Acta Informatica 21, 251–269 (1984)

Acta Informatica

A Logic Covering Undefinedness in Program Proofs

H. Barringer, J.H. Cheng, and C.B. Jones

Department of Computer Science, University of Manchester, Manchester M13 9PL, GB

Summary. Recursive definition often results in partial functions; iteration gives rise to programs which may fail to terminate for some imputs. Proofs about such functions or programs should be conducted in logical systems which reflect the possibility of "undefined values". This paper provides an axiomatization of such a logic together with examples of its use.

1. Background

Many approaches have been developed to proving programs correct – most can also be applied to developing programs so that they are correct. Propositional and predicate calculus are used in nearly all such methods. The standard texts on mathematical logic (e.g. [21]) assume that formulae denote one of two truth values (**true, false**). The approach considered here is to accept that certain formulae do *not* have such a denotation. A term such as factorial of -1 can be thought of as not denoting an integer; a formula, which contains such a term, as not denoting a truth value. The logic which is presented below copes with such situations.

This section contains some introductory examples of the problems caused by partial functions. Section 2 contains a description of a model theory for a logic of partial functions; the corresponding proof theory is discussed in Sect. 3. The full axiomatization is given in the appendices. Example proofs comprise the fourth section of this paper; these are followed by a final section which reviews alternative approaches and draws some conclusions.

Partial functions occur commonly in computing, both in actual programs (or parts thereof) and in functions used to reason about the effect of programs. Partial functions arise quite naturally with recursive definitions. There are, of course, classes of recursive functions which are constructed in such a way that it is obvious that they are defined over their stated domain. In other cases, the set of values for which a function is defined is not obvious. For example, a subtraction function over the natural numbers might be written:

$$\text{subp}(i,j) \triangleq \textbf{if } i=j \textbf{ then } 0 \textbf{ else } \text{subp}(i+1,j)+1$$

Thinking operationally, it might be said that the result of an application of subp, where the value of i is greater than that for j, is undefined - the function recurses infinitely. Although this example is small, it is indicative of the way in which recursion can give rise to partial functions. The example is also simple enough to show a formal proof - cf. Sect. 4. Formalising the above statement, it is claimed that, assuming that i and j are natural numbers:

$$\text{subp}(i,j) = j - i$$

providing:

$$i \leqq j.$$

The problem of undefined terms arises, however, if this is written as:

$$\forall i,j \,.\, i \leqq j \Rightarrow \text{subp}(i,j) = j - i.$$

When the antecedent of the implication is false, the term subp(i,j) is undefined.

Consider, as another example of a partial function, the following definition of 'maxs' – a function which yields the maximum of a set of integers. Given a function 'max' whose result is the larger of its two integer arguments:

$$\begin{aligned}\text{maxs}(s) \triangleq\ & \textbf{let } e \in s \textbf{ in} \\ & \textbf{if card } s = 1 \textbf{ then } e \\ & \textbf{else } \max(e, \text{maxs}(s - \{e\})).\end{aligned}$$

The choice of an element e in set s can only be made if s is not empty. The fact that maxs is partial gives rise to formulae like:

$$\forall s \,.\, s \neq \{\,\} \Rightarrow \text{maxs}(s) \in s.$$

Here again, the consequent contains a term which is not defined if the antecedent is false. In the context of specification languages (e.g. VDM [4]), partial functions arise for data types such as mappings and lists. The basic list operators for head (hd), tail (tl) and indexing are partial in an obvious way. Thus the problem with "undefined values" recurs with:

$$t = \langle\,\rangle \vee t = \text{append}(\textbf{hd}\, t, \textbf{tl}\, t).$$

A similar problem is seen with application of mappings; [4] uses expressions like:

$$id \in \textbf{dom}\, \rho \wedge \rho(id) \in \text{Proctype}.$$

If id is not in the domain of the mapping ρ, then $\rho(id)$ is undefined.

Predicates themselves can be defined recursively and this introduces new problems. Consider a mapping:

$$M = X \underset{m}{\rightarrow} X$$

with a function:

$$\begin{aligned}&\text{path}: X \times M \rightarrow X\text{-set} \\ &\text{path}(x,m) \\ &\quad \triangleq \{x\} \cup (\textbf{if } x \notin \textbf{dom}\, m \textbf{ then } \{\,\} \textbf{ else } \text{path}(m(x), m)).\end{aligned}$$

This is partial in that it will "loop" if the mapping does not represent a well-founded relation. One way of writing the well-founded predicate is:

$$\begin{aligned}&\text{is-wf: } M \to \text{Bool}\\&\text{is-wf}(m) \triangleq \forall x \,.\, \text{noloop}(x, m)\\&\text{noloop: } X \times M \to \text{Bool}\\&\text{noloop}(x, m)\\&\quad \triangleq x \notin \mathbf{dom} m \vee \text{noloop}(m(x), m).\end{aligned}$$

The difficulty with such a definition is that it fails to denote a truth value for reflexive mappings. Thus:

$$\models \text{is-wf}(m) \Rightarrow \forall x \,.\, x \in \text{path}(x, m)$$

does not hold for reflexive mappings. However, by writing:

$$\text{is-wf}(m) \models \forall x \,.\, x \in \text{path}(x, m)$$

it is possible to express the required idea. The model theoretic interpretation of such a statement is that in those "worlds" where every assumption clause is (both defined and) true, the conclusion is (defined and) true; if any clause in the list of assumptions is either undefined or false, the statement vacuously holds. Koletsos - in [22] - gives this as his "weak" interpretation; a "strong" form, in which some assumption must be false if the conclusion is false, is also given. The "strong" form can be inverted in an obvious way and is used in [5].

Computer scientists have adopted various measures to cope with the problem of proofs about partial functions. Leaving aside those who simply ignore the problem, the most popular approach appears to be to introduce special propositional operators. Thus, [17] uses the conventional two-valued "and" operator ($\wedge$) as well as using conditional expressions to define:

$$p \,\&\, q \triangleq \textbf{if } p \textbf{ then } q \textbf{ else false}$$

Dijkstra [11] uses **cand** in the same way. The use of the two operators is mildly inconvenient (what, for example, are the distributive rules?) and rarely supported by a proof theory; the fact that the sequential "&" operator is, in general, non-commutative brings an unfortunate computational consideration into the logic. The operational semantics work called "VDL" (see [23]) defined all logical operators via conditional expressions. This has the disadvantage that it is not then clear when operands can be commuted.

In [18] an attempt was made to cope with "undefined values" without special propositional operators. To a large extent, bounded quantifiers constrained function arguments; where this was insufficient, conditional expressions were written. Some of the expressions which resulted were clumsy.

Other approaches are reviewed in Sect. 5.

2. Model Theory

The basic problem under consideration is undefined values which arise from recursive functions. Following the work on denotational semantics, such "val-

ues" are considered to be 'bottom" ($\perp$) elements. The application of any "strict" predicate (including weak equality) to bottom will not yield a truth value. Blamey [5] views this as a "gap" in the truth values. If this "gap" is written as a bottom element, truth tables - with values abbreviated as "t", "f", "$\perp$" - can be drawn which provide a model theory for the propositional calculus. These truth tables are presented to convey an intuitive idea of the logic which is defined in Sect. 3 by a proof theory. In that theory, "$\perp$" is viewed as a "gap" and is never written. The obvious way to make the operators as "generous" as possible is to extend the standard two-valued truth tables by giving a result whenever enough information is available. This gives rise to the table (as in [20]) for "or":

$\vee$	t	f	$\perp$
t	t	t	t
f	t	f	$\perp$
$\perp$	t	$\perp$	$\perp$

As observed by McCarthy [24], this result cannot be achieved by defining the operator by conditional expressions: the variable in the conditional must always be evaluated; undefinedness of the "inevitable" variable always results in an undefined value for the conditional expression. The symmetrical table appears to require some form of parallel elaboration. The related table for "and" is:

$\wedge$	t	f	$\perp$
t	t	f	$\perp$
f	f	f	f
$\perp$	$\perp$	f	$\perp$

McCarthy also makes the interesting observation that the employment of the axiom:

$$(\textbf{if } p \textbf{ then } a \textbf{ else } a) = a$$

where p can be undefined, gives rise to a system in which the conditional expression definition given above for "&" yields this symmetrical truth table.

These truth tables are symmetric and thus the operators defined are commutative. For "not", the table is:

$\sim$	
t	f
f	t
$\perp$	$\perp$

Notice that the law of the "excluded middle":

$$p \vee \sim p$$

is not a tautology.

The table chosen for implication is that derived from defining:

$$p \Rightarrow q$$

as:

$$\sim p \vee q$$

thus:

$\Rightarrow$	t	f	$\perp$
t	t	f	$\perp$
f	t	t	t
$\perp$	t	$\perp$	$\perp$

There is some controversy about this decision (see [32]). It might be argued that:

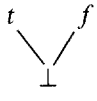
$$p \Rightarrow p$$

should be a tautology. However, the given table treats:

$$\text{mod}(5,0) = \text{mod}(5,0) \Rightarrow \text{mod}(5,0) = \text{mod}(5,0)$$

as undefined. The reduction of the implication to the law of the "excluded middle" is an argument against the proposed tautology.

More importantly, the truth-tables given above are monotonic in the following ordering on truth values:



It can be shown that the operators $\sim$ and $\vee$, with the constants t and $\perp$, are expressively complete for monotone truth-tables. Notice some constants are needed since they can not otherwise be formed by formulae.

The quantifiers of predicate calculus are $\forall$ and $\exists$. As in 2-valued logic, $\forall$ can be treated as a generalised conjunction and $\exists$ as generalised disjunction. All of the conventions such as definitions of bound and free variables still apply. However, as the domains have been augmented with an improper element, it is stipulated that the quantifiers range only over the proper elements. This is a crucial difference from LCF (cf. discussion in Sect. 5) where the propagation of undefined elements can be inconvenient.

The discussion above relates to the monotone part of the logic. The proof system given here can normally be used to prove functions correct without discussing "undefined values". However, where it is necessary to distinguish the defined truth values (t and f) and the undefined one ($\perp$), an operator $\triangle$ [16, 33] has been introduced. Its truth table is:

E	$\triangle E$
t	t
f	t
$\perp$	f

Note that $\triangle$ is not monotone. Thus the connectives of the propositional calculus (in descending order of precedence) are $\triangle$, $\sim$, $\wedge$ and $\vee$. It can be proved that, with the addition of $\triangle$, any operator can be defined.

The system used here has two equality predicates: '=' is "weak equality" [25]; it is strict and will give $\perp$ as a value for $s1 = s2$ if either (or both) of $s1$ and $s2$ are the improper element; for $s1$ and $s2$ both proper, $s1 = s2$ gives t if they denote the same (proper) element, and f otherwise. The "strong equality" predicate '==' is two-valued and gives t if both $s1$ and $s2$ denote the same element (proper or improper) and f otherwise. Thus, for a simple flat domain with elements 0, 1, 2, $\perp_n$:

=	0	1	2	$\perp_n$
0	t	f	f	$\perp$
1	f	t	f	$\perp$
2	f	f	t	$\perp$
$\perp_n$	$\perp$	$\perp$	$\perp$	$\perp$

==	0	1	2	$\perp_n$
0	t	f	f	f
1	f	t	f	f
2	f	f	t	f
$\perp_n$	f	f	f	t

Notice that "weak" equality is (strict and) monotone while "strong" equality is not monotonic.

3. Proof Theory

In order to give formal proofs, axioms and inference rules must be given. The axiomatization chosen for the logic used in this paper is given in full in the appendices. Appendix I contains the monotone system expressed in terms of basic operators, definitions of other operators and derived rules; Appendix II characterizes the non-monotone connectives.

The proof style known as "natural deduction" can be used to provide proofs which are easy to view from a general overview down to the particular details. The proof system given here adopts a natural deduction style - certain important modifications to the standard form (cf. [13]) of such proofs are discussed below.

In order to be useful, a proof theory must be consistent and, if possible, complete. The given axiomatization is both consistent and complete with respect to the truth-table model of the preceding section. The proof theory given here is a natural decuction system originating from two sequent calculi by Koletsos [22] and by Hoogewijs [16]. Guidelines for the transformation from the sequent calculi to natural deduction can be found in Prawitz [28].

Many formulae which are tautologies in two-valued logic are not tautologies in the system presented here because of the need for definedness; under the assumption of definedness (δ) the system here becomes the same as conventional predicate calculus. The "axiomatization" here consists almost entirely of deduction rules. Examples of deduction rules (with assumptions - separated by commas - above the line, and conclusions below) are those which permit the introduction and elimination of propositional operators:

$$\vee\text{-}I \quad \frac{Ei}{E1 \vee E2} \quad (1 \leqq i \leqq 2)$$

$$\vee\text{-}E \quad \frac{E1 \vee E2, E1 \vdash E, E2 \vdash E}{E}$$

$$\wedge\text{-}I \quad \frac{E1, E2}{E1 \wedge E2}$$

$$\wedge\text{-}E \quad \frac{E1 \wedge E2}{Ei} \quad (1 \leqq i \leqq 2).$$

The $\vee$-E rule is "indirect" because not everything above the line is a formula. $E1 \vdash E2$ represents a deduction with $E1$ as its assumption and $E2$ as conclusion. This is the same as the model theoretic notion ($\models$) thus when $E1$ is true, $E2$ is also true; when $E1$ is false or undefined, $E2$ can have any value.

Similarly, deduction rules for the quantifiers can be given (see Appendix I for notation used in substitution):

$$\forall\text{-}I \quad \frac{p(x)}{\forall x \,.\, p(x)} \quad (x \text{ is arbitrary})$$

$$\forall\text{-}E \quad \frac{\forall x \,.\, p(x), s = s}{p(s/x)}.$$

These rules are, in fact, the only ones used in the examples of proofs given below.

These deduction rules are all valid in a two-valued logic. The crucial differences are in the *omissions*. In the logic given here, there are no rules which permit proof by contradiction - or, equivalently, the law of the "excluded middle" does not hold. As a consequence, special rules are required which permit the introduction and elimination of negations of terms.

$$\sim \vee\text{-}I \quad \frac{\sim E1, \sim E2}{\sim(E1 \vee E2)}$$

$$\sim \vee\text{-}E \quad \frac{\sim(E1 \vee E2)}{\sim Ei} \quad (1 \leqq i \leqq 2).$$

The key difference between the natural deduction scheme used here and the conventional (2-valued) system can now be seen to result from the fact that the "deduction theorem" does not hold in the logic presented here. Although

$(E1 \Rightarrow E2)$ is defined as $(\sim E1 \vee E2)$, a different rule for "$\Rightarrow$-I" is needed. The reason is that it is trivially true that:

$$\frac{E}{E}$$

but one cannot conclude $E \Rightarrow E$ as this is equivlent to $\sim E \vee E$, which holds only when E is defined. A formula E is defined if:

$$E \vee \sim E$$

is true. The abbreviation δE is introduced for this formula. Notice that, unlike ΔE, δE is monotone, Thus the relevant rule for introducing implication is:

$$\Rightarrow\text{-}I \quad \frac{E1 \vdash E2, \delta E1}{E1 \Rightarrow E2}$$

Substitution is an important step in proofs. (The syntactic expression $p[s2/s1]$ represents the expression p with $s2$ replacing *some* occurrence of $s1$ – see Appendix I.) The following rule holds:

$$=\text{-subs} \quad \frac{s1 = s2, p}{p[s2/s1]}.$$

With the given basic rules, it is possible to derive rules which facilitate proofs. An example of a proof of one of the distributive laws is:

$(E1 \vee E2) \wedge (E1 \vee E3) \vdash E1 \vee E2 \wedge E3$

1. $E1 \vee E2$	$\wedge$-$E, pr0$
2. $E1 \vee E3$	$\wedge$-$E, pr0$
3. $E1 \vdash E1 \vee E2 \wedge E3$	
3.1 $E1 \vee E2 \wedge E3$	$\vee$-$I, pr3$
4. $E2 \vdash E1 \vee E2 \wedge E3$	
4.1 $E2$	$pr4$
4.2 $E3 \vdash E1 \vee E2 \wedge E3$	
4.2.1 $E2 \wedge E3$	$\wedge$-I, 4.1, $pr4.2$
4.2.2 $E1 \vee E2 \wedge E3$	$\vee$-I, 4.2.1
4.3 $E1 \vee E2 \wedge E3$	$\vee$-E, 2, 3, 4.2
5. $E1 \vee E2 \wedge E3$	$\vee$-E, 1, 3, 4

A common proof of this law in two-valued natural deduction uses the law of the "excluded middle" (see [13] for example). The proof as presented here is, of course, quite acceptable in two-valued systems. As might be expected, this proof was harder to obtain.

4. Example Proofs

In order to present proofs about programs, results must be used about the problem domains (e.g. natural numbers, trees). Here, such results are brought in by axioms whose justification is *not* the purpose of this paper. It is also necessary to develop a "style" of using the proof system which minimizes the use of arguments about undefined; several observations are made below in this connection.

The first example shows how a simple property of real numbers can be deduced in spite of the fact that one of the terms might be undefined.

$\vdash \forall x\,.\,x=0 \vee x/x=1$

1. $\vdash x=0 \vee x/x=1$
   - 1.1 $x=0 \vee x \neq 0$ — *pr*1, numbers
   - 1.2 $x=0 \vdash x=0 \vee x/x=1$ — $\vee$-*I*, *pr*1.2
   - 1.3 $x \neq 0 \vdash x=0 \vee x/x=1$
     - 1.3.1 $x/x=1$ — *pr*1.3, numbers
     - 1.3.2 $x=0 \vee x/x=1$ — $\vee$-*I*, 1.3.1
   - 1.4 $x=0 \vee x/x=1$ — $\vee$-*E*, 1.1, 1.2, 1.3
2. $\forall x\,.\,x=0 \vee x/x=1$ — $\forall$-*I*, 1

Notice, however, that it would not be possible to prove that:

$$x/0=1 \vee \sim(x/0=1).$$

For the next example, a proof is given of the "obvious" property about the partial function subp mentioned in Sect. 1. Given:

$$\text{subp}(i,j) \triangleq \textbf{if } i=j \textbf{ then } 0 \textbf{ else } \text{subp}(i+1,j)+1.$$

The desired property can be stated as:

$$\forall i,j\,.\,j-i \geqq 0 \Rightarrow \text{subp}(i,j)=j-i.$$

This property is proved by induction. The basic facts about natural numbers are introduced by an induction schema:

$$\frac{\vdash w(0/k) \qquad k \leqq 0, w(k) \vdash w(k+1/k)}{\forall k\,.\,k \geqq 0 \Rightarrow w(k)}$$

This form of the rule hides both "$\Rightarrow$-*I*" and a "$\forall$-*I*"; the reason for using the turnstile in the inductive step is discussed in Sect. 5.

In order to reason about subp in the following proof, the following axiom and inference rule are used:

$$d1 \quad \frac{}{\text{subp}(n,n)=0}$$

$$d2 \quad \frac{n1 \neq n2, \text{subp}(n1+1,n2)=n3}{\text{subp}(n1,n2)=n3+1}$$

Informally, these are properties which follow from the definition. These rules could be read more directly from other ways of presenting recursive definitions. Notice that the problem of the recursion not yielding a result is covered by the hypothesis of $d2$. The *use* of these rules will establish that the result of subp is defined. There is clearly a need to establish that such "reformulations" are consistent with some semantics for recursive functions and this is addressed after the main proof.

The following is the proof of the required property by natural deduction:

$\vdash \forall i,j\,.\,j-i \geqq 0 \Rightarrow \mathrm{subp}(i,j)=j-i$

1. $\vdash \forall k\,.\,k \geqq 0 \Rightarrow \mathrm{subp}(j-k,j)=k$
   - 1.1 $\mathrm{subp}(j-0,j)=0$ — $d1$
   - 1.2 $k \geqq 0, \mathrm{subp}(j-k,j)=k \vdash \mathrm{subp}(j-(k+1),j)=k+1$
     - 1.2.1 $k \geqq 0$ — pr 1.2
     - 1.2.2 $k+1>0$ — integers
     - 1.2.3 $j-(k+1) \neq j$ — integers
     - 1.2.4 $\mathrm{subp}(j-k,j)=k$ — pr 1.2
     - 1.2.5 $j-k=j-(k+1)+1$ — integers
     - 1.2.6 $\mathrm{subp}(j-(k+1)+1,j)=k$ — $=$-subs., -1, -2
     - 1.2.7 $\mathrm{subp}(j-(k+1),j)=k+1$ — $d2$, -1, -4
   - 1.3 $\forall k\,.\,k \geqq 0 \Rightarrow \mathrm{subp}(j-k,j)=k$ — indn, 1.1, 1.2
2. $j-i \geqq 0 \Rightarrow \mathrm{subp}(j-(j-i),j)=j-i$ — $\forall$-E, 1
3. $j-i \geqq 0 \Rightarrow \mathrm{subp}(i,j)=j-i$ — 2, integers
4. $\forall i,j\,.\,j-i \geqq 0 \Rightarrow \mathrm{subp}(i,j)=j-i$ — $\forall$-I twice, 3

The proof rules used about subp ($d1$ and $d2$) serve to insulate the logical system given here from the extra-logical facts about the functions. It is, however, possible to show how such rules are justified. There are two ways of giving a semantics to recursive functions: denotational and operational.

A function like subp can be taken to denote the least fixed point (cf. [25]) of:

$$\mathrm{subp}=\mu F$$
$$F=\lambda f\,.\,\lambda i,j\,.\,\textbf{if } i=j \textbf{ then } 0 \textbf{ else } f(i+1,j)+1.$$

If subp is read as an operational description of how to compute a result, it is possible to justify rules $d1$ and $d2$ by natural deduction. The operational interpretation results in reading the definition symbol ($\triangleq$) as strong equality ($==$). The proof also relies upon the following elimination rules for conditional expressions:

$$\textbf{if-then-}E \quad \frac{E, (\textbf{if } E \textbf{ then } s1 \textbf{ else } s2)==s}{s1==s}$$

$$\textbf{if-else-}E \quad \frac{\sim E, (\textbf{if } E \textbf{ then } s1 \textbf{ else } s2)==s}{s2==s}.$$

The proofs follow. (Notice that these proofs involve the - non-monotone - "strong" equality.):

$\vdash \text{subp}(n,n)=0$

1.	$\text{subp}(n,n)==\textbf{if } n=n \textbf{ then } 0 \textbf{ else } \text{subp}(n+1,n)+1$	definition of subp
2.	$n=n$	integers
3.	$\text{subp}(n,n)==0$	**if-then**-E, 1, 2
4.	$0=0$	=-cons
5.	$\text{subp}(n,n)=0$	$==\rightarrow=$, 3, 4

$n1+n2,\ \text{subp}(n1+1,n2)=n3 \vdash \text{subp}(n1,n2)=n3+1$

1.	$\text{subp}(n1,n2)==$ **if** $n1=n2$ **then** 0 **else** $\text{subp}(n1+1,n2)+1$	definition of subp
2.	$n1+n2$	pr
3.	$\sim(n1=n2)$	integers
4.	$\text{subp}(n1,n2)==\text{subp}(n1+1,n2)+1$	**if-else**-E, 1, 3
5.	$\text{subp}(n1+1,n2)=n3$	pr
6.	$\text{subp}(n1,n2)==n3+1$	=-subs, 4, 5
7.	$n3=n3$	=-var
8.	$1=1$	=-cons
9.	$(n3+1)=(n3+1)$	integers, 7, 8
10.	$\text{subp}(n1,n2)=n3+1$	$==$ to $=$, 6, 9

In [19], proofs are given about algorithms which use binary trees to represent mappings from Keys to Data. The interesting feature of those proofs is the use of structural induction. Here, a much simplified problem is used *only* to illustrate such induction over tree-like objects.

Suppose:

$$\text{Tree}=\text{Node}\cup\text{Data}$$
$$\text{Node}=\{mk-\text{Node}(l,r) \mid l\ r\in\text{Tree}\}$$

and selector functions are available such that:

$$L\colon \text{Node}\rightarrow\text{Tree}$$
$$R\colon \text{Node}\rightarrow\text{Tree}$$
$$L(mk-\text{Node}(l,r))=l$$
$$R(mk-\text{Node}(l,r))=r.$$

The relevant induction axiom is:

$$\frac{t\in\text{Data}\vdash p(t) \qquad t\in\text{Node}, p(L(t)), p(R(t))\vdash p(t)}{t\in\text{Tree}\vdash p(t)}$$

The function:

$$\text{collect}(t)\triangleq \textbf{if } t\in\text{Data } \textbf{then } \{t\} \textbf{ else } \text{collect}(L(t))\cup\text{collect}(R(t))$$

is only defined over Trees. Here again the knowledge about the function can be isolated in two rules:

$$d3 \quad \frac{t \in \text{Data}}{\text{collect}(t) = \{t\}}$$

$$d4 \quad \frac{t \in \text{Node},\ \text{collect}(L(t)) = s1,\ \text{collect}(R(t)) = s2}{\text{collect}(t) = s1 \cup s2}.$$

It might be necessary to show:

$$t \in \text{Tree} \vdash \text{collect}(t) \neq \{\ \}.$$

Such a proof is quite straightforward:

$t \in \text{Tree} \vdash \text{collect}(t) \neq \{\ \}$

1. $t \in \text{Data} \vdash \text{collect}(t) \neq \{\ \}$	
1.1 $\text{collect}(t) = \{t\}$	$\text{pr}1, d3$
1.2 $\text{collect}(t) \neq \{\ \}$	$-1, \text{pr}1$
2. $t \in \text{Node}, \text{collect}(L(t)) \neq \{\ \}, \text{collect}(R(t)) \neq \{\ \} \vdash$	$\text{collect}(t) \neq \{\ \}$
2.1 $\text{collect}(t) = \text{collect}(L(t)) \cup \text{collect}(R(t))$	$\text{pr}2, d4$
2.2 $\text{collect}(t) \neq \{\ \}$	$-1, \text{pr}2, \text{set}$
3. $\text{collect}(t) \neq \{\ \}$	$\text{indn}, \text{pr}, 1, 2$

5. Discussion

This section begins with a review of other relevant work. Jean-Raymond Abrial (see [1,2]) uses a logic in which the law of the "excluded middle" holds; his treatment of functions, however, avoids the problem of "undefined" values. This is achieved by viewing the results of all functions as sets and arranging that application outside what might be thought of as the actual domain yields an arbitrary set value - any predicate is either true or false but it is not possible to show which.

An elegant and theoretically sound approach has come from the work on denotational semantics (see references in [25]). Extensions are made to the domains of partial functions so that an undefined application of a partial function will result in the improper element (bottom). Thus, for partial predicates, the naturally extended co-domain is exactly three-valued - true, false and bottom [3]. Unfortunately, no proof theory is given from this approach.

LCF [12] handles undefinedness explicitly. However, its underlying logic is still two-valued. In effect, two levels of truth values are used. The "terms", which represent values computed, can be undefined - thus there are three values for the Boolean type - TT, FF, UU. "Forms", which are assertions about values (terms), are two-valued (TRUTH and FALSITY). The bridge between these two levels is through a strong equality (==) which is not monotone. The decision to make the quantifiers range over undefined as well

as "proper" elements complicates proofs by requiring that extra cases be considered.

In PL/CV2, two other solutions can be adopted and neither changes the underlying 2-valued logic. As mentioned in both [8] and [9], one solution introduces some anomaly as both mod(5,0)=mod(5,0) and mod(5,0)≠mod(5,0) can be false, whereas ∼(mod(5,0)=mod(5,0)) and ∼(mod(5,0)≠mod(5,0)) are both true. The other solution allows mod(5,0)=mod(5,0) to be true. This decision, to bend the logic to suit such trivial cases, while its usefulness is in doubt, may result in some risks; in practice, it is more likely the case that a programmer has written such a term by mistake.

Another possible approach, similar to those in PL/CV2, is to force partial predicates into total ones by not extending the target domain, e.g. using strong equality [25] [3]. The annoying aspect is that it will destroy the monotonicity, which is a pre-requisite for application of the fixed point theorem, and hence may render some difficulties when reasoning about partial function using the "forced total predicates".

Owe's *P*-logic is designed specifically to deal with partial functions [26]. For every formulae *e*, "*e*" denotes its well-definedness; "*e*" is total in Owe's formalization. This "well-definedness" predicate is not dissimilar to Δ as used in the logic presented in this paper. The key difference is the separation in Owe's logic of truth and definedness. Two proofs are needed: one for *e* the other for "*e*". This is the result of the concept of "partial implementation" in which a mathematically true formula may become undefined in an implementation.

Logicians (philosophers and mathematicians) have studied the problem of what truth value should be given to a sentence containing a term "which has no denotation"? An example given by Russell is the sentence:

"The present King of France is bald".

Logicians have proposed various solutions to this problem. Many of these solutions give rise to non-classical logics (see [14, 15] for an account of many of the documented approaches). Non-classical logics dealing with three values date from the 1920's. The differences among three-valued logics depend on the interpretation of the third truth value (see [29]). In Kleene's system, the third value is to model the undecidable predicates ([20], cf. [15]). Based on Kleene's system, variations can still be found. These can be classified as monotone logics ([22, 30, 5, 6]) and non-monotone logics ([33, 16]). These logics have influenced the work presented here but the most direct source is [22].

Turning now to comments on the logic presented here. The natural deduction style of proof appears to result in proofs which are easy to follow at different levels of detail. Thus, it is possible to both read and sketch proofs without going to a level where each step is reduced to the use of a single proof rule. Because the law of the "excluded middle" does not hold in this three-valued logic, some care is necessary in providing a natural deduction proof style. It is possible to sketch proofs by making deductions which are known to be valid in the model theory (derived rules can, if necessary, be proved later).

Even given the basic set of axioms, it was not immediately apparent how to avoid clouding proofs with case distinctions concerning undefined. Key steps towards this goal include the chosen style of the induction axiom (as opposed to one using implication in the inductive step) and the rules ($d1$ and $d2$) for the recursive function subp.

The example in which the predicates can be undefined (path, $is-wf$) has not been taken further in this paper. It does look as though it could result in the use of more general bounded quantifiers of the form:

$$\forall x \underline{st}\, p(x) . q(x).$$

Such a theory would not be difficult to formalize in the style adopted in this paper.

An objection which could be raised against the use, in specifications, of the (non-strict) logical connectives is that they are not realizable in programming languages. This problem occurs with any non-strict operators: the conditional logical operators with their sequential interpretations are not always available either. In fact, many programming languages are defined so that the question of whether a compiler evaluates the second operand of a conjunct with a first operand which evaluates to false is left open. There is then a problem of implementing programs developed using the logic presented here. There is a similar situation which is already well-known: the (unbounded) arithmetic operators are also not available in any programming language. In fact, the situations are similar. In both cases it is necessary to undertake a last step of development in which the "ideal" operators are mapped onto those of the programming language. Clearly, it will normally be necessary to use conditional expressions to realize the non-strict propositional connectives.

Work on algebraic presentations of data types also has to be concerned with questions of weak and strong equality - see, for example, [7]. Other important treatments are based on intuitionism [10, 27, 31].

It is believed that the logic presented here will make it easier to present proofs of programs. For brevity, this paper has been confined to recursive functions. Rules for program proofs and their integration with this logic are presented in [19].

Acknowledgements. HB and CBJ are grateful to the SERC for support via research grants and JHC wishes to express his thanks to the Croucher Foundation of Hong Kong for a scholarship supporting his post-graduate studies. Mrs Julie Hibbs typed and patiently revised this text on Vuwriter. Valuable comments were received from the referees, Peter Aczel, Stephan Blamey, Ole-Johan Dahl, Edsger W. Dijkstra, Ian Hayes, A. Hoogewijs, Berndt Krieg-Bruckner, Carroll Morgan, Olaf Owe, Gordon Plotkin, Dana Scott, Stephan Sokolowski and Michael Spivey.

Appendix I. Monotone Operators

Conventions

(1) E, $E1$, ... denote logical expressions
(2) $x, y, \ldots$ denote variables over proper elements in a universe

(3) $c, c1, \ldots$ denote constants over proper elements in a universe
(4) $s, s1, \ldots$ denote terms which may contain partial functions
(5) $p(x)$ denotes a formula in which x occurs free
(6) $p(s/x)$ denotes a formula obtained by substituting all occurrences of x by s in p. If a clash between free and bound variables would occur, suitable renaming is performed before the substitution.
(7) $p[s2/s1]$ denotes a formula obtained by substituting some occurrence of $s1$ by $s2$. If a clash between free and bound variables would occur, then suitable renaming is performed before the substitution.

Basic Operators

Name	*Rule*
$\vee -I$	$\dfrac{Ei}{E1 \vee E2} \quad (1 \leqq i \leqq 2)$
$\vee -E$	$\dfrac{E1 \vee E2,\ E1 \vdash E,\ E2 \vdash E}{E}$
$\sim \vee -I$	$\dfrac{\sim E1,\ \sim E2}{\sim(E1 \vee E2)}$
$\sim \vee -E$	$\dfrac{\sim(E1 \vee E2)}{\sim Ei} \quad (1 \leqq i \leqq 2)$
$\sim \sim -I$	$\dfrac{E}{\sim \sim E}$
$\sim \sim -E$	$\dfrac{\sim \sim E}{E}$
contr.	$\dfrac{E1,\ \sim E1}{E2}$
$\exists -I$	$\dfrac{p(s/x),\ s=s}{\exists x \cdot p(x)}$
$\exists -E$	$\dfrac{\exists x \cdot p(x),\ p(y/x) \vdash E}{E}$ (y is arbitrary and not free in E)
$\sim \exists -I$	$\dfrac{\sim p(x)}{\sim \exists x \cdot p(x)}$ (x is arbitrary)
$\sim \exists -E$	$\dfrac{\sim \exists x \cdot p(x),\ s=s}{\sim p(s/x)}$
$=$-subs.	$\dfrac{s1=s2,\ p}{p[s2/s1]}$
$=$-contr.	$\dfrac{\sim(s=s)}{E}$
$=$-cons.	$\dfrac{}{c=c}$

$=$-var. $\dfrac{}{x=x}$

consts. $\dfrac{\sim t}{E}\ \dfrac{}{t}\ \dfrac{u}{E}\ \dfrac{\sim u}{E}$

$=$-reflx. $\dfrac{s1=s2}{si=si}\quad (1\leqq i\leqq 2)$

$\sim=$-reflx. $\dfrac{\sim(s1=s2)}{si=si}\quad (1\leqq i\leqq 2)$

$=$-2-val. $\dfrac{s1=s1,\ s2=s2}{s1=s2\vee\sim(s1=s2)}$

Definitions of Other Connectives

f-defn. $\dfrac{\sim t}{\overline{\overline{f}}}$

$\wedge$-defn. $\dfrac{\sim(\sim E1\vee\sim E2)}{\overline{\overline{E1\wedge E2}}}$

$\Rightarrow$-defn. $\dfrac{\sim E1\vee E2}{\overline{\overline{E1\Rightarrow E2}}}$

$\Leftrightarrow$-defn. $\dfrac{\mathrm{E1}\dashv\mathrm{E2}\wedge\mathrm{E2}\dashv\mathrm{E1}}{\overline{\overline{\mathrm{E1}\Leftrightarrow\mathrm{E2}}}}$

$\forall$-defn. $\dfrac{\sim\exists x\cdot\sim p(x)}{\overline{\overline{\forall x\cdot p(x)}}}$

δ-defn. $\dfrac{E\vee\sim E}{\overline{\overline{\delta E}}}$

Derived Rules

$\wedge-I$ $\dfrac{E1,E2}{E1\wedge E2}$

$\wedge-E$ $\dfrac{E1\wedge E2}{Ei}\quad (1\leqq i\leqq 2)$

$\sim\wedge-I$ $\dfrac{\sim Ei}{\sim(E1\wedge E2)}\quad (1\leqq i\leqq 2)$

$\sim\wedge-E$ $\dfrac{\sim(E1\wedge E2),\ \sim E1\vdash E,\ \sim E2\vdash E}{E}$

comm. $\dfrac{E1\vee E2}{E2\vee E1}\qquad\dfrac{E1\wedge E2}{E2\wedge E1}$

ass. $\dfrac{(E1\vee E2)\vee E3}{\overline{\overline{E1\vee(E2\vee E3)}}}\quad\dfrac{(E1\wedge E2)\wedge E3}{\overline{\overline{E1\wedge(E2\wedge E3)}}}$

It is now legitimate to use n-fold versions of $\vee -I/E$, etc. For example:

$\wedge -I$ $\quad \dfrac{E1, E2, \ldots, En}{E1 \wedge E2 \wedge \ldots \wedge En}$

$\vee \wedge$-dist. $\quad \dfrac{E1 \vee (E2 \wedge E3)}{(E1 \vee E2) \wedge (E1 \vee E3)}$

$\wedge \vee$-dist. $\quad \dfrac{E1 \wedge (E2 \vee E3)}{(E1 \wedge E2) \vee (E1 \wedge E2)}$

deM. $\quad \dfrac{\sim(E1 \wedge E2)}{\sim E1 \vee \sim E2} \quad \dfrac{\sim(E1 \vee E2)}{\sim E1 \wedge \sim E2}$

$\Rightarrow -I$ $\quad \dfrac{E1 \vdash E2, \delta E1}{\sim E1 \vee E2}$ or $\dfrac{E1 \vdash E2, \delta E1}{E1 \Rightarrow E2}$

$\Rightarrow -E$ $\quad \dfrac{\sim E1 \vee E2, E1}{E2}$ or $\dfrac{E1 \Rightarrow E2, E1}{E2}$

$\Rightarrow$-vac. $\quad \dfrac{E2}{E1 \Rightarrow E2} \quad \dfrac{\sim E1}{E1 \Rightarrow E2}$

$\Rightarrow$-contrp. $\quad \dfrac{E1 \Rightarrow E2}{\sim E2 \Rightarrow \sim E1}$

$\Leftrightarrow$-ass. $\quad \dfrac{(E1 \Leftrightarrow E2) \Leftrightarrow E3}{E1 \Leftrightarrow (E2 \Leftrightarrow E3)}$

$\forall -I$ $\quad \dfrac{p(x)}{\forall x \cdot p(x)}$ (x is arbitrary)

$\forall -E$ $\quad \dfrac{\forall x \cdot p(x), s = s}{p(s/x)}$

$\sim \forall -I$ $\quad \dfrac{\sim p(s/x), s = s}{\sim \forall x \cdot p(x)}$

$\sim \forall -E$ $\quad \dfrac{\sim \forall x \cdot p(x), \sim p(y/x) \vdash E}{E}$ (y is arbitrary and bound in E)

$\quad \dfrac{\forall x \cdot p(x)}{p(y/x)} \quad \dfrac{\sim \exists x \cdot p(x)}{\sim p(y/x)}$

$=$-comm. $\quad \dfrac{s1 = s2}{s2 = s1}$

$=$-trans. $\quad \dfrac{s1 = s2, s2 = s3}{s1 = s3}$

Appendix II. Non-Monotone Operators

Name	*Rule*
$\Delta -I$	$\dfrac{E}{\Delta E} \quad \dfrac{\sim E}{\Delta E}$

$$\Delta\text{-}E \qquad \frac{\Delta E, E \vdash E1,\ \sim E \vdash E1}{E1}$$

$$\sim\Delta\text{-}I \qquad \frac{\Delta E \vdash E1,\ \Delta E \vdash \sim E1}{\sim\Delta E}$$

$$\sim\Delta\text{-}E \qquad \frac{\sim\Delta E \vdash E1,\ \sim\Delta E \vdash \sim E1}{\Delta E}$$

Definition of $==$

$$==\text{-defn.} \qquad \frac{s1 == s2}{(s1 = s2 \wedge \Delta(s1 = s1 \wedge s2 = s2)) \vee \sim(\Delta(s1 = s1) \vee \Delta(s2 = s2))}$$

Derived Rule

$$==\text{to}= \qquad \frac{s1 == s2,\ si = si}{s1 = s2} \quad (1 \leqq i \leqq 2)$$

References

1. Abrial, J.R.: Formal Programming. Privately circulated, March 1982
2. Abrial, J.R.: A Theoretical Foundation to Formal Programming. Privately circulated, May 1982
3. Bird, R.: Programs and Machines. New York: John Wiley, 1976
4. Bjorner, D., Jones, C.B.: Formal Specification and Software Development. New York: Prentice-Hall, 1982
5. Blamey, S.R.: Partial-Valued Logic. Ph.D. Thesis, Oxford University, 1980
6. Blamey, S.R.: Partial Logic. In: Handbook of Philosophical Logic. (To appear)
7. Broy, M., Wirsing, M.: Partial Abstract Types. Acta Informat. **18**, 47–64 (1982)
8. Constable, R.L., O'Donnell, M.J.: A Programming Logic. Cambridge, MA: Winthrop Publishers, 1978
9. Constable, R.L., Johnson, S.D., Eichenlaub, C.D.: An Introduction to the PL/CV2 Programming Logic. Berlin, Heidelberg, New York: Springer 1982
10. Constable, R.L.: Partial functions in Constructive Formal Theories. Lecture Notes in Computer Science, Vol. 145. Berlin, Heidelberg, New York: Springer, 1983
11. Dijkstra, E.W.: A Discipline of Programming. New York: Prentice-Hall, 1976
12. Gordon, M.J., Milner, R., Wadsworth, C.P.: Edinburgh LCF. Lecture Notes in Computer Science, Vol. 78. Berlin, Heidelberg, New York: Springer, 1979
13. Gries, D.: Science of Programming. Berlin, Heidelberg, New York: Springer, 1981
14. Haack, S.: Deviant Logic. Cambridge: Cambridge University Press, 1974
15. Haack, S.: Philosophy of Logics. Cambridge: Cambridge University Press, 1978
16. Hoogewijs, A.: On a Formalization of the Non-definedness Notion. Z. Math. Logik **25**, 213–221 (1979)
17. Jones, C.B.: Formal Development of Correct Algorithms: An Example Based on Earley's Recognizer. SIGPLAN **7**, (1972)
18. Jones, C.B.: Software Development: A Rigorous Approach. London: Prentice-Hall, 1980
19. Jones, C.B.: Systematic Program Development. Talk given at CWI Amsterdam December 1983. (In press)
20. Kleene, S.C.: Introduction to Metamathematics. Princeton: Van Nostrand, 1952
21. Kleene, S.C.: Mathematical Logic. New York: John Wiley, 1967
22. Koletsos, G.: Sequent Calculus and Partial Logic. M.Sc. Thesis, Manchester University, 1976
23. Lucas, P., Walk, K.: On the Formal Description of Pl/I. Ann. Rev. Automatic Progr. **6**, 105–182 (1969)

24. McCarthy, J.: A Basis for a Mathematical Theory of Computation. In: Computer Programming and Formal Systems. P. Braffort, D. Hirschberg (eds.). Amsterdam: North-Holland, 1967
25. Manna, Z.: Mathematical Theory of computation. New York: McGraw Hill, 1974
26. Owe, O.: Program Reasoning Based on a Logic for Partial Functions. Privately circulated, 1982
27. Plotkin, G.D.: Types and Partial Functions. Seminar, University of Manchester, 1984
28. Prawitz, D.: Natural Deduction. Stockholm: Almgrist and Wiksell, 1965
29. Rescher, N.: Many-valued Logic. New York: McGraw Hill, 1969
30. Scott, D.S.: Combinators and Classes. In: Lecture Notes in Computer Science, Vol. 37. Böhm, C. (ed.). Berlin, Heidelberg, New York: Springer, 1975
31. Scott, D.S.: Identity and Existence in Intuitionistic Logic. In: Lecture Notes in Mathematics, Vol. 735. Berlin, Heidelberg, New York: Springer, 1979
32. Wang, H.: The Calculus of Partial Predicates and Its Extension to Set Theory I. Math. Logik 7, 283–288 (1961)
33. Woodruff, P.: Logic and Truth-Value Gaps in Philosophical Problems in Logic. Lambert, K. (ed.). Reidel, 1970

Received September 27, 1983 / June 18, 1984