# Dijkstra-Scholten 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 26, 1998

**Abstract**

The paper focusses on the logical backgrounds of the Dijkstra-Scholten program development style for correct programs. For proving the correctness of a program (i.e. the fact that the program satisfies its specifications), one often uses a special form of predicate calculus in this style of programming. We call this the *Dijkstra-Scholten (DS) predicate calculus*, since [DS90] is the first place in which it is described. DS 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 DS 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 DS predicate calculus: the set-theoretical and the algebraic approach. In doing so, we give DS 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 DS 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 DS 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-Scholten 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, which we call Dijkstra-Scholten or DS (predicate) calculus for two reasons: it has been described in [DS90] for the first time and by using the prefix 'DS' we avoid confusion with classical predicate calculus. We aim in this paper at giving proper definitions of the concepts involved, and we 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 DS 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 DS calculus, relative to a set of typed symbols, including program variables. This leads to the notion of 'predicate'. We also mention well-typed programs and Hoare triples. The deductive system of DS calculus is described in section 3, together with the format in which deductions are presented in the DS calculus style. Its standard semantics (the *state semantics*) is described in section 4.

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

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

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

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

In section 8 we show how the different approaches to DS 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 in the Dijkstra-Scholten style

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 the logic is based which is used in this method for constructing correct programs. 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 [dB80], [G81], [AO91] or [GS95a], since it uses axioms and inference rules. However, we give a more precise characterization of what (predicate-)logical formulas are, splitting the set of variables into logical variables and programming variables. Moreover, we introduce the notion 'well-typedness' in order to give an adequate definition of the kind of predicates used in the Dijkstra-Scholten predicate calculus. Finally, we describe how programs and Hoare triples fit in this setting.

**Predicate-logical formulas**

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

As is usual in logic, predicate-logical formulas have so-called *terms* as constituents, denoting the *objects* represented in the formulas. Terms are constructed from an alphabet, as usual consisting of
– variables and constants, and
– 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 and we usually omit unnecessary parentheses.

Hence, we can give the following recursive definition of *terms*:

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

Terms *are defined by:*
*(1) Each* $n \in \mathcal{V}_l$, $x \in \mathcal{V}_p$, $c \in \mathcal{C}$ *is a term.*
*(2) If* $f \in \mathcal{F}$ *with arity* $n$ *and if* $t_1, \ldots, t_n$ *are terms, then* $f.t_1.\ldots.t_n$ *is a term.*

In the alphabet for the predicate-logical formulas we have three more sets:
– relation symbols,
– logical connectives, and
– logical quantifiers.
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 only. For example, propositional formulas are $(f.n.0 > x) \wedge (y = 0)$ and $\neg(b \equiv \mathtt{true})$. (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.)

The logical quantifiers are the usual ones: $\forall$ and $\exists$. A quantification of a formula consists of prefixing a logical quantifier to the formula, together with a *quantified variable* and a *domain*. The quantified variable, say $n$, *binds* all variables $n$ occurring *free* in the *scope* of the quantifier. We assume that the reader is familiar with the notions of *free-ness* and *bound-ness* of variables and with the scope of a quantifier. For an example, see below.

The domain of a quantifier is a special formula denoting a set. We call these formulas *domain formulas* or *types*; they have a syntax of their own which we shall not give explicitly, since this would force us to describe in detail what the habits are in mathematical language for denoting sets and subsets. We simply assume that types are *imported* in the predicate-logical formulas. Usually, types include

– basic types, like $\mathbf{Z}$ and $\mathbf{B}$, denoting integers and Booleans,

– compound types of the form $T_1 \times \ldots T_n \to T$, denoting $n$-argument functions.

(See below for more comments on types.)

Quantified formulas are written in a special format in DS predicate calculus, 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. The variable $x$ is free in this formula.

One makes a flexible use of this format. This enables one to write 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 type (the domain formula). Note the convention that the domains in such abbreviations are subsets of $\mathbf{Z}$ by default.

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

**Definition 2.2** *The* alphabet of predicate-logical formulas *is the alphabet of the terms extended with a new set $\mathcal{R}$, sets $\mathcal{L}_1$ and $\mathcal{L}_2$ of (unary and binary) logical connectives and the logical quantifiers $\forall$ and $\exists$. There is a fixed* arity function $\mathcal{R} \to \mathbf{N}\backslash\{0\}$.

Predicate-logical formulas *are defined by:*

*(1) If $r \in \mathcal{R}$ with arity $n$ and if $t_1, \ldots, t_n$ are terms, then $r.t_1.\ldots.t_n$ is a predicate-logical formula.*

*(2) If $\varphi$ and $\psi$ are predicate-logical formulas and if $\sim \in \mathcal{L}_1$ and $\bullet \in \mathcal{L}_2$, then $\sim \varphi$ and $\varphi \bullet \psi$ are predicate-logical formulas.*

*(3) If $n \in \mathcal{V}_l$, if $\tau$ is a domain formula and if $t$ is 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.

**More about variables and types**

Not all formulas are useful in the Dijkstra-Scholten predicate calculus: there are conditions on variables and types that must be obeyed.

*As to the variables:*

**Definition 2.3** *We say that a logical formula $Q$ is* well-formed *or* wf *in this calculus, if 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 occurrence of a 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 a program, the types of programming variables are given in so-called *declarations* which have a scope of their own. We assume that the reader is familiar with these matters.

We borrow the notation $t : T$ from type theory, for expressing that term $t$ has type $T$. Similarly, we write $\varphi : \mathbf{B}$ for 'formula $\varphi$ has type $\mathbf{B}$ (the Booleans)'.

The notion of *well-typedness* for terms and formulas follows the syntactic structure, starting with the prescription of types for all alphabet symbols (the *basic typing*). Note that a good definition of well-typedness (for which we only give the outline) brings along that the type of a formula is always $\mathbf{B}$.

For example, the symbols $x$, $f$ (with, say, arity three) and $R$ (with arity two) can have types $\mathbf{Z}$, $\mathbf{Z} \times \mathbf{Z} \times \mathbf{B} \to \mathbf{Z}$ and $\mathbf{Z} \times \mathbf{Z} \to \mathbf{B}$, respectively. Then $f.x.x.\mathtt{true}$ is a well-typed term of type $\mathbf{Z}$ and $R.x.x$ is a well-typed formula of type $\mathbf{B}$. But $f.\mathtt{true}.x.\mathtt{true}$ is *not* well-typed.

Subtyping is allowed: for example, a term of type $\mathbf{N}$ is also of type $\mathbf{Z}$.

**Definition 2.4** *We say that the logical formula $Q$ is* well-typed *if it is well-formed and it can be properly typed with respect to the basic typing.*

(For more details we refer to [BN96].)

Examples of well-typed formulas are (assuming that $x : \mathbf{Z}$, $y : \mathbf{Z}$, $b : \mathbf{B}$ and $f : \mathbf{Z} \to \mathbf{Z}$):
– $(\forall n : n \in \mathbf{N} : n \geq x)$,
– $\neg b \Rightarrow ((\exists n : n \in \mathbf{N} : f.n < f.(f.x)) \vee (b \equiv \mathtt{true}))$.

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

**Definition 2.5** *$P$ is a* predicate *if $P$ is a well-typed formula (with respect to the basic typing).*

**Programs**

We refer to the literature (see e.g. [M90]) for more information about the syntax of programming formulas. Here we only give an example:
$x := y$; **if** $x > 0 \to x := x - 1$ ▯ $x \leq 0 \to x := x + 1$ **fi**

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

**Definition 2.6** *A program $\pi$ is* well-typed, *if all programming variables occurring in $\pi$ have a type in the basic typing.*

Such a basic typing for programming variables is often given in a declaration; e.g.: $\underline{int}\ x, y$ expresses that $x : \mathbf{Z}$ and $y : \mathbf{Z}$.

Correctness of a program can be deduced by means of so-called *Hoare triples*:

**Definition 2.7** *A* Hoare triple *is a triple $\{Q\}\pi\{R\}$, where both $Q$ and $R$ are well-typed logical formulas (predicates) with respect to a basic typing, and $\pi$ is a well-typed program with respect to the same basic typing.*

Originally, Hoare triples refer to partial correctness (programs may not terminate). We give here the interpretation for total correctness: $Q$ is the *precondition* of $\pi$ and $R$ is the *post-condition* of $\pi$; the idea is that the program $\pi$, if started while $Q$ holds, will terminate, and upon termination $R$ will hold.

For example, the following is a Hoare triple, if $x : \mathbf{Z}$ and $y : \mathbf{Z}$ in the basic typing:
$\{x > 0\}\ \ x := x + 2;\ y := 3\ \ \{x > 1 \wedge y > 1\}$.

# 3 Dijkstra-Scholten predicate calculus

E.W. Dijkstra and C.S. Scholten use the name 'predicate calculus' for a calculational style of working with predicate logic, meant for program development (see [DS90]). The original approach by these authors was *semantic*; see the next section for a description of the DS 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 DS 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 Scholten 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 $P, Q, R, \ldots$ as meta-variables for predicates.

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

**Axioms**
Users of DS 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: $(x > 0 \wedge (x > 0 \vee (b \equiv \texttt{true})) \equiv x > 0)$.

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 DS 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 DS calculus (called *Leibniz's rule* or the *compatibility rule for equivalence*) is the following:

**Definition 3.1** Leibniz's rule *is:*

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

Here $\ldots P \ldots$ is a formula in which $P$ occurs as a subformula, and $\ldots Q \ldots$ 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 $\ldots P \ldots$ being *rewritten* into the right hand side $\ldots Q \ldots$.

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:

$\frac{Q \quad P \equiv Q}{P}$ . This can be shown as follows (As usual, we write $\vdash P$ for: we have a derivation for $P$): Assume $\vdash Q$ and $\vdash P \equiv Q$. The second assumption gives, by the use of the equational modus ponens and the instance $\vdash (P \equiv Q) \equiv (Q \equiv P)$ of the symmetry axiom: $\vdash Q \equiv P$. Then, using the equational modus ponens again, from $\vdash Q$ and $\vdash Q \equiv P$ we infer $\vdash P$.

It follows that rewriting is a symmetric operation: if $R$ can be rewritten into $S$, then $S$ can also be rewritten into $R$.

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

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

We show this as follows: 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**

A *theorem* is, as usual, a formula derivable from the axioms by zero or more applications of

the deduction rules. It is the intention that theorems in DS calculus are exactly the theorems in classical predicate logic (where the programming variables are considered to be constants). Semantically spoken, they are the (universally) valid formulas. However, one seldom mentions semantics in DS 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 DS calculus is the following:

$S$
$\equiv$ {hint 1}
$T$
$\equiv$ {hint 2}
$U$

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, **1**, 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 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:

$S$
$\Rightarrow$ {hint 3}
$T$

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 establishing the meaning of programming variables. In this section we describe this standard semantics for the syntax which we introduced in section 2. In this approach, the semantics of a predicate is basically a Boolean function on the 'state space'.

**The semantics of terms and formulas**

The standard semantics for the DS calculus leaves no choice for the 'meaning' of the majority of the symbols occurring in terms and formulas. They act as constants. The only syntactic objects for which the 'meaning' may vary, are non-constant function and relation symbols and the programming variables. We give a short overview of the standard semantics.

The *domain* $D$ (also called the *semantic data domain* or the *domain of interpretation*) is a *disjoint union* of sets, for example $D = \mathbf{N} \cup \mathbf{Z} \cup \mathbf{B}$, being the union of the sets of naturals, integers and Booleans. All alphabet symbols occurring in terms and formulas, except the (logical and programming) variables, are *interpreted* in $D$ by an *interpretation* (mapping) $I$. This interpretation $I$ sends constants to constant values in $D$, function symbols and relation symbols to (set-theoretical) function and relations (with corresponding arities) over $D$. Many constants, function symbols and relation symbols have a fixed interpretation. For example, $I$ sends the symbol '0' to the (natural or integer) number 0, the relation symbol '$>$' to the relation $>$ (greater than) on $\mathbf{N}$ or $\mathbf{Z}$ and the logical symbol '$\wedge$' to the Boolean function $\wedge$ (conjunction) on $\mathbf{B} \times \mathbf{B}$.

The standard interpretation for *types* also behaves naturally, e.g., it sends the *type* $\mathbf{N}$ to the *set* $\mathbf{N}$, etc. Moreover, the type $S \to T$ of a unary function symbol $f$ is interpreted as the *set* $S \to T$, etc. One assumes that interpretations are *type-preserving*, e.g. for term $t$ of type $U$ and interpretation $I$, it always holds that $I.t$ belongs to (the set corresponding with) $U$.

The *semantic structure* or *model* $\mathcal{M} = (D, I)$ is extended with a *valuation* of variables (also called *variable assignment*) for the logical and programming variables occurring in terms and formulas, giving *values* in $D$ to the variable symbols. Hence, a valuation is a mapping from $\mathcal{V}_l \cup \mathcal{V}_p$ to $D$. Such a valuation is usually called a *state* in programming semantics. Like interpretations, states are type-preserving, e.g. for $x$ of type $V$ it holds that $s.x$ is an element of the corresponding set $V$.

The pair of a semantic structure $\mathcal{M}$ and a state $s$ determines the 'meaning' or *data value* of each term or formula. For a term $t$, this data value is denoted $\mathcal{M}[\![t]\!](s)$, being an element of the domain $D$. (Note that the name $\mathcal{M}$ of the semantic structure is 'overloaded' here: it is also used as a function symbol.) For a well-typed formula $P$, the data value $\mathcal{M}[\![P]\!](s)$ is a Boolean value, expressing whether $P$ is true or false in $\mathcal{M}$ with respect to $s$. We write $\mathcal{M}, s \models P$ iff $\mathcal{M}[\![P]\!](s) = \texttt{true}$.

Both for terms $t$ and well-typed formulas (predicates) $P$, the data values $\mathcal{M}[\![t]\!](s)$ or $\mathcal{M}[\![P]\!](s)$ can be determined by a set of obvious rules, following the syntactic structure of the term or formula. We refer to the literature for details. (See e.g. [AO91].)

Semantic structure $\mathcal{M}$ is called a *model of* predicate $P$, abbreviated $\mathcal{M} \models P$, if $\mathcal{M}[\![P]\!](s)$ is `true` for all states $s$. A predicate $P$ is *(universally) valid* if all semantic structures $\mathcal{M}$ are models for $P$. (We come back to universally valid predicates in Section 6.)

Example: Let $P$ be the predicate $f.x \geq 0$, let $\mathcal{M}$ be a semantic structure with $\mathbf{Z} \subseteq D$,

let $x \in \mathcal{V}_p$, $x : \mathbf{Z}$ and assume $I(f) = \lambda_{u \in \mathbf{Z}}(u)$ (the identity function). Let $\mathcal{M}'$ be the same semantic structure, but with $I(f) = \lambda_{u \in \mathbf{Z}}(u^2)$ (the square function). Then $\mathcal{M}[\![P]\!](s) = \mathbf{true}$ for states $s$ with $s.x = 1$ and **false** for states $s$ with $s.x = -1$. Hence, $\mathcal{M}$ is not a model of $\varphi$. However, $\mathcal{M}' \models \varphi$.

Since all logical variables in wf formulas are bound, the data value of a wf formula is independent of the data values of logical variables. (This is an easily provable theorem.) Hence, although valuations for logical variables are needed for the recursive definition of the data value of a term or a formula, we need not bother about this valuation for logical variables occurring in wf formulas. So, since predicates are wf formulas, we may conceive a state as a function from $\mathcal{V}_p$, the set of programming variables only, to $D$, as is usually done in programming semantics. (In many applications, *all* function and relation symbols are constant. The data values then only depend on the state mapping $s : \mathcal{V}_p \to D$.)

We recall that *predicates* are defined relative to some basic typing: formula $P$ is a predicate if it can be well-typed with respect to a basic typing (Definition 2.5). Assume that the formula $(b \Rightarrow (\exists n : n \in \mathbf{N} : n < x)) \wedge (\neg b \Rightarrow x < 0)$ has been shown to be a predicate ($x : \mathbf{Z}$ and $b : \mathbf{B}$ are given in the basic typing). Then the value of this predicate depends only on $s.x$ and $s.b$ (and not on $s.n$). Moreover, by type-preservation, $s.x \in \mathbf{Z}$ and $s.b \in \mathbf{B}$.

In this manner, the value of each predicate (relative to a basic typing) can be determined, given a semantic structure $\mathcal{M}$ and a valuation (state) $s$.

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

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

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 $\lambda$, as in the lambda calculus.

**The semantics of programs**

A deterministic, terminating program $\pi$ 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 deterministic, terminating program $\pi$ 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]).

Given a basic typing, a semantic structure $\mathcal{M}$ and a Hoare triple $\{Q\}\pi\{R\}$, we can now define total correctness by expressing when this Hoare triple is *valid* with respect to $\mathcal{M}$, denoted $\mathcal{M} \models \{Q\}\pi\{R\}$:

**Definition 4.1** $\mathcal{M} \models \{Q\}\pi\{R\}$ *if for all states* $s$ *the following holds:* $\mathcal{M}, s \models Q$ *implies* $\mathcal{M}, \pi.s \models R$.

(For another definition of total correctness of Hoare triples and for the definition of partial correctness, see [AO91], p. 65.)

# 5 Abstractions from the state semantics

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 DS 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 (syntactically) different formulas are 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 DS 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 $s.Q$, since a predicate $Q$ is here a function on $S$). In terms of subsets, $Q$ is a subset of $S$. Similarly, a set $\{Q.i \mid i \in \mathbf{I}\}$ of predicates $Q.i$ indexed by $\mathbf{I}$ (e.g.: $\mathbf{I} = \mathbf{Z}$), is a set of subsets of $S$.

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

$$\begin{array}{rcl} (\forall i : Q.i : R.i) & = & \lambda_{s\in S}\forall_{i\in\mathbf{I}}(Q.i.s \;\Rightarrow\; R.i.s) \quad , \\ \neg Q & = & \lambda_{s\in S}(\neg Q.s) \quad . \end{array}$$

The logical symbols on the left are the symbols from DS 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

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

Observe that the set $\mathbf{I}$, 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], [ABHVW92] 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

$$Q \to R \;=\; \sim Q \sqcup R$$

and an equivalence operator by

$$Q \leftrightarrow R \;=\; (Q \to R) \sqcap (R \to Q) \quad .$$

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

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

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 $\wedge \vee \neg \equiv \quad \Rightarrow$. 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, the reader must guard against any tendency to assign the logical connectives their usual interpretation prematurely.

## 6 The square brackets

The extension of DS 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 DS predicate calculus**

In [DS90], the language of DS 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 DS calculus of [DS90], but also $\forall x \forall y.(x > y)$. Note that, for programming variables, we still need a basic typing: both in $x > y$ and in $\forall x \forall y.(x > y)$, the programming variables $x$ and $y$ are *not explicitly typed* in the formula (albeit that they are bound in the latter predicate).

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 the 'universal closure' $\overline{\forall x} Q$, where the $\forall$-quantifier ranges over the various types of the elements of $\overline{x}$, the set of all programming variables in $Q$; the mentioned types must be given in a basic typing.

In the (uninterpreted) DS 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 $\mathcal{M}$ is a model of $[Q]$ if and only if $\mathcal{M}$ is a model of $Q$, as we will explain below, so the square brackets may seem redundant. However, their real usefulness becomes clear when they appear within formulas, especially in extensions of DS 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

$$[Q \equiv R] \;\Rightarrow\; [f.Q \equiv f.R] \quad , \tag{1}$$

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

$$[(Q \equiv R) \;\Rightarrow\; (f.Q \equiv f.R)] \quad . \tag{2}$$

(The notion of punctuality is used to simplify proof obligations for programs. For example, in order to prove that the assertion $P \wedge R \equiv Q \wedge R$ is correct at a certain place in a program, it is sufficient to prove $P \equiv Q$ at that place, since the predicate transformer that maps $X$ to $X \wedge R$ is punctual.)

The difference between (1) and (2) 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.

Clearly, (2) implies (1), i.e. all punctual predicate transformers are (indeed) predicate transformers, but (1) does not always imply (2) (see below).

Without the square brackets, capturing the difference between punctual and non-punctaual predicate transformers is much more difficult.

The two notions of functions on predicates coincide in many cases. However, in extensions of DS 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

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

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 DS predicate calculus as we explained in section 4.

It is natural to define $\mathcal{M}[\![\ [Q]\ ]\!](s) \equiv \texttt{true}$ as: $\mathcal{M}[\![Q]\!](s) \equiv \texttt{true}$ *for all states* $s$. The latter is denoted by: $\mathcal{M} \models Q$ (see section 4). Since there are no free programming variables in $[Q]$, there is no difference between $\mathcal{M}, s \models [Q]$ (i.e., $\mathcal{M}[\![\ [Q]\ ]\!](s) \equiv \texttt{true}$ in state $s$) and $\mathcal{M} \models [Q]$ (i.e., $\mathcal{M}[\![\ [Q]\ ]\!](s) \equiv \texttt{true}$ in all states). Consequently, $\mathcal{M}$ is a model of $[Q]$ iff $\mathcal{M}$ is a model of $Q$.

One can express $[Q]$ in words as follows: $[Q]$ holds with respect to $\mathcal{M}$, iff $Q$ holds *in all states* with respect to $\mathcal{M}$. 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 $Q$.

Given a basic typing, we can decide whether a predicate $Q$ is universally valid (i.e., whether all semantic structures are models of $Q$, see Section 4 again). As we saw above, this is the same question as whether $[Q]$ is universally valid.

Examples of universally valid predicates (or *universal predicates*, as they are called) are all instances of tautologies, in particular all instances of axioms, e.g.: $(x > 0) \Rightarrow ((y > 0) \Rightarrow (x > 0))$. There are also universal predicates which hold on mathematical, not logical evidence, for example the predicate $(f(x) > 1) \Rightarrow (f(x) > 0)$. Note that the universal character of a predicate may depend on the basic typing: $x \geq 0$ is universally valid if $x : \mathbf{N}$, but not if $x : \mathbf{Z}$.

Note the difference between $[Q]$ and $\mathcal{M} \models Q$: the former expression is a *formula* (or rather: a predicate) in the extended DS calculus, it can hold or not, dependent of the semantic structure $\mathcal{M}$; the second expression is an expression in the meta-language, saying that $Q$ *does* hold with respect to $\mathcal{M}$. The latter is equivalent with saying that $Q$ is a universal predicate.

**Set-theoretical abstraction**

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

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

A consequence of this definition is

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

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:

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

**Algebraic abstraction**

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

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

With this definition, we have

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

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 semantics, 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 semantics that it is possible, for every programming construct $\pi$, 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 semantics 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 $\sigma$ 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:

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

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 $S \to \mathbf{B} \to \mathbf{B}$, where $\mathbf{B}$ denotes the set of Boolean values, such that

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

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 $\mathbf{B} \to \mathbf{B}$) is conjunctive (or monotonic). Since the number of functions of type $\mathbf{B} \to \mathbf{B}$ 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 DS 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 DS 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

$P$ is valid iff $Q$ 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, for instance in the proof method for IF on page 188, the same formula is rendered as

To prove $P$, 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.

**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 $E$ 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

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

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

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

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

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

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 section 6. 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

$$\textbf{do} \ B \rightarrow \pi \ \textbf{od}$$

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

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

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

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

In [DS90] the semantics is expressed as

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

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]).

One can, however, continue to do program derivation in practice using only first order logic. Here the trick is never to calculate the weakest precondition of a repetition, but to prove properties of it with the aid of the main invariance theorem (cf. [DS90, Theorem (9,26)]).

**6.** The following observations show that the authors of [DS90] do not stick very rigorously to the algebraic approach. This does not lead to real problems, but it may be confusing for the reader.

(a) 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.

(b) For another example, note 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, the basic typing of the symbols from the language is left implicit in the usual presentation of DS 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

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

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

$$\forall_{\sigma \in S} \forall_{\tau \in S}\,(u.\sigma = u.\tau) \quad .$$

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

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

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

$$(\forall c :: [u = c \;\equiv\; [u = c]]) \quad , \tag{7}$$

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 $T$,

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

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

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

**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

$$
\begin{array}{l}
[t = 0 \;\vee\; t = 1] \quad , \\
\neg[t = 0] \quad , \\
\neg[t = 1] \quad .
\end{array}
$$

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

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

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 DS 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

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

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 $s$,

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

**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]. The proposal involves the introduction of two new axioms, viz.

$$(\forall c :: (\exists x :: [x = c]))$$

and

$$(\forall x :: [(\exists c :: c = x)]),$$

where $x$ ranges over structures of a type $T$ and $c$ ranges over scalars of the same type $T$.

## 10 Mixing language and meta-language

In this section we give two examples where the mixture of language and meta-language has caused problems. Of course, using language *and* meta-language in a text is quite common and even convenient in many applications. But one has to be careful that the 'overloading' of a symbol does not lead to confusion. In both examples below, difficulties appear because a symbol from the language is too lightly 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

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

However, a derivation

$$\begin{array}{lll} & P & \\ \equiv & & \{\} \\ & Q & \\ \equiv & & \{\} \\ & R & \end{array}$$

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:

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

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

$$[h.P.T \ \equiv \ (P \Rightarrow f.T)] \tag{11}$$

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

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

so

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

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

$$\{Q\} \ \pi \ \{R\} \tag{12}$$

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

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

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

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

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

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

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 $\pi$. 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

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

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-Scholten 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 and types,
- 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 DS 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 DS 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 E.-R. 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.

[dB80] J.W. de Bakker, *Mathematical Theory of Program Correctness*. Prentice-Hall International, Englewood Cliffs, NJ, 1980.

[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.

[BN96] L. Bijlsma and R. Nederpelt, 'Predicate calculus: concepts and misconceptions', *Computing Science Reports* **96-21** (1996), Department of Computing Science, Eindhoven University of Technology.

[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.