$ equals the constant predicate that is everywhere **true** if and only if functions Q and R take the same value for every argument, i.e., are the same function. It follows that (1) holds for every function f , so in this approach every function from and to predicates is a predicate transformer.

Expressed in terms of subsets of S , the definition of the square brackets becomes much simpler:

$$[Q] = \begin{cases} S & \text{if } Q = S , \\ \emptyset & \text{if } Q \neq S . \end{cases}$$

Algebraic abstraction

With \top short for the top element (the infimum of the empty set) and \perp for the bottom element (its supremum), we can introduce the square brackets by defining

$$[Q] = \begin{cases} \top & \text{if } Q = \top , \\ \perp & \text{if } Q \neq \top . \end{cases}$$

With this definition, we have

$$[Q \leftrightarrow R] = \begin{cases} \top & \text{if } Q = R , \\ \perp & \text{if } Q \neq R , \end{cases}$$

so the lattice element $[Q \leftrightarrow R]$ may be viewed as a representation of the truth value of equality of Q and R . This is the approach taken in [Dij94].

(Note: In [DS90], a theorem of the form $[Q] = \top$ is abbreviated to $[Q]$, or in some cases even to Q . Thus there is no notational distinction between lattice elements and the statements made about them.)

7 A comparison between the styles

In this section, we give several observations comparing the different styles: the state-semantical approach on the basis of the standard model, as described in section 4, the set-theoretical abstraction given in section 5, and the algebraic abstraction also explained in section 5.

1. The set-theoretical approach has the advantage over the standard model that it is possible, for every programming construct π , to introduce a predicate transformer $wp.\pi$ and interpret the Hoare triple

$$\{Q\} \pi \{R\}$$

as the square-bracketed implication

$$[Q \Rightarrow wp.\pi.R] .$$

This approach does not work in the standard model because, as we shall see in the next section, applying wp to a repetition does not give a first order formula.

2. It should be observed that the algebraic approach is more general than the set-theoretical approach. To see this, consider a set S and observe that a subset P of S is a singleton set if and only if for every subset Q of S the inequality

$$P \subseteq Q \not\equiv P \subseteq (S \setminus Q)$$

holds. With this in mind, one may define a *point predicate* as a predicate P satisfying

$$[P \Rightarrow Q] \not\equiv [P \Rightarrow \neg Q]$$

for all predicates Q . In the set-theoretical approach, point predicates certainly exist; they are precisely the singleton subsets of S or, equivalently, boolean functions of the form

$$P = \lambda_{s \in S} (s = \sigma) ,$$

where σ is some element of S . However, there exist models for the algebraic postulates that have no point predicates [MB89, page 29].

3. On the other hand, the set-theoretical approach has the advantage over the algebraic method that some proofs can be considerably simplified by explicit mention of S . As an example, consider the theorem:

$$\begin{aligned} &\text{A punctual predicate transformer } f \text{ is conjunctive (i.e. } f.(X \wedge Y) \equiv f.X \wedge f.Y) \\ &\text{if and only if } f \text{ is monotonic (i.e. if } P \Rightarrow Q \text{ then } f.P \Rightarrow f.Q) . \end{aligned} \quad (3)$$

A general definition of punctuality is given in formula (2) of the previous section; however, in the set-theoretical approach, a predicate transformer f is punctual if and only if there exists a mapping g of type $S \rightarrow \mathbf{B} \rightarrow \mathbf{B}$, where \mathbf{B} denotes the set of boolean values, such that

$$\forall_{s \in S} (f.Q.s = g.s.(Q.s)) \quad (4)$$

for all predicates Q (proof see [Bij93b, Theorem 9]). To prove theorem (3), remark that, by (4), f is conjunctive (or monotonic) if and only if, for every s , boolean function $g.s$ (with type $\mathbf{B} \rightarrow \mathbf{B}$) is conjunctive (or monotonic). Since the number of functions of type $\mathbf{B} \rightarrow \mathbf{B}$ is only 4, it is easy to check by enumeration of cases that conjunctivity and monotonicity coincide for such functions. A proof using only algebraic concepts, on the other hand, requires more than two pages of calculations.

PART II: Misconceptions

8 Mixing the styles

As we showed in the previous sections, there are several essentially different ways in which predicate calculus may be introduced: purely syntactically (as a proof system); semantically; set-theoretically; or algebraically. In this section, we shall say more about the differences between the approaches and try to show how several authors have a tendency to blur the distinction by introducing steps that, properly speaking, belong to one of the other approaches.

We mention seven cases from the literature which lead to difficulties.

1. The book [GS93] presents a treatment of predicate calculus (called ‘equational logic’ there) in terms of axioms and inference rules; the square brackets, however, are not introduced. Wherever they should legitimately occur, their use is circumvented by means of natural language. For example, a formula which is meant to be

$$[P \equiv [Q]]$$

is rendered as

$$P \text{ is valid iff } Q \text{ is valid.}$$

(See page 204 of [GS93].)

Note that this rendering is incorrect; it should be: (P iff (Q is valid)) is valid. Moreover, it obscures the information that P is a constant, which follows from $[P \equiv [Q]]$.

2. Elsewhere, the same formula is rendered as

$$\text{To prove } P, \text{ it suffices to prove } Q.$$

The same criticism applies, but now it is also obscured that we have an equivalence rather than an (inverted) implication. In particular, the latter method is used in the introduction of all program constructs. (For an example, see page 190.)

3. A curious feature of the calculus in [GS93] is the presence of both an axiom and an inference rule called ‘Leibniz’. The inference rule (page 12) reads, verbatim:

$$\frac{X = Y}{E[z := X] = E[z := Y]} \quad .$$

This is the same inference rule that we presented in section 3. (X , Y and E are formulas, z is a variable.)

The axiom (page 60) reads,

$$(e = f) \Rightarrow (E[z := e] = E[z := f]) \quad . \tag{5}$$

(Here e and f take over the roles of X and Y .) Using a terminology introduced above, this states that any formula F is a punctual function of its subexpressions. However, this leads to a contradiction if we admit not only the square brackets, but also Hoare triples and weakest preconditions into formulas. (Note that this is, again, an extension of our notion of formula!)

For instance, let E denote the Hoare triple

$$\{\mathit{true}\} y := 2 \{y = z\} .$$

Then $E[z := y]$ evaluates to **true**, whereas $E[z := 3]$ evaluates to **false**. Applying the axiom now gives

$$(y = 3) \Rightarrow (\mathit{true} = \mathit{false}) ,$$

which is equivalent to $y \neq 3$. Plainly this should not be a theorem; this shows why the authors are unable to regard constructs like Hoare triples as formulas and part of the theory must be developed in a noncalculational metalanguage. On the other hand, if the expression E in (5) is to be chosen from a limited selection, there is no need to introduce this axiom at all since it might then have been proved by induction on the syntax of E . Such a proof is, in fact, spelled out in [D94b].

Had the square brackets been a part of our calculus from the beginning, we could have replaced (5) with

$$[e = f] \Rightarrow [E[z := e] = E[z := f]] ,$$

which is valid regardless of the structure of E .

4. We introduced the name *predicate transformer* for a function f from and to predicates satisfying formula (1) of the previous section. In the context of set-theoretical abstraction, (1) just expresses the extensionality property of set-theoretical functions and is therefore automatically satisfied for all f of the proper type. In the algebraic abstraction the square brackets have been defined in such a way that (1) vacuously holds. However, in the approach using state semantics a predicate is a formula and it is quite possible that distinct predicates Q and R satisfy $[Q \equiv R]$. Thus (1) becomes a genuine proof obligation whenever one wishes to define a predicate transformer. For example, one might define $f.Q$ to be **true** if Q contains an even number of negation symbols, and **false** otherwise. This defines a function from and to predicates, but not a predicate transformer. Reference [G81] uses the semantic approach and defines many predicate transformers, but never mentions the proof obligation.

A less artificial example of this sort is the ‘co-invariant generator’ studied in [BD96].

5. A more serious difficulty appears when one uses predicate transformers to define the semantics of a programming language. For instance, consider the ‘weakest precondition’ semantics of repetition

do $B \rightarrow \pi$ **od**

with respect to postcondition R . In [D76, G81], this semantics is given as

$$(\exists i :: H.i) ,$$

where, for natural i , predicate $H.i$ is defined by

$$\begin{aligned} H.0 &= \neg B \wedge R , \\ H.(i+1) &= H.0 \vee (B \wedge wp.\pi.(H.i)) . \end{aligned}$$

In [DS90] the semantics is expressed as

$$(\forall Q : [(B \vee R) \wedge (\neg B \vee wp.\pi.Q) \equiv Q] : Q) ,$$

where dummy Q ranges over all predicates. Neither of these is a well-formed formula of first order logic, and indeed, no such formula is possible [B86, Proposition 1]. Hence, if one wants to retain weakest precondition semantics in the semantic approach, a more powerful logic than first order logic is needed, and completeness is consequently lost (cf. [BES95]).

6. The authors of [DS90] do not stick to the algebraic approach very rigorously: elements from the semantic approach are admitted whenever this seems opportune. For instance, on page 32, the boolean scalars are defined as the Q for which $[Q] = Q$; subsequently, the theorem that $Q \leftrightarrow R$ is a boolean scalar if Q and R are both boolean scalars is dealt with by the comment that boolean scalars are defined ‘on the trivial space’, a remark that properly belongs to the semantic approach.

7. For another example, remark that [DS90] defines substitution as the result of textual replacement of variable names by expressions, a definition that is not meaningful in the algebraic approach. (Algebraically, a substitution could have been defined as any endofunction on the lattice that is both universally conjunctive and universally disjunctive; see [BS94] on this subject.)

9 Mixing the types

As we saw in section 2, contexts are left implicit in the usual presentation of predicate calculus. Hence, the types of programming variables are not always clear. In this section we give examples from [DS90] where this gives problems.

First, we give definitions and properties of mathematical objects called structures and scalars.

Reconsider the set-theoretical view of predicates, as given in section 6, where predicates are defined as boolean functions on some fixed domain S (the state space). For any set T , a *structure of type T* is by definition a mapping of S into T , with the convention that elements of T are identified with constant functions of S into T .

For binary operator \oplus on T and structures u and v of type T , we define

$$u \oplus v = \lambda_{s \in S} (u.s \oplus v.s) . \tag{6}$$

This makes $u \oplus v$ into a structure mapping S into the codomain of \oplus . In particular, $u = v$ does not express equality of u and v , but denotes a predicate on S .

A *scalar* is a constant function on the state space S . Hence, a scalar u is a special structure of some type T , with the property

$$\forall \sigma \in S \forall \tau \in S (u.\sigma = u.\tau) .$$

The proposition that a structure u is a scalar can be elegantly coded in terms of the square brackets. We show this by using the derivation format described in section 3, taking the liberty to use this format also in the higher order setting of structures. We have

$$\begin{aligned}
& \forall_{\sigma \in S} \forall_{\tau \in S} (u.\sigma = u.\tau) \\
\equiv & \quad \{\text{indirect equality}\} \\
& \forall_{c \in T} \forall_{\sigma \in S} \forall_{\tau \in S} (u.\sigma = c \Rightarrow u.\tau = c) \\
\equiv & \quad \{\text{distribution of antecedent}\} \\
& \forall_{c \in T} \forall_{\sigma \in S} (u.\sigma = c \Rightarrow \forall_{\tau \in S} (u.\tau = c)) \\
\equiv & \quad \{\text{implication from right to left by instantiation } \tau := \sigma\} \\
& \forall_{c \in T} \forall_{\sigma \in S} (u.\sigma = c \equiv \forall_{\tau \in S} (u.\tau = c)) \\
\equiv & \quad \{\text{abstraction}\} \\
& \forall_{c \in T} (\lambda_{s \in S} \forall_{\sigma \in S} (u.\sigma = c \equiv \forall_{\tau \in S} (u.\tau = c))).s \\
\equiv & \quad \{\text{definition of square brackets}\} \\
& \forall_{c \in T} [u = c \equiv [u = c]].s
\end{aligned}$$

for any s . Hence the proposition that u is a scalar is expressed by

$$(\forall c :: [u = c \equiv [u = c]]) \quad , \quad (7)$$

where c ranges over scalars of type T .

In this section, we shall focus our attention upon one of the axioms of [DS90], the *one-point rule*. It states that, for f a function from structures of some type T to predicates on S and v a structure of type T ,

$$[(\forall u : [u = v] : f.u) \equiv f.v] \quad . \quad (8)$$

We show (8) in the derivation format of predicate calculus. Let us assume that u ranges over some set X of structures. Then, for any s :

$$\begin{aligned}
& (\forall u : [u = v] : f.u).s \\
\equiv & \quad \{\text{lifting}\} \\
& \forall_{u \in X} ([u = v].s \Rightarrow f.u.s) \\
\equiv & \quad \{\text{definition of square brackets}\} \\
& \forall_{u \in X} (\forall_{\sigma \in S} (u.\sigma = v.\sigma) \Rightarrow f.u.s) \\
\equiv & \quad \{(1)\} \\
& \forall_{u \in X} (\forall_{\sigma \in S} (u.\sigma = v.\sigma) \Rightarrow f.v.s) \\
\equiv & \quad \{\text{distribution of consequent}\} \\
& (\exists_{u \in X} \forall_{\sigma \in S} (u.\sigma = v.\sigma)) \Rightarrow f.v.s \\
\equiv & \quad \{\text{instantiation } u := v\} \\
& f.v.s \quad .
\end{aligned}$$

1. It has already been observed in section 6 that the quantified formulas of [DS90] do not mention the type of the dummy. We now show that this generates problems in connection with the one-point rule. To demonstrate this, we begin by treating an example.

Example:

Let t be an integer structure (i.e. a structure of type \mathbf{Z}) such that

$$\begin{aligned} & [t = 0 \vee t = 1] \text{ ,} \\ & \neg[t = 0] \text{ ,} \\ & \neg[t = 1] \text{ .} \end{aligned}$$

So t is a mapping of S to $\{0, 1\}$ which is not one of the two constant mappings from S to $\{0, 1\}$. Such a t exists if the state space consists of at least two points, which we assume from now on. Consider the (higher order) formula

$$(\forall u : u = 0 \vee u = 1 : \neg[u = t]) \text{ .} \quad (9)$$

It is not too hard to see that this formula is valid for u ranging over integer scalars, but not for integer structures in general. A formal proof of this in the format of predicate calculus may be found in [Bij93a].

The formal proof of (9) for the case where u ranges over integer scalars makes explicit use of the scalarity of u , viz., to invoke (7). However, the proof of its negation for the case where u ranges over integer structures only consists of trading between the quantification's domain and term, followed by an appeal to the one-point rule (8); it never mentions the status of u . So why is not this derivation valid for the first case as well?

The answer is that the one-point rule (8) as given in [DS90] is not in general valid. Actually, (8) ought to require that either u ranges over all structures, or u ranges over all scalars and v is also a scalar. To show where this condition originates, we mention that the last step in the derivation of (8), as given above, is only allowed if $v \in X$.

2. The condition involving scalarity of u and v seems so be absent in [DS90]. The book does mention, on page 66, that u and v should be 'of the same type'; but an identical warning is given, on page 119, for the *generalized* one-point rule

$$[(\forall u : u = v : f.u) \equiv f.v] \text{ ,} \quad (10)$$

which does hold regardless of scalarity of u and/or v , provided f is punctual, i.e. (2) holds. In the set-theoretical approach, we can show this as follows. With X either the structures or the scalars of some type T we have, for any s ,

$$\begin{aligned} & (\forall u : u = v : f.u).s \\ \equiv & \quad \{\text{lifting}\} \\ & \forall_{u \in X} (u.s = v.s \Rightarrow f.u.s) \\ \equiv & \quad \{(2)\} \\ & \forall_{u \in X} (u.s = v.s \Rightarrow f.v.s) \\ \equiv & \quad \{\text{distribution of consequent}\} \\ & (\exists u \in X (u.s = v.s)) \Rightarrow f.v.s \\ \equiv & \quad \{\text{instantiate } u := \lambda_{\sigma \in S} v.\sigma\} \\ & f.v.s \text{ .} \end{aligned}$$

3. In fact, (10) is used in the proof of the invariance theorem, on page 184 of [DS90], in a case where u ranges over scalars and v is nonscalar. So presumably an integer structure and an

integer scalar are regarded as being of the same type, and the condition for applicability of (8) is indeed incomplete. This state of affairs is particularly unfortunate since the book contains quantifications over both scalars and structures, and usually does not mention explicitly which kind is meant. Worse: on page 119 a quantification over structures is miraculously transformed into one over scalars with no other explanation than ‘dummy renaming’.

Note: Dijkstra’s reaction to the criticism in this section, and a proposal for modification of the rules in [DS90] to deal with it, can be found in [D94a].

10 Mixing language and meta-language

In this section we give two examples where the mixture of language and meta-language has caused problems. In both examples, the difficulties appear because a symbol from the language is also used at a meta-level. In the first case it is the equivalence \equiv which is also used in derivations, in the second example we see that it is not advisable to use the implication symbol \Rightarrow in the meta-language if this symbol is used in the object language, as well.

1. Because \equiv is an associative operator, one may write $P \equiv Q \equiv R$ instead of $(P \equiv Q) \equiv R$ or $P \equiv (Q \equiv R)$. If P , Q and R are themselves lengthy formulas, one may be forced to use a multiline format like

$$\begin{array}{l} P \\ \equiv Q \\ \equiv R \end{array} .$$

However, a derivation

$$\begin{array}{l} P \\ \equiv \quad \{\} \\ Q \\ \equiv \quad \{\} \\ R \end{array} .$$

has the completely different meaning $[P \equiv Q] \wedge [Q \equiv R]$. Observe that it is the presence of the hints, even though they may be empty, that introduces the conjunction and the square brackets. In order to emphasize the difference, some authors [DS90, GS93] use $=$ instead of \equiv in derivations. This, however, does not remove the need to remember the presence of the implicit brackets.

What happens if the distinction between the use of equivalence in predicates and derivations is blurred may be seen in Exercise 1.1(i) of [G81]. There, the reader is asked to evaluate $m = (n \wedge p = q)$ in the state where p has the value T and the other variables have the value F . The answer given in the back of the book reads, verbatim:

$$m = (n \wedge p = q) = F = (F \wedge T = F) = F = (F = F) = F = T = F .$$

The intended parsing of this formula can only be guessed at.

2. We start with a theorem on punctuality. (We recall that the characterization of a punctual predicate transformer is given in (2).) We use the theorem below to show that ‘pushing down’ Hoare-triples to the object language leads to absurdity.

Let f be a predicate transformer. For every predicate P , let predicate transformer $h.P$ be defined by

$$[h.P.T \equiv (P \Rightarrow f.T)] \quad (11)$$

for all predicates T .

Theorem: Assume that $h.P$ is punctual for every P . Then f is punctual.

Proof: For predicates Q and R we have

$$\begin{aligned} & f.Q \Rightarrow f.R \\ \equiv & \quad \{\text{definition of } h, (11)\} \\ & h.(f.Q).R \\ \Leftarrow & \quad \{\text{punctuality of } h.(f.Q)\} \\ & h.(f.Q).Q \wedge (Q \equiv R) \\ \equiv & \quad \{\text{definition of } h, (11)\} \\ & (f.Q \Rightarrow f.Q) \wedge (Q \equiv R) \\ \equiv & \quad \{ \} \\ & Q \equiv R \quad , \end{aligned}$$

so

$$[(Q \equiv R) \Rightarrow (f.Q \Rightarrow f.R)] \quad .$$

Punctuality of f follows by symmetry.

An error sometimes found in textbooks of programming (e.g. [C90, page 83] and arguably [G81, page 109]) is the following. The author introduces the Hoare triple

$$\{Q\} \pi \{R\} \quad (12)$$

as an abbreviation for the implication $Q \Rightarrow wp.\pi.R$, and postulates

$$\{Q\} \pi \{R\} \wedge (R \Rightarrow U) \Rightarrow \{Q\} \pi \{U\} \quad . \quad (13)$$

Note that the second implication symbol in this formula is a meta-level implication. The apparent object language character of the second implication gives undesired consequences: It follows from (13) that

$$R \equiv U \Rightarrow (\{Q\} \pi \{R\} \equiv \{Q\} \pi \{U\}) \quad .$$

Hence, for every precondition Q , Hoare triples are punctual functions of the postcondition. Define $h.P$ by

$$[h.P.T \equiv \{P\} \pi \{T\}] \quad ,$$

i.e.,

$$[h.P.T \equiv (P \Rightarrow wp.\pi.T)] \quad .$$

By the theorem on punctuality given above, it then follows that $wp.\pi$ is punctual for every π . This conclusion is absurd, as we demonstrated in the Example given in Section 6.

The correct way to operate would be to define the Hoare triple (12) as $[Q \Rightarrow wp.\pi.R]$, and replace (13) by

$$\{Q\} \pi \{R\} \wedge [R \Rightarrow U] \Rightarrow \{Q\} \pi \{U\} .$$

The implication in this formula is no longer a meta-level symbol, but an object language symbol. However, the square brackets around $R \Rightarrow U$ prevent absurdities as above.

11 Conclusions

In the first part of this paper, we presented a detailed description of the logical foundations of the Dijkstra-Hoare predicate calculus. In distinguishing the different approaches and comparing their characteristics, we made a clear separation between

- the syntactical style, being a variant of the well-known, deduction-based predicate logic,
- its semantics, which is similar to the usual semantics for first order logic, but for the treatment of programming variables, for which we introduced contexts,
- the set-theoretical abstraction, which treats predicates as subsets of a given set, implying that different formulas can describe identical predicates,
- the algebraic abstraction, being an embedding of predicate logic in lattice theory.

We also positioned the square brackets notation of Dijkstra and Scholten in this framework, thus clarifying what was obscure before.

In the second part of the paper, we demonstrated with a large number of examples how the confusion regarding the logical concepts of predicate calculus has led to erroneous results. We revealed errors as they occur in many loci in the literature and we presented corrections for these errors. Both the identification of fallacies and their repair appeared to be an easy job, due to the explicitness of the foundations of predicate calculus which we achieved in Part I.

Acknowledgements

We like to thank the Eindhoven Tuesday Afternoon Club, Herman Geuvers and Twan Laan for their precise reading of the first part of this paper and for their comments.

References

- [ABHVW92] C.J. Aarts, R.C. Backhouse, P. Hoogendijk, T.S. Voermans, and J.C.S.P. van der Woude, *A relational theory of datatypes*. Eindhoven University of Technology, 1992.
- [AO91] K.R. Apt and A.O. Olderog, *Verification of sequential and concurrent programs*. Springer-Verlag, New York, 1991.

- [B86] R.J.R. Back, 'Proving total correctness of nondeterministic programs in infinitary logic'. *Acta Informatica* **15** (1981), 233–249.
- [BD96] D. Billington and R.G. Dromey, 'The co-invariant generator: an aid in deriving loop bodies'. *Formal Asp. Comput.* **8** (1996), 108–126.
- [BES95] R. Berghammer, B. Elbl and U. Schmerl, 'Formalizing Dijkstra's predicate transformer wp in weak second-order logic'. *Theor. Comp. Sci.* **146** (1995), 185 – 197.
- [Bh86] R.C. Backhouse, *Program construction and verification*. Prentice-Hall International, London, 1986.
- [Bij93a] A. Bijlsma, *A case of context dependence in predicate calculus*. Memorandum AB36. Eindhoven University of Technology, 1993.
- [Bij93b] A. Bijlsma, *Punctuality, conjunctivity, and monotonicity*. Memorandum AB45. Eindhoven University of Technology, 1993.
- [BS94] A. Bijlsma and C.S. Scholten, 'Point-free substitution'. *Sci. Comput. Prog.* **27** (1996), 205 – 214.
- [C90] E. Cohen, *Programming in the 1990s: an introduction to the calculation of programs*. Springer-Verlag, New York, 1990.
- [D76] E.W. Dijkstra, *A discipline of programming*. Prentice-Hall, Englewood Cliffs N.J., 1976.
- [D94a] E.W. Dijkstra, *Our book's omission on quantification over scalar subtypes*. Memorandum EWD1184. The University of Texas at Austin, 1994.
- [D94b] E.W. Dijkstra, *Boolean connectives yield punctual expressions*. Memorandum EWD1187. The University of Texas at Austin, 1994.
- [Dij94] R.M. Dijkstra, *A mathematical approach to logic*. Memorandum rutger20. University of Groningen, 1994.
- [Dij96] R.M. Dijkstra, ' "Everywhere" in predicate algebra and modal logic'. *Inf. Proc. L.* **58** (1996), 237 – 243.
- [DS90] E.W. Dijkstra and C.S. Scholten, *Predicate calculus and program semantics*. Springer-Verlag, New York, 1990.
- [F91] M. Fitting, *First-order logic and automated theorem proving*. Springer-Verlag, New York, 1991.
- [G81] D. Gries, *The science of programming*. Springer-Verlag, New York, 1981.
- [GS93] D. Gries and F.B. Schneider, *A logical approach to discrete math*. Springer-Verlag, New York, 1993.

- [GS95a] D. Gries and F.B. Schneider, 'Equational propositional logic'. *Inf. Proc. L.* **53** (1995), 145–152.
- [GS95b] D. Gries and F.B. Schneider, *Adding the everywhere operator to propositional logic*. Memorandum. Computer Science Department, Cornell University, 1995.
- [H69] C.A.R. Hoare, 'An axiomatic approach to computer programming'. *Comm. ACM* **12** (1969), 576–580, 583.
- [K90] A. Kaldewaij, *Programming: the derivation of algorithms*. Prentice-Hall International, London, 1990.
- [M90] B. Meyer, *Introduction to the theory of programming languages*. Prentice-Hall International, London, 1990.
- [MB89] J.D. Monk and R. Bonnet (eds.), S. Koppelberg, *Handbook of boolean algebras*, vol. I. North-Holland, Amsterdam, 1989.
- [N87] R.P. Nederpelt, *De taal van de wiskunde: een verkenning van wiskundig taalgebruik en logische redeneerpatronen*. Versluys, Almere, 1987.
- [Ni94] P. Nickolas, 'The completeness of functional logic'. *Formal Asp. Comput.* **6** (1994), 39–59.
- [S94] J. Staples, P.J. Robinson, and D. Hazel, 'A functional logic for higher level reasoning about computation'. *Formal Asp. Comput.* **6** (1994), 1–38.
- [vdW91] J.C.S.P. van der Woude, 'Plat-etudes for Carel ende elegance', in: W.H.J. Feijen and A.J.M. van Gasteren (eds.), *C.S. Scholten dedicata: van oude machines en nieuwe rekenwijzen*. Academic Service, Schoonhoven, 1991.

In this series appeared:

93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness, p. 24
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ -calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxonomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π -conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. Verhoeff	The testing Paradigm Applied to Network Structure, p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Computational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β -Reduction in Church's $\lambda \rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall\text{CTL}^*$, $\exists\text{CTL}^*$ and CTL^* , p. 28.
94/25	T. Kloks	$K_{1,3}$ -free and W_4 -free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefănescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π -conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis T. Kloks	On the equivalence covering number of splitgraphs, p. 4.	
94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.	
94/41	T. Kloks D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.	
94/42	J. Engelfriet J.J. Vereijken	Concatenation of Graphs, p. 7.	
94/43	R.C. Backhouse M. Bijsterveld	Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.	
94/44	E. Brinksma R. Gerth W. Janssen S. Katz M. Poel C. Rump	J. Davies S. Graf B. Jonsson G. Lowe A. Pnueli J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.	
94/46	R. Bloo F. Kamareddine R. Nederpelt	The λ -cube with classes of terms modulo conversion, p. 16.	
94/47	R. Bloo F. Kamareddine R. Nederpelt	On Π -conversion in Type Theory, p. 12.	
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.	
94/49	J.C.M. Baeten J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.	
94/50	H. Geuvers	A short and flexible proof of Strong Normalization for the Calculus of Constructions, p. 27.	
94/51	T. Kloks D. Kratsch H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.	
94/52	W. Penczek R. Kuiper	Traces and Logic, p. 81	
94/53	R. Gerth R. Kuiper D. Peled W. Penczek	A Partial Order Approach to Branching Time Logic Model Checking, p. 20.	
95/01	J.J. Lukkien	The Construction of a small CommunicationLibrary, p.16.	
95/02	M. Bezem R. Bol J.F. Groote	Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.	
95/03	J.C.M. Baeten C. Verhoef	Concrete process algebra, p. 134.	
95/04	J. Hidders	An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.	
95/05	P. Severi	A Type Inference Algorithm for Pure Type Systems, p.20.	
95/06	T.W.M. Vossen M.G.A. Verhoeven H.M.M. ten Eikelder E.H.L. Aarts	A Quantitative Analysis of Iterated Local Search, p.23.	
95/07	G.A.M. de Bruyn O.S. van Roosmalen	Drawing Execution Graphs by Parsing, p. 10.	
95/08	R. Bloo	Preservation of Strong Normalisation for Explicit Substitution, p. 12.	
95/09	J.C.M. Baeten J.A. Bergstra	Discrete Time Process Algebra, p. 20	
95/10	R.C. Backhouse R. Verhoeven O. Weber	Mathpad: A System for On-Line Preparation of Mathematical Documents, p. 15	

95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.
95/21	A. van Deursen	Axiomatizing Early and Late Input by Variable Elimination, p. 44.
95/22	B. Arnold, A. v. Deursen, M. Res	An Algebraic Specification of a Language for Describing Financial Products, p. 11.
95/23	W.M.P. van der Aalst	Petri net based scheduling, p. 20.
95/24	F.P.M. Dignum, W.P.M. Nuijten, L.M.A. Janssen	Solving a Time Tabling Problem by Constraint Satisfaction, p. 14.
95/25	L. Feijs	Synchronous Sequence Charts In Action, p. 36.
95/26	W.M.P. van der Aalst	A Class of Petri nets for modeling and analyzing business processes, p. 24.
95/27	P.D.V. van der Stok, J. van der Wal	Proceedings of the Real-Time Database Workshop, p. 106.
95/28	W. Fokkink, C. Verhoef	A Conservative Look at term Deduction Systems with Variable Binding, p. 29.
95/29	H. Jurjus	On Nesting of a Nonmonotonic Conditional, p. 14
95/30	J. Hidders, C. Hoskens, J. Paredaens	The Formal Model of a Pattern Browsing Technique, p.24.
95/31	P. Kelb, D. Dams and R. Gerth	Practical Symbolic Model Checking of the full μ -calculus using Compositional Abstractions, p. 17.
95/32	W.M.P. van der Aalst	Handboek simulatie, p. 51.
95/33	J. Engelfriet and J.J. Vereijken	Context-Free Graph Grammars and Concatenation of Graphs, p. 35.
95/34	J. Zwanenburg	Record concatenation with intersection types, p. 46.
95/35	T. Basten and M. Voorhoeve	An algebraic semantics for hierarchical P/T Nets, p. 32.
96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/04	S. Mauw	Example specifications in phi-SDL.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61

96/14	S.H.J. Bos and M.A. Reniers	The l^2 C-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/16	P. Hoogendijk and O. de Moor	What is a data type?, p. 29.
96/17	E. Boiten and P. Hoogendijk	Nested collections and polytypism, p. 11.
96/18	P.D.V. van der Stok	Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.
96/19	M.A. Reniers	Static Semantics of Message Sequence Charts, p. 71
96/20	L. Feijs	Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.