Report Number: WUCS-89-37

1989-09-27

The Specification Statement Refined

Authors: Wei Chen and Jan Tijmen Udding

In this paper, we present a rigorous treatment of so-called logical constants, which are used to relate the values of program variables between the precondition and the postcondition of a program. In order to do so, we generalize the latest proof rule for procedures and give a new definition for the specification statement. We show that the specification statement with this definition is the greatest lower bound of all its implementations under the usual refinement ordering and that it is A-distributive. We also demonstrate that a previous treatment of logical constants in specification statements does not have these properties.

THE SPECIFICATION STATEMENT REFINED

Wei Chen
Jan Tijmen Udding

WUCS-89-37

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

The Specification Statement Refined

Wei Chen
Jan Tijmen Udding

WUCS-89-37

The Specification Statement Refined

Wei Chen

Jan Tijmen Udding

Department of Computer Science

Washington University

Campus Box 1045

St. Louis, MO 63130

September 27, 1989

Summary: In this paper, we present a rigorous treatment of so-called logical constants, which are used to relate the values of program variables between the precondition and the postcondition of a program. In order to do so, we generalize the latest proof rule for procedures and give a new definition for the specification statement. We show that the specification statement with this definition is the greatest lower bound of all its implementations under the usual refinement ordering and that it is $\wedge$-distributive. We also demonstrate that a previous treatment of logical constants in specification statements does not have these properties.

0 Introduction

The construction of a program usually starts with its specification, proceeds through a number of refinements, and ends with the desired program. This

method of stepwise refinement was pioneered by Dijkstra [4] and followed by others, e.g. [5]. A theoretical development in this direction [1, 8, 6] treats a specification as a program construct called a specification statement, thus providing a uniform view of a specification and a program. Refinement can now be viewed as a partial order on programs.

Usually, so-called logical constants appear in a specification statement, where they merely communicate values of program variables between precondition and postcondition. They are not allowed to appear in any implementation and thus differ from program constants. The semantics of a specification statement should reflect this nature of logical constants. In [7] Morgan and Gardiner introduce a so-called conjunction statement to handle logical constants in a specification statement. But, as we make clear in the sequel, their definition can be improved upon. In the following we present a new definition for the specification statement that deals with logical constants. We show that it is the most general definition possible, a property which we call *sharpness*, following Bijlsma, Matthews and Wiltink [3]. Our work is based on their proof rule for procedures, which sharpens Gries's proof rule for procedures [5]. The new definition of the specification statement is a generalization of their proof rule, as the definition of Morgan and Gardiner is a generalization of Gries's proof rule.

Notions and notations. We use T to denote the predicate that is true everywhere and F to denote the predicate that is false everywhere. For a predicate P, $[P]$ denotes the universal quantification of P over all program variables. We view a program S as a predicate transformer that maps any predicate R into the weakest precondition of S, denoted by $S.R$. The Hoare triple $\{ P \}\ S\ \{ Q \}$ thus means $[P \Rightarrow S.Q]$. Formally, a program S_0 can be refined to S_1, denoted by $S_0 \sqsubseteq S_1$, if $[S_0.R \Rightarrow S_1.R]$ for all predicates R. We also say that S_1 is a *refinement* of S_0. When, in addition, S_1 is an executable program, i.e. when it does not contain a specification statement, S_1 is said to *implement* S_0, denoted by S_1 **impl** S_0.

1 The Specification Statement

As in [1, 8, 6] we consider programs in Dijkstra's guarded programming language [4] to be executable and extend the language with a so-called specification statement, which is not executable. In its elementary form the specification statement represents a statement operating on a variable, say

v, whose execution reaches a state satisfying a certain postcondition, say *post*, when starting in a certain precondition, say *pre*. Nothing is specified in case the statement execution starts in a state not satisfying *pre*; it may terminate in any state or not terminate at all. We denote such a statement by $v : [pre, post]$. In order to define its semantics, we define for any postcondition R the condition that v must satisfy for the statement to end up in a state satisfying R.

First of all, it is clear from this informal explanation that we cannot guarantee anything if we do not start in a state v satisfying *pre*. Therefore, assume that v is a state satisfying *pre*. Now v is a proper starting state if any final state satisfies R. The possible final states when starting in v are characterized by *post*. Hence, v is a proper starting state if $(\forall v :: post \Rightarrow R)$.

Putting the pieces together we get as the definition of the elementary specification statement

$$(v : [pre, post]).R \;=\; pre \wedge (\forall v :: post \Rightarrow R) \qquad (0)$$

for any postcondition R. This is exactly the definition in [6] and [8].

Usually, in addition to program variable v, *pre* and *post* contain other free variables, called constants. In this sense a program is generic. These constants come in two types, program constants and logical constants. An implementation is allowed to refer to program constants, although it should not change them. The purpose of logical constants is to relate the initial value of v to its final value. In its elementary form the specification statement does not distinguish the two types but treats all constants as program constants, as we show in the next example.

Example 0 Let v be an integer program variable. We consider the specification statement $v : [v = V, v = V + 1]$. Using (0), we derive

$$(v : [v = V, v = V + 1]).R \;=\; (v = V) \wedge R^v_{V+1}$$

which allows both $v := v + 1$ and $v := V + 1$ as a refinement.

An implementation of a specification statement with a logical constant is not allowed to use that constant anywhere. Interpreting V as a logical constant, the only implementation allowed would be $v := v + 1$. □

The example shows that we have to indicate explicitly whether a constant is logical, for which we introduce the following syntactic construct. If V is

to be interpreted as a logical constant we write the specification statement as

$$v : V\,[pre, post]$$

Example 1 Taking the same pre- and postcondition as in the previous example, but now interpreting V as a logical (integer) constant, the specification statement $v : V\,[v = V, v = V + 1]$ stands for a whole range of specification statements. For example, it states that if $v = 0$ initially then its final value will be $v = 1$, and if $v = 5$ initially then its final value will be $v = 6$. Basically, this specification statement says that v's value will be increased by 1.

For the analysis of logical constants it is interesting to notice what the specification statement allows as possible final states when starting in a state satisfying $v = V$. If V is interpreted as a program constant then only one final state is possible, viz. the state satisfying $v = V + 1$. If V is interpreted as a logical constant then the final state can be any integer. □

In [7] the conjunction statement, $|[\mathbf{con}\ l \bullet S]|$, is introduced to accommodate logical constants.

In order to define the semantics of this more general specification statement, we proceed as in the elementary case. Let R be the postcondition for which we want to determine all initial states v that are guaranteed to lead to a state satisfying R. If we start in a state v for which no value of V exists such that *pre* holds the specification statement does not guarantee anything, so we can not guarantee termination in R. Therefore, assume that the initial state v satisfies $(\exists\, V :: pre)$.

Since R should hold in any possible final state in order for v to be a proper starting state, we first characterize the set of possible final states when starting in v. In order to distinguish initial and final values we express the final states with a fresh variable s. Rather than characterizing the possible final states we characterize the states s that cannot possibly be a final state of the specification statement. The intention of the specification statement is to guarantee a final state satisfying *post* provided that we start in a state satisfying *pre*, no matter what value V has. Therefore, if we can find, for a final value s, a value of V such that *pre* holds, which means that we can successfully start, and such that $post^v_s$ does not hold, then s cannot possibly

be a final value. Turning the argument around, we cannot exclude s as a final value if $(\forall V :: pre \Rightarrow post_s^v)$.

Example 2 Consider the specification statement

$$v : V\,[v = V \vee v = V + 1, v = V + 1 \vee v = V + 2].$$

Given that we start in a state v satisfying $(\exists V :: v = V \vee v = V + 1)$, the predicate characterizing the set of final states is

$$(\forall V :: (v = V \vee v = V + 1) \Rightarrow (s = V + 1 \vee s = V + 2)).$$

This can be simplified to $(s = v + 1 \vee s = v + 2) \wedge (s = v \vee s = v + 1)$ which reduces to $s = v + 1$. □

Initial state v now guarantees termination in a state satisfying R if all possible final states imply R. Putting the pieces together again, we define the semantics of the general specification statement to be the following.

Definition 0 For any predicate R we define $(v : V\,[pre, post]).R$ to be

$$(\exists V :: pre) \wedge (\forall s :: (\forall V :: pre \Rightarrow post_s^v) \Rightarrow R_s^v).$$

□

Example 3 Taking the specification statement

$$v : V\,[v = V \vee v = V + 1, v = V + 1 \vee v = V + 2]$$

again, we find for postcondition R the weakest precondition R_{v+1}^v, as was to be expected. □

This definition of the specification statement differs from the definition of the conjunction statement in [7]. We come back to this difference after the next section in which we prove the suitability of this definition and a certain sharpness property. It can easily be seen that the above definition observes the following healthiness conditions, as do other constructs in the guarded command language.

Law of $\wedge$-distributivity: $[S.(\forall d :: R) = (\forall d :: S.R)]$
when d is neither a program variable nor a program constant of S.

Law of (semi)-$\vee$-distributivity: $[(\exists d :: S.R) \Rightarrow S.(\exists d :: R)]$
when d is neither a program variable nor a program constant of S.

Law of monotonicity: $[Q \Rightarrow R] \Rightarrow [S.Q \Rightarrow S.R]$.

Actually, the laws of $\vee$-distributivity and monotonicity are consequences of the law of $\wedge$-distributivity. Taking for *post* the predicate F and for *pre* T, we see that the law of excluded miracle does not hold. As a consequence, we do not have the law of $\wedge$-independence, but we do have the following law.

Law of (semi)-$\wedge$-independence: $[(S.Q) \wedge R \Rightarrow S.(Q \wedge R)]$
when none of the program variables of S appear free in R.

Remark We have defined the meaning of the specification statement, in terms of its weakest precondition, by carefully examining the smallest set of possible final states for some initial state. In the same way we could have defined the strongest postcondition of the specification statement by determining the smallest set of possible initial states for some final state. In order to do so, however, the interpretation of the specification statement when started in a state not satisfying *pre* needs to be more carefully investigated. Since we can avoid that problem in the *wp* semantics of the specification statement, we leave the strongest postcondition for what it is. □

2 Suitability of the specification statement

With the introduction of the specification statement, there is no need for a specification language. When deriving a program we can start with the specification statement specifying the problem. Then we refine that statement into another program, which can contain other specification statements, representing new and preferably simpler programming tasks. This process continues until an executable program is reached. Hence, program development is carried out uniformly within a single framework.

This raises two questions about the specification statement. First of all, the intention of the specification statement is to have, right from the beginning, a (non-executable) solution to the problem of finding a program

S satisfying $(\forall V :: \{pre\}\ S\ \{post\})$. Then we can carry out the entire program development by refinement only. Therefore, the question is, does the specification statement solve the given problem? The second question is the converse of the first one. Can any program satisfying a certain specification be conceived of as a refinement of the specification statement satisfying that specification? An affirmative answer would tell us that any program can be derived within this framework by starting with the proper specification statement. The following theorem states that both questions can be answered affirmatively with the above definition of the specification statement.

Theorem 0 For any program S we have

$$(v : V\,[pre,\ post]) \sqsubseteq S \text{ iff } (\forall V :: \{pre\}\ S\ \{post\}).$$

Proof For the only-if part we derive, assuming $(v : V\,[pre,\ post]) \sqsubseteq S$,

$$\begin{array}{ll} & S.post \\ \Leftarrow & \{ \text{ definition of } \sqsubseteq \text{ and assumption } \} \\ & (v : V\,[pre,\ post]).post \\ = & \{ \text{ definition of the specification statement } \} \\ & (\exists V :: pre) \wedge (\forall s :: (\forall V :: pre \Rightarrow post_s^v) \Rightarrow post_s^v) \\ \Leftarrow & \{ \text{ instantiation } \} \\ & pre \wedge (\forall s :: (pre \Rightarrow post_s^v) \Rightarrow post_s^v) \\ = & \{ \text{ calculus } \} \\ & pre \wedge (\forall s :: pre \vee post_s^v) \\ = & \{ \text{ calculus } \} \\ & pre \end{array}$$

For the if part, we assume the right hand side, i.e. $(\forall V :: pre \Rightarrow S.post)$. We assume f to be any value of v and R to be any postcondition. It suffices to show that

$$((v : V\,[pre,\ post]).R)_f^v \Rightarrow (S.R)_f^v$$

We derive

$$
\begin{array}{ll}
 & (S.R)^v_f \\
\Leftarrow & \{ \text{ law of monotonicity } \} \\
 & ((S.(\forall V :: pre^v_f \Rightarrow post)) \wedge (\forall v :: (\forall V :: pre^v_f \Rightarrow post) \Rightarrow R))^v_f \\
\Leftarrow & \{ \text{ see derivation below } \} \\
 & ((\exists V :: pre) \wedge (\forall v :: (\forall V :: pre^v_f \Rightarrow post) \Rightarrow R))^v_f \\
= & \{ \text{ substitution calculus, } s \text{ does not occur free in } post \text{ or } R \} \\
 & ((\exists V :: pre) \wedge (\forall s :: (\forall V :: pre \Rightarrow post^v_s) \Rightarrow R^v_s))^v_f \\
= & \{ \text{ definition of the specification statement } \} \\
 & (v : V\,[pre, post].R)^v_f
\end{array}
$$

Next we elaborate one step in the above derivation, viz. $(\exists V :: pre)^v_f \Rightarrow (S.(\forall V :: pre^v_f \Rightarrow post))^v_f$

$$
\begin{array}{ll}
 & (S.(\forall V :: pre^v_f \Rightarrow post))^v_f \\
= & \{ \text{ law of } \wedge\text{-distributivity } \} \\
 & (\forall V :: S.(pre^v_f \Rightarrow post))^v_f \\
\Leftarrow & \{ \text{ calculus and the law of } \vee\text{-distributivity } \} \\
 & (\forall V :: S.\neg pre^v_f \vee S.post)^v_f \\
\Leftarrow & \{ \text{ laws of } \wedge\text{-independence and } \wedge\text{-distributivity } \} \\
 & (\forall V :: ((S.T) \wedge \neg pre^v_f) \vee ((S.T) \wedge S.post))^v_f \\
= & \{ \text{ calculus } \} \\
 & (S.T \wedge (\forall V :: pre^v_f \Rightarrow S.post))^v_f \\
\Leftarrow & \{ \text{ law of monotonicity and substitution calculus } \} \\
 & ((\exists V :: S.post) \wedge (\forall V :: pre \Rightarrow S.post))^v_f \\
\Leftarrow & \{ \text{ assumption } \} \\
 & (\exists V :: pre)^v_f
\end{array}
$$

□

An immediate consequence of this theorem is that $v : V\,[pre, post]$ is a solution to problem of finding a program S such that $\{\, pre \,\}\, S\, \{\, post \,\}$. But it is not executable, so the task becomes to refine $v : V\,[pre, post]$ to an implementation.

3 Sharpness of the specification statement

In this section we show that our definition of the specification statement provides a true abstraction of all its implementations. In this abstraction, all implementation details are abstracted away, but the properties that all implementations have in common remain. For any postcondition R, the specification statement yields false in a certain state only if there is an implementation that is not guaranteed to end up in R when starting in that state. In this sense the specification statement defines the weakest precondition that can be inferred solely from its pre-/post-condition pair.

Another way of looking at it is to observe that all predicate transformers on program variable v constitute a complete lattice with the refinement order $\sqsubseteq$. The bottom element is **abort** and the top element is $v : [T, F]$, also called **magic** or **miracle**, cf. [2, 8]. What we would like to show, as argued in the previous paragraph, is that the specification statement is the greatest lower bound of all its implementations: any program which is refined by all implementations of a specification statement is refined by that specification statement itself. Actually, we have to settle for a little bit less: We can show this only if we know that the specification statement has at least one implementation. If a specification statement does not have any implementation it is not necessarily equal, as a predicate transformer, to **miracle**, although it would require a major miracle to implement it. For example, a specification could solve the halting problem, which allows no implementation, but as a predicate transformer it is not **magic**. We do have, however, the following theorem.

Theorem 1 Suppose $v : V\,[pre, post]$ has at least one implementation. Then $v : V\,[pre, post]$ is the greatest lower bound of all its implementations.

Proof Let M_0 be an implementation of $v : V\,[pre, post]$ and let S be a lower bound of all implementations of $v : V\,[pre, post]$, i.e.

$$(\forall M :: M \textbf{ impl } (v : V\,[pre,\, post]) \Rightarrow S \sqsubseteq M). \qquad (1)$$

We show that $S \sqsubseteq (v : V\,[pre,\, post])$ or, equivalently,

$$[\neg(v : V\,[pre,\, post]).R \Rightarrow \neg S.R]$$

for any postcondition R. Therefore, assume R to be some postcondition and f to be an arbitrary value of v such that

$$\neg((v : V\,[pre,\, post]).R)^v_f$$

Due to (1) it suffices to show that there exists an implementation M of $v : V\,[pre,\, post]$ such that $\neg(M.R)^v_f$. We construct an implementation M from M_0 depending upon which conjunct in the definition of the specification statement is false.

Case 1: $\neg(\exists V :: pre^v_f)$.
Informally, this means that the specification statement does not specify anything about a final state if we start in state f. Therefore, we might as well take as implementation M which acts as M_0 but which aborts in case we start in state f. In other words, we take as implementation

$$M \;:\; \textbf{if } v \neq f \rightarrow M_0 \textbf{ fi}$$

Now we formally prove that M satisfies our three proof obligations. Clearly, M is executable and $\neg(M.R)^v_f$ holds. The only remaining proof obligation is to show that M is a refinement of the specification statement. Using Theorem 0 it suffices to prove that $pre \Rightarrow M.post$. For that purpose we derive

$M.post$

$=$ { definition of M }

$v \neq f \wedge M_0.post$

$\Leftarrow$ { M_0 is a refinement of the specification statement, Theorem 0 }

$v \neq f \wedge pre$

$=$ { Case 1 }

pre

Case 2: $\neg(\forall\, s :: (\forall\, V :: pre_f^v \Rightarrow post_s^v) \Rightarrow R_s^v)$.
Informally, this means that there exists a final state s that we cannot possibly exclude as a result when starting in f, according to the specification statement, but for which the corresponding postcondition does not hold. Therefore, let s be such a state, i.e. such that

$$(\forall\, V :: pre_f^v \Rightarrow post_s^v) \wedge \neg R_s^v. \tag{2}$$

Hence a program which would act as M_0 everywhere, except when starting in state f in which case its result would be s is an implementation of the specification statement. Moreover, when starting in state f it ends up in a state not satisfying R, as desired. Therefore, we define our program M to be

$$\textbf{if } v \neq f \rightarrow M_0 \;[\!]\; v = f \rightarrow v := s \textbf{ fi}$$

Now we formally prove that M satisfies its three conditions. Clearly, M is executable and $\neg(M.R)_f^v$ holds, due to (2). The only remaining proof obligation is to show that M is a refinement of the specification statement. Due to Theorem 0, it suffices to prove that $pre \Rightarrow M.post$ for which we derive

$$\begin{array}{ll} & M.post \\ = & \quad \{ \text{ definition of } M \ \} \\ & (v \neq f \Rightarrow M_0.post) \wedge (v = f \Rightarrow post_s^v) \\ \Leftarrow & \quad \{ \ M_0 \text{ refines the specification statement, Theorem 0 } \} \\ & pre \wedge (v = f \Rightarrow post_s^v) \\ = & \quad \{ \text{ calculus, using (2) } \} \\ & pre \end{array}$$

□

Although Theorem 1 is more general than the sharpness theorem in [3], the basic construction used in the proof is very similar. The proof here is cleaner due to its more general context.

4 Comparison with the conjunction statement

In this section we investigate the relationship between our definition of the specification statement and the conjunction statement in [7]. We confine ourselves to the conjunction statement over an elementary specification statement, since that is what our definition of a specification statement amounts to.

It is clear from Definition 0 and (0) that our definition of the specification statement coincides with the specification statement in [7] when no logical constants are involved. However, as shown in the next example, when logical constants are involved our specification statement differs from the conjunction statement. The definition of the conjunction statement over an elementary specification statement amounts to an existential quantification over the logical constant in the elementary specification statement, i.e. $(\exists\, V :: pre \wedge (\forall\, v : post : R))$.

Example 4 Choosing *pre* and *post* as in Example 3, i.e.

$$pre \;=\; (v = V \vee v = V + 1) \text{ and } post \;=\; (v = V + 1 \vee v = V + 2),$$

we obtain for the conjunction statement with postcondition R the precondition $R^v_{v+1} \wedge (R^v_v \vee R^v_{v+2})$. Taking for R the predicate $v = 0$, the result is F, whereas our definition would yield $v = -1$. The conjunction statement does not take into account that an increment of v by 0 or by 2 is unsuitable as an implementation, since that would not do for *all* choices of V, which is what the statement specifies. □

The suitability condition (Theorem 0) also holds for the conjunction statement, but the conjunction statement is not sharp: our definition refines theirs. Among other things, it means that miracle specifications in disguise are more easily spotted with our definition, as the next example shows.

Example 5 We change the previous example specification slightly, viz. into

$$v : [v = V \vee v = V + 1, v = V + 1 \vee v = V + 3].$$

With Definition 0 we obtain T as a precondition for any postcondition R, which means that it specifies a miracle. The result of the conjunction statement would be $(R^v_{v+1} \wedge R^v_{v+3}) \vee (R \wedge R^v_{v+2})$, which is not equal to T. □

Probably more noteworthy is the fact that we do not lose $\wedge$-distributivity, whereas the conjunction statement does. This is so even when the conjunction statement is confined to be over elementary specification statements, as the following example shows.

Example 6 We take *pre* and *post* again as in Example 3,i.e.

$$pre \;=\; (v = V \vee v = V + 1) \text{ and } post \;=\; (v = V + 1 \vee v = V + 2).$$

Example 4 shows that the result of the conjunction statement is F when taking for R the predicate $v = 0$. When obtaining the conjunction statement for predicates $v \leq 0$ and $v \geq 0$, the conjunction of which is $v = 0$, we get $v \leq -1$ and $v \geq -1$ respectively, the conjunction of which is $v = -1$ rather than F. □

This means that we do not have to abolish the law of $\wedge$-distributivity in program development after all, which may be a great asset. It allows one to prove the correctness of a program incrementally. Conditions can be strengthened and only the additional predicate in isolation gives rise to new proof obligations.

5 Conclusion

We have presented a new definition of the specification statement that incorporates a rigorous treatment of logical constants. The definition itself is a generalization of a proof rule for procedures [3]. While maintaining many of their results, we have gained a clean presentation and a better understanding of logical constants. In particular, by considering the program lattice induced by the refinement ordering, it becomes clear that the specification statement should be the greatest lower bound of all its implementations, discarding the implementational detail, retaining everything else.

Although the significance of this work remains to be seen, it is a satisfying result that we can deal with logical constants without the need to discard the law of $\wedge$-distributivity. The only healthiness condition that we lose is the law of the excluded miracle.

6 Acknowledgement

This work is financially supported by the Department of Computer Science at Washington University in St. Louis through its chairman, Dr. Jerome R. Cox, Jr. Thanks are also due to Ken C. Cox whose comments improved the style of presentation.

References

[1] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Number 131 in Mathematical Centre Tracts. CWI, Amsterdam, 1980.

[2] R. J. R. Back and J. von Wright. A lattice-theoretic basis for a specification language. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science, pages 139–156. Springer-Verlag, 1989.

[3] A. Bijlsma, P. A. Matthews, and J. G. Wiltink. A sharp proof rule for procedures in wp semantics. *Acta Informatica*, 26:409–419, 1989.

[4] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[5] D. Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[6] C. C. Morgan. The specification statement. *ACM Transaction on Programming Languages*, 10:403–419, 1988.

[7] C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. In *On the Refinement Calculus*, Technical Monograph PRG-70, pages 103–134. Oxford University, 1988.

[8] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.