Avoiding the Undefined by Underspecification

David Gries * and Fred B. Schneider **

Computer Science Department, Cornell University
Ithaca, New York 14853 USA

Abstract. We use the appeal of simplicity and an aversion to complexity in selecting a method for handling partial functions in logic. We conclude that avoiding the undefined by using underspecification is the preferred choice.

1 Introduction

Everything should be made as simple as possible, but not any simpler.
—Albert Einstein

The first volume of Springer Verlag's *Lecture Notes in Computer Science* appeared in 1973, almost 22 years ago. The field has learned a lot about computing since then, and in doing so it has leaned on and supported advances in other fields. In this paper, we discuss an aspect of one of these fields that has been explored by computer scientists, handling undefined terms in formal logic.

Logicians are concerned mainly with studying logic —not using it. Some computer scientists have discovered, by necessity, that logic is actually a useful tool. Their attempts to use logic have led to new insights —new axiomatizations, new proof formats, new meta-logical notions like relative completeness, and new concerns. We, the authors, are computer scientists whose interests have forced us to become users of formal logic. We use it in our work on language definition and in the formal development of programs. This paper is born of that experience.

The operation of a computer is nothing more than uninterpreted symbol manipulation. Bits are moved and altered electronically, without regard for what they denote. It is we who provide the interpretation, saying that one bit string represents money and another someone's name, or that one transformation represents addition and another alphabetization. Our programs define sequences of uninterpreted symbol manipulations, and we elect to construe these as meaningful transformations on data.

How do we justify a belief that a program performs a meaningful transformation —how do we know that it "works"? Few programs produce outputs that are continuous in their inputs, so testing some input-output pairs and generalizing does not work. It is only by deriving, from a program's text, properties satisfied

* Supported by NSF grant CDA-9214957 and ARPA/ONR grant N00014-91-J-4123.
** This material is based on work supported in part by ARPA/NSF Grant No. CCR-9014363, NASA/ARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198.

by the program that we can be convinced that a program works. And how do we derive those properties? Through the use of formal logic —whose proofs, again, are simply uninterpreted symbol manipulations. Thus, uninterpreted symbol manipulation is the key: the computer uses it to execute programs, and we use it to reason about programs.

Logics were not originally intended to be used in this setting. They were intended primarily to formalize thought processes and the implications of employing various deductive apparatus. Logics were tailored to make their *study* easier. Little thought was given to "proof engineering" —the development and presentation of proofs and formalizations of logic that would make using logic easier. But, fueled by the need for programmers to manipulate logical formulas, programming methodologists began developing logics that were suited to use.

Although work on formal correctness of programs began in the late 1960's, the inadequacies of the standard formulations of propositional and predicate logic did not become apparent until the early 1980's. Since the formal logical systems usually taught (e.g. natural deduction) were unwieldy to use, researchers turned to a time-honored technique from mathematics. For decades, substitution of equals for equals —Leibniz's law— has been pervasive among users of mathematics. It allows them to infer new truths from old ones in a simple and efficient way. Why not make such a calculational style available to users of logic? A first cut at an equational logic for computer scientists appeared in the monograph [4]. Three years later, a freshman-sophomore-level text incorporating the approach appeared [5]. The new approach offers hope for a new view of logic and an entirely different method of teaching logic and proof.

One of the principles guiding research by those studying the formal development of programs has been simplicity and uniformity of concept and notation. Tools should be as simple as possible, if they are to be used reliably. Unnecessary distinctions and details should be avoided. For example, a logic should be usable for developing proofs without formal detail and complexity overwhelming. And, the method of proof presentation should lend itself to proof principles and strategies. Engineering and style are legitimate concerns when constructing proofs, just as they are in other creative activities.

In this article, we illustrate the use of simplicity as a goal by discussing the treatment of undefined terms and partial functions in equational propositional logic **E** and its extension to predicate logic [5]. Partial functions are ubiquitous in programming —some basic mathematical operations are partial (e.g. division), some basic programming operations are partial (e.g. array subscripting $b[n]$), and many functions that arise through recursive definitions are partial. Therefore, a logic for reasoning about programs must handle partial functions.

All treatments of the undefined exploit what logics do well —reason about variables, terms, and predicates that have *a priori* defined values. Some treatments of the undefined involve adding a new value $\perp$ (to represent "undefined") that may be assumed by a variable, term, or predicate. We discuss this approach in Sec. 2. Other treatments rule out writing formulas in which a variable, term, or predicate cannot be evaluated. Approaches along those lines are discussed

Table 1. Inference rules of Logic **E**

(1)	**Substitution :**	$\vdash P \quad \longrightarrow \; \vdash P[\nu := Q]$
(2)	**Leibniz :**	$\vdash P = Q \quad \longrightarrow \; \vdash E[\nu := P] = E[\nu := Q]$
(3)	**Transitivity :**	$\vdash P = Q,\; Q = R \quad \longrightarrow \; \vdash P = R$
(4)	**Equanimity :**	$\vdash P, P \equiv Q \quad \longrightarrow \; \vdash Q$

in Sec. 3. Finally, the approach we embrace is discussed in Sec. 4: regard all variables, terms, and predicates as being total, but in some cases underspecified.

2 Adding a new constant for undefined

In propositional logic **E** of [5], substitution of equals for equals (Leibniz (2) of Table 1) is the main inference rule. Further, equivalence $\equiv$ [3] plays a more prominent role than is usual in propositional logics. Heavy use is made of the associativity and symmetry of $\equiv$ in order to reduce the number of different theorems that have to be enumerated. For example, the theorem $p \equiv p \equiv \mathit{true}$ can be parsed in several ways: $(p \equiv p) \equiv \mathit{true}$ indicates that $\equiv$ is reflexive, while $p \equiv (p \equiv \mathit{true})$ and $p \equiv (\mathit{true} \equiv p)$ indicate that *true* is the identity of $\equiv$. In addition, many problems are more succinctly and easily formalized and have simpler solutions if properties of $\equiv$ are exploited —see e.g. [5].

In fact, the properties of equivalence are so useful that it is unwise to sacrifice them in an extension to handle the undefined. Equivalence should remain an equivalence relation —reflexive, symmetric, and transitive— and should have the identity *true*. In addition, inference rule Leibniz (substitution of equals for equals), along with the others of Table 1, should remain sound.

Consider changing just the model of evaluation of expressions by introducing a third constant $\perp$ to represent an undefined value. Thus, the value of a propositional variable or term in a state is one of *true*, *false*, and $\perp$. Since $p \equiv p \equiv \mathit{true}$ is a theorem of logic **E**, and since $\equiv$ is associative and symmetric, the following must hold:

- $\perp \equiv \perp$ evaluates to *true* (since $(\perp \equiv \perp) \equiv \mathit{true}$ is a theorem).
- $\perp \equiv \mathit{true}$ evaluates to $\perp$ (since $(\perp \equiv \mathit{true}) \equiv \perp$ is a theorem).
- $\mathit{true} \equiv \perp$ evaluates to $\perp$ (since $(\mathit{true} \equiv \perp) \equiv \perp$ is a theorem).

Now, what should be the value of $\perp \equiv \mathit{false}$?

[3] For booleans b and c, $b = c$ and $b \equiv c$ both denote equality. However, in our logic, operator $=$ is handled conjunctionally, i.e. $b = c = d$ is syntactic sugar for $b = c \wedge c = d$, while $\equiv$ is handled associatively —since in each state, $b \equiv (c \equiv d)$ and $b \equiv (c \equiv d)$ have the same value, the parentheses may be removed. Later, when dealing with predicate calculus, $=$ is regarded as polymorphic and $b = c$ is an (type-correct) expression iff b and c have the same type.

- Suppose $\perp \equiv \mathit{false}$ evaluates to false. Then $(\perp \equiv \mathit{false}) \equiv \mathit{false}$ evaluates to true and $\perp \equiv (\mathit{false} \equiv \mathit{false})$ evaluates to $\perp$, so $\equiv$ would not be associative.
- Suppose $\perp \equiv \mathit{false}$ evaluates to true. Then $(\perp \equiv \mathit{false}) \equiv \mathit{false}$ evaluates to false and $\perp \equiv (\mathit{false} \equiv \mathit{false})$ evaluates to $\perp$, so $\equiv$ would not be associative.
- Suppose $\perp \equiv \mathit{false}$ evaluates to $\perp$. Then $(\perp \equiv \perp) \equiv \mathit{false}$ evaluates to false and $\perp \equiv (\perp \equiv \mathit{false})$ evaluates to true, so $\equiv$ would not be associative.

With every choice of value for $\perp \equiv \mathit{false}$, associativity of equivalence has to be sacrificed.

Other standard logical properties no longer hold as well, when a constant $\perp$ is added. For example, suppose we take the strict approach: the value of an expression is $\perp$ if any of its constituents are $\perp$. Then, the law of the excluded middle, $p \vee \neg p$, and zero of $\vee$, $\mathit{true} \vee p \equiv \mathit{true}$, are no longer valid.

Any way we turn, the logic has to be changed.

Bijlsma [1] gives a model of evaluation that seems to contradict this conclusion. Through formal manipulation, he arrives at the following proposal:

> Suppose expression E contains some variables v whose values are $\perp$ in a state s. E evaluates to true in s iff E evaluates to true with all possible combinations of the values true and false for the values of variables v. Similarly, E evaluates to false in s iff E evaluates to false with all possible combinations of the values true and false for the values of v. Otherwise, E evaluates to $\perp$ in s.

For example, $v \vee \neg v$ and $\perp \equiv \perp$ both evaluate to true in all states, but $v \vee \neg w$ does not.

The apparent contradiction of our conclusion and Bijlsma's proposal is explained by our —hitherto implicit— desire for a *compositional* model of evaluation —the value of an expression must be completely determined by the values of its subexpressions. Bijlsma's proposal is not compositional. For example, consider a state in which v has the value $\perp$. Thus, v and $\neg v$ both have the value $\perp$, so we might assume that $v \vee \neg v$ evaluates to $\perp \vee \perp$, which evaluates to $\perp$. However, by Bijlsma's evaluation rule, $v \vee \neg v$ evaluates to true.

Each extension of expression evaluation to include $\perp$ compositionally requires extensive changes in the logic. We are not willing to accept changes in the logic if a more suitable alternative for dealing with undefined terms exists.

As the field has explored handling undefined terms, various proposals have been made to extend two-valued logics. Jones and Middleburg [8], for example, developed a typed version of logic LPF of partial functions. Their extension requires an operator that denotes "definedness" and two kinds of equality (weak and strong). Many useful properties in classical logic (eg. excluded middle and associativity of $\equiv$) do not hold. For us, their logic is far too complicated for use.

3 Abortive approaches that avoid undefined

Some approaches to avoiding partial functions assume a typed predicate logic. In a typed logic, every expression has a type, which can be determined syntactically from the expression itself and the types of its primitive constituents (constants, variables, and function symbols). For example, the type of addition $+$ may be $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Types provide structure. Also, compilers for typed programming languages offer significant help in uncovering mistakes early, through type checking, and this help could be made available for a typed predicate calculus as well. Further, having type be a syntactic property allows a more implicit use of properties associated with the type than would otherwise be possible in a safe way. If we know that a variable v is of type $\mathbb{Z}$ (integer), we can use properties of integers —rather than reals or complex numbers— in manipulating formulas containing v, without having to list all those properties explicitly in the formula or to give accompanying text explaining the properties being used. The type annotation $v{:}\mathbb{Z}$ is enough to alert the reader to the properties that can be used.

One approach to avoiding the undefined is to turn partial functions into total functions by restricting the types of their arguments. For example, instead of giving division the type

$$\mathbb{R} \times \mathbb{R} \to \mathbb{R} \quad ,$$

we give it the type

$$\mathbb{R} \times (\mathbb{R} - \{0\}) \to \mathbb{R} \quad .$$

Unfortunately, with this approach, any set can be a type. Therefore, there is no hope of having type be a syntactic property that can be checked mechanically, since the type system is undecidable. This approach is discussed further in [2].

A second approach (which can be used with a typed or an untyped logic) to avoiding the undefined is to view a function $f : B \to C$ as a relation on $B \times C$. Thus, $f.b = c$ is written as $(b, c) \in f$. Function application $f.b$ cannot be written, and just about the only way to refer to a value of a function is in an expression $(b, c) \in f$. To use this form, one must have an expression c for the value of $f.b$, which is awkward if not impossible in many situations. This approach is discussed briefly in [2].

A third approach (which can be used with a typed or an untyped logic) to avoiding the undefined was suggested by Scott [9]: classify all atomic formulas that contain an undefined term as *false*. This means that the law of the excluded middle holds. However, with almost any partial function, we will be able to find conventional laws that no longer hold. For example, the law of trichotomy of arithmetic,

$$x = 0 \ \vee \ x > 0 \ \vee x < 0 \quad ,$$

no longer holds, since $x/0 = 0 \vee x/0 > 0 \vee x/0 < 0$ evaluates to *false* (terms $x/0 = 0$, $x/0 > 0$ and $x/0 > 0$ are all undefined). This approach therefore seems unworkable.

4 Avoiding undefined through underspecification

Consider avoiding the undefined by using underspecification:

> All operations and functions are assumed to be defined for all values of their operands —they are *total* operations and functions. However, the value assigned to an expression need not be uniquely specified in all cases.

Thus, the value of every (type-correct) expression is defined. We don't have partial functions, we have underspecified functions. Therefore, our logics don't have to deal with undefined values. The propositional and pure predicate logics need not be changed. This approach to handling the undefined is mentioned in [3].

In this approach, division $/ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ has to be total, so what should be the value of $x/0$? According to the above principle, we leave it unspecified. The value of $x/0$ could be 2.4 or $20 \cdot x$ or any value in $\mathbb{R}$; we simply don't say what it is.

Axioms and theorems mentioning an underspecified operation $b \circ c$ usually involve an implication whose antecedent describes the set of states in which $b \circ c$ is uniquely specified. [4] In states in which the antecedent is *false*, the expression evaluates to *true*, even though $b \circ c$ evaluates to some (unknown) value. For example, the law $y/y = 1$ should be written as

(5) $\quad y \neq 0 \;\Rightarrow\; y/y = 1$

because y/y is uniquely specified iff $y \neq 0$.

We can prove theorems that rely on such laws in the standard manner. For example, to prove

$$y \neq 0 \Rightarrow x \cdot y/y = x \quad ,$$

we assume the antecedent $y \neq 0$ and prove the consequent:

$$\begin{array}{ll} & x \cdot y/y \\ = & \quad \langle \text{Modus ponens with assumption } y \neq 0 \text{ and (5) yields } y/y = 1 \rangle \\ & x \cdot 1 = x \\ = & \quad \langle \text{Identity of } \cdot \rangle \\ & x \end{array}$$

[4] In some formal systems, a predicate $Dom.$‘E’ is introduced to represent the set of states in which expression E is uniquely specified. For example, $Dom.$‘x/y’ denotes $y \neq 0$. Note that Dom is not a function symbol of the logic. Instead, $Dom.$‘E’ is shorthand for a predicate. To understand the need for quoting the argument of Dom, recall that a function application like $abs.E$ applies function abs to the value of E. For example, $abs(2-5)$ evaluates to $abs(-3)$, which evaluates to 3. Thus, $Dom(x/0)$ makes no sense, since it would result in $x/0$ being evaluated. Rather, the argument of Dom is some representation of an expression, and the result is a predicate that describes the set of states in which the expression is uniquely specified.

Using underspecification requires a slight change to the notion of validity. A formula E is *valid* if, for every combination of values that can be assigned to an underspecified term of E, E evaluates to *true* in all states. For example, for $x, y{:}\mathbb{R}$, $x/x = y/y$ is not valid, because one possible value for x/x is 2, one possible value for y/y is 3, and $2 \neq 3$. However, $x/x = x/x$ is valid.

The handling of recursively defined functions has been a concern of several articles that explore the undefined, most notably [2] and [8]. Article [8] introduces the following function $subp$,

$$\begin{array}{l} subp : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\ (\forall i{:}\mathbb{Z} \mid : subp(i,i) = 0) \\ (\forall i,j{:}\mathbb{Z} \mid i \neq j : subp(i,j) = subp(i,j+1)+1) \end{array}$$

and uses proofs of the following property as a leitmotif.

$$(6) \quad (\forall i,j{:}\mathbb{Z} \mid : i \geq j \Rightarrow subp(i,j) = i - j) \quad .$$

Function application $subp(i,j)$ is normally considered to be defined only if $i \geq j$. However, with the approach of underspecification, $subp(i,j)$ is always defined but its value is unspecified if $i < j$. Property (6) is proved as follows.

We prepare for a proof by induction by manipulating (6). Type annotations for the dummies are omitted.

$$\begin{array}{ll} & (\forall i,j \mid : i \geq j \Rightarrow subp(i,j) = i - j) \\ = & \quad \langle\, i \geq j \ \equiv\ i - j \geq 0 \,;\ \text{Trading; Nesting} \rangle \\ & (\forall i \mid : (\forall j \mid i - j \geq 0 : subp(i,j) = i - j)) \end{array}$$

We prove the last formula by proving

$$(\forall j \mid i - j \geq 0 : subp(i,j) = i - j)$$

for arbitrary i, by induction on natural numbers $i - j$.

Base case. For $i - j = 0$, we have $i = j$. Hence

$$\begin{array}{ll} & subp(i,j) = i - j \\ = & \quad \langle \text{Case } i = j \,;\ \text{Arithmetic} \rangle \\ & subp(i,i) = 0 \quad \text{—Definition of } subp \end{array}$$

Inductive case. For $i - j > 0$, we assume inductive hypothesis $subp(i,j+1) = i - (j+1)$ and prove $subp(i,j) = i - j$. For arbitrary natural number $i - j$, we have,

$$\begin{array}{ll} & subp(i,j) \\ = & \quad \langle \text{Definition of } subp \rangle \\ & subp(i,j+1)+1 \\ = & \quad \langle \text{Inductive hypothesis} \rangle \\ & i - (j+1) + 1 \\ = & \quad \langle \text{Arithmetic} \rangle \\ & i - j \end{array}$$

Jones [7] warns that handling undefined through underspecification can lead to difficulties if recursive definitions inadvertently overspecify in such a way that unwanted properties can be deduced. For example, suppose we define $fact.i : \mathbb{Z} \to \mathbb{Z}$ by

(7) $fact.i = \textbf{if } i = 0 \textbf{ then } 1 \textbf{ else } i \cdot fact(i-1) \textbf{ fi}$.

Then, we can use recursive definition (7) to deduce

$$\begin{array}{rcl} fact(-1) & = & -1 \cdot fact(-2) \\ fact(-2) & = & -2 \cdot fact(-3) \\ & \cdots & \end{array}$$

In this case, $fact$ has indeed been overspecified, since $fact(-1) = -fact(-2)$ can be deduced. But the fault lies in the recursive definition rather than in handling undefined using underspecification. The problem can be solved by defining $fact$ so that it indicates nothing about $fact.i$ for negative i:

$$fact.0 = 1$$
$$(\forall i{:}\mathbb{Z} \mid i > 0 : fact.i = i \cdot fact(i-1))$$

We conclude that underspecification is the preferred way to deal with undefined terms. With this approach, we can to continue to use simple two-valued logic, with all its nice properties, and still be formal, rigorous, and clear. Only when dealing with a partial function does it become necessary to exert some minimal extra effort, which takes the form of dealing with antecedents that describe the set of arguments for which the partial function is uniquely specified.

References

1. Bijlsma, A. Semantics of quasi-boolean expressions. In Feijen, W.H.J., et al (eds.) *Beauty is Our Business.* Springer Verlag, New York, 1990, 27–35.
2. Cheng, J.H., and C.B. Jones. On the usability of logics which handle partial functions. In: C. Morgan and J.C.P. Woodcock (eds.). Third Refinement Workshop, pp. 51–69. Workshops in Computing Series, Heidelberg, 1991.
3. Constable, R.L., and M.J. O'Donnell. *A Programming Logic.* Winthrop, Cambridge, Massachusetts, 1978.
4. Dijkstra, E.W., and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer Verlag, New York 1990.
5. Gries, D., and F.B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag, New York, 1993.
6. Gries, D., and F.B. Schneider. Equational propositional logic. *IPL 53* (1995), 145-152.
7. Jones, C.B. Partial functions and logics: a warning. *IPL 54* (1995), 65–68.
8. Jones, C.B., and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica 31* (1994), 399–430.
9. Scott, D.S. Existence and description in formal logic. In R. Schoenman (ed.) *Bertrand Russell, Philosopher of the Century.* St. Leonards: Allen and Unwin, 1967, 181–200.