On the usability of logics which handle partial functions*

J.H. Cheng[†], C.B. Jones[‡]

September 15, 1994

Abstract

Partial functions are common in computing – they arise from recursive definitions and from the basic operators of data types like sequences. The need to prove results about logical formulae which include terms using partial functions must cast doubt on the applicability of classical logic. Approaches to handling partial functions which attempt to salvage classical logic include avoiding function application and the use of variant equality notions. The alternative is to accept the need for a non-classical logic. The most frequently used non-classical logic adopts non-strict (but also non-commutative) conditional versions of the propositional operators. An alternative – which is more fully explored below – uses commutative and/or operators as defined in the truth-tables which Kleene attributes to Lukasiewicz. A consistent and complete proof system for such a logic has been used in the program development method known as VDM. An important feature of this approach is the way the definitions of partial functions are brought into proofs. This paper analyzes different approaches to reasoning about partial functions. In particular, it compares the usability of different logics in handling the sort of partial functions which arise in the specification and design of computer software.

*This is a reprint of a paper which is to be printed in the *Proceedings of the Third Refinement Workshop*, (editors) C. Morgan and J. Woodcock, Springer-Verlag, 1990. As a courtesy to the publishers, that version should be cited.

[†]I.S.T., Cambridge

[‡]Department of Computer Science, University of Manchester

Contents

1 Introduction

A partial function is one whose application does not necessarily yield a (defined) value. Partial functions are often defined recursively; for example:

$$zero : \mathbb{Z} \to \mathbb{Z}$$
$$zero(i) \quad \triangleq \quad \textbf{if } i = 0 \textbf{ then } i \textbf{ else } zero(i-1)$$

defines a function for which the property:

$$\forall i \in \mathbb{Z} \cdot i \geq 0 \ \Rightarrow \ zero(i) = 0$$

might be expected to hold ($\mathbb{Z}$ is the set of integers). Closer investigation discloses, however, that this logical expression is problematic with the classical logical operators and notion of equality: the bound on i permits a value of -1 for which, although the antecedent of the implication is false, the consequent contains a term ($zero(-1)$) which – under reasonable interpretations of the definition of *zero* – does not denote an integer. In order to make this plausible quantified expression have a meaning, some way of handling terms which include such references to partial functions must be found. In this simple example, the problem could be circumvented by changing the bound on the quantifier to range over the natural numbers ($\mathbb{N}$). Another alternative is to define equality so as to prevent the undefinedness of the term causing an undefined logical expression. In [BCJ84], a logic is discussed in which non-strict logical operators handle the difficulty.

This paper analyzes different approaches to the construction of formal proofs about partial functions. The aim is to discuss alternative approaches and to compare some of the problems which arise in their application to various examples. The body of the paper (Sections 2–9) presents the main approaches found in the literature; Section 10 discusses a number of related papers; and the conclusion (Section 11) indicates why the current authors believe their approach to be usable in software design.

Before embarking on the analysis of different approaches, it is worth taking a closer look at the equality relation. In particular, recognising that different notions of equality are appropriate in different contexts is the key to some approaches to handling partial functions. (One might want to avoid manipulating several forms of equality in proofs about functions, but the underlying distinction should first be understood.) It should be clear that the = within the conditional expression must be computable and therefore strict (i.e. yields $\perp$ if either operand is $\perp$). Writing the undefined values as 'bottom' elements ($\perp$), its truth-table is:

$=$	0	1	2	$\dots$	$\perp_{\mathbb{N}}$
0	**true**	**false**	**false**		$\perp_{\mathbb{B}}$
1	**false**	**true**	**false**		$\perp_{\mathbb{B}}$
2	**false**	**false**	**true**		$\perp_{\mathbb{B}}$
$\vdots$					
$\perp_{\mathbb{N}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$		$\perp_{\mathbb{B}}$

The definitional equality needs a different interpretation. Operationally speaking, applying the definition on *zero* to -1 rewrites to exactly $zero(-2)$: they are strongly equal in the sense that the former is $\perp$ precisely because the latter is $\perp$. The truth-table for this non-strict operator is:

$==$	0	1	2	$\ldots$	$\perp_{\mathbb{N}}$
0	**true**	**false**	**false**		**false**
1	**false**	**true**	**false**		**false**
2	**false**	**false**	**true**		**false**
$\vdots$					
$\perp_{\mathbb{N}}$	**false**	**false**	**false**		**true**

It is, of course, now standard to view recursive definitions like that for *zero* as denoting the least-fixed point of their defining equation (see [Man74, Sto77, Sch86]). But, since the least-fixed point in this case is $\{(i,\ 0) \mid i \in \mathbb{N}\}$, this does nothing to resolve the problem of the non-denoting term.

Apart from recursively defined functions, many basic functions (which are often written as prefix or infix operators) in software specification and design are also partial. For example, in the notation of [Jon90], writing **hd** for the head of a sequence, **tl** for its tail, and [] for an empty sequence:

$$\forall t \in \mathbb{N}^* \cdot t = [\,] \vee t = append(\mathbf{hd}\, t, \mathbf{tl}\, t)$$

has non-denoting terms in the second disjunct if the first is true; or, with $pred: X \times X \to \mathbb{B}$: an expression like:

$$\mathbf{len}\, t_1 = \mathbf{len}\, t_2 \wedge \forall i \in \mathbf{dom}\, t_1 \cdot pred(t_1(i), t_2(i))$$

can index outside the sequence t_2 if it is shorter than t_1 (and, thus, the first conjunct is false); or again, with $\rho \in (Id \xrightarrow{m} den)$:

$$id \in \mathbf{dom}\, \rho \wedge \rho(id) \in Proctype$$

the map application can be undefined if the first conjunct is false. (Such expressions occur often, for example, in [BJ82].)

Here, it can be seen that, if the partial terms yield $\perp$, the strict logical operators in disjunctions and conjunctions are inadequate. In the last example, it is also clear that re-interpreting equality does not help since $\in$ is the operator whose first operand is a potentially undefined term.

The following sections analyze various approaches to such examples. Section 6 introduces further problems concerned with potentially undefined predicates being used as the hypotheses of sequents ($\Gamma \vdash p$, where Γ denotes a list of formulae which is the hypothesis of the sequent). The approaches in Sections 2, 3, and 4 manage, in various ways, to live with the standard Predicate Calculus; those in Sections 5, 6, and 7 adopt non-standard logics of one kind or another; the approaches in Sections 8 and 9 are influenced by Scott's domain theory, but the former uses a layered-logic approach whereas the latter uses a constructive free logic.

2 Totality over a restricted set

As is hinted in the introduction, it is sometimes possible to find a set over which the problem of partiality disappears; thus:

$$\forall n \in \mathbb{N} \cdot zero(n) = 0$$

But for other functions, such as:

$$subp : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

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

reformulation of properties like:

$$\forall i, j \in \mathbb{Z} \cdot i \geq j \;\Rightarrow\; subp(i,j) = i - j$$

requires cumbersome expressions such as:

$$\forall i, j \in \{(i,j) \in \mathbb{Z} \times \mathbb{Z} \mid i \geq j\} \cdot subp(i,j) = i - j$$

(One of the current authors attempted to carry through such a programme using bounded quantifiers in [Jon80].) Another difficulty is that the use of bounds which are given by such general set comprehensions makes the type structure undecidable. As discussed in [Lin87], this results in type inferences like (writing $(X \mid p)$ for the type whose elements are in $\{x{:}\, X \mid p(x)\}$):

$$\frac{x \in X;\; p(x)}{x \in (X \mid p)}$$

which could introduce many extra steps into proofs. This problem can be reduced if references to partial functions occur in carefully guarded contexts. Thus, if conditionals are used, inference rules like:

$$\textbf{if-}I\; \frac{p \;\vdash\; e_1 \in Y;\; \neg\, p \;\vdash\; e_2 \in Y}{(\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2) \in Y}$$

can simplify proofs.

Clearly, the sequence property can be written in terms of a *Nonemptylist* set:[1]

$$\forall t \in \mathit{Nonemptylist} \cdot t = \mathit{append}(\mathbf{hd}\, t, \mathbf{tl}\, t)$$

It is, however, less clear how to translate the property on mappings using bounded quantifiers. In [Jon80] conditional expressions were used as an additional (non-strict) logical operator permitting:

$$\textbf{if } id \in \textbf{dom}\, \rho \textbf{ then } \rho(id) \in \mathit{Proctype} \textbf{ else false}$$

So bounded quantifiers can be used to avoid some of the problems of partial functions in logical expressions: one must ensure by the bound sets that functions are never applied outside their domains. If suitable constraints could always be given, this would ensure that only the classical (strict) logical operators would be required. These constraints cannot, unfortunately, always be found and it is necessary to introduce at least one non-strict operator (e.g. conditional expressions). It would thus be necessary to develop a proof theory which linked the classical operators and quantifiers to the conditional expressions. Perhaps most worrying is the observation that the attempt to formulate logical expressions with such bounds causes gratuitous difficulty in proof construction. These observations have led a number of researchers to move to the approach – discussed in Section 5 below – in which the conditional concept is absorbed into the meaning of the logical operators.

[1] It should be noted that it is no longer possible to treat bounded quantification as an abbreviation ($\forall x \in X \cdot p(x)$ as $\forall x \cdot x \in X \;\Rightarrow\; p(x)$, and $\exists x \in X \cdot p(X)$ as $\exists x \cdot x \in X \wedge p(x)$) but it is possible to define inference rules for bounded quantifier introduction which do not rely on these abbreviations. (In fact, this has been done for the Logic of Partial Functions discussed in Section 6.)

3 Viewing functions as relations

Another way of preserving the strict (classical) propositional operators is to avoid function application! Publications on the specification language known as 'Z' [Abr84b, Hay87, Spi88] appear to use more than one way of handling partial functions. At least [Abr87] proposes avoiding application in the case of recursively defined functions. Instead of writing $f(x) = y$, the function can be treated as its graph and the proposition formulated by testing whether the pair (x, y) is a member of that graph. Thus, the property about *zero* can be formulated:

$$\forall i \in \mathbb{Z} \cdot i \geq 0 \;\Rightarrow\; (i, 0) \in zero$$

Since $(-1, j) \in zero$ is false for any j, the classical (two-valued) interpretation of implication is adequate.

In constructing proofs about recursive functions, an obvious way to treat their definitions is to view them as inducing inference rules:

$$zero\text{-}b \;\frac{}{(0,0) \in zero}$$

$$zero\text{-}i \;\frac{i \neq 0; (i-1, k) \in zero}{(i, k) \in zero}$$

In order to express the property for *subp*, one has to either Curry the function or treat it as a function whose domain is an element of a Cartesian product:

$$\forall i, j \in \mathbb{Z} \cdot i \geq j \;\Rightarrow\; ((i, j), i - j) \in subp$$

It is unclear how easily one could train oneself to find such expressions as convenient as function applications. Notice that it is also necessary to decide which applications are to be formulated in terms of membership of the graph of the function (all? any, potentially, partial? recursive only?). Furthermore, the reformulation of expressions which are more complex than $f(x) = y$ often needs extra quantifiers: for $\rho(id) \in Proctype$, it is necessary to write $\exists y \cdot (x, y) \in \rho(id) \wedge y \in Proctype$ (and it is an extra property of function graphs that avoids the need to write this as $\forall y \cdot (x, y) \in \rho(id) \;\Rightarrow\; y \in Proctype$).

4 Fixing notions of equality

The virtue of the proposals in the preceding sections is that they stay within the framework of classical (two-valued) logic; their disadvantage is that they require expressions and proofs involving partial functions to be written in ways which differ significantly from those which are familiar for total functions. This section presents another way of preserving the classical logic operators.

As can be seen from the examples given above, the potentially undefined terms often occur as operands of an equality relation. Can one use different notions of equality? If it can be arranged that the equality operator always yields a Boolean value, the classical logical operators can be used. Such an equality operator can obviously not be strict. The most obvious choice is to use the strong equality discussed in Section 1. (This approach is used in [Bro87b].) Thus, the property to be proved about *subp* is written:

$$\forall i, j \in \mathbb{Z} \cdot i \geq j \;\Rightarrow\; subp(i, j) == i - j$$

The definition of $subp$ is treated as being defined by the same equality:

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

and the following two inference rules are used for reasoning about conditional expressions.

$$\textbf{if}\text{-}E1 \; \frac{p}{(\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2) == e_1}$$

$$\textbf{if}\text{-}E2 \; \frac{\neg\, p}{(\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2) == e_2}$$

The proof of the required property is presented in Figure 1 in the style of [Jon90] and uses only classical inference rules for the predicate calculus.

from $i, j \in \mathbb{Z}$		
1	$i = i - 0$	h, $\mathbb{Z}$
2	$(\textbf{if } i = i - 0 \textbf{ then } 0 \textbf{ else } subp(i, (i-0)+1) + 1) == 0$	**if**-$E1$(1)
3	$subp(i, i-0) == \textbf{if } i = i - 0 \textbf{ then } 0 \textbf{ else } subp(i, (i-0)+1) + 1$	$subp$-$defn$
4	$subp(i, i-0) == 0$	$==$ -$subs$(2,3)
5	**from** $n \in \mathbb{N}$; $subp(i, i-n) == n$	
5.1	$i - (n+1) \in \mathbb{Z}$	h, h5, $\mathbb{Z}$
5.2	$i \neq i - (n+1)$	h, h5, $\mathbb{Z}$
5.3	$(\textbf{if } i = i - (n+1) \textbf{ then } 0 \textbf{ else } subp(i, i-(n+1)+1) + 1)$ $== subp(i, i-(n+1)+1) + 1$	**if**-$E2$(5.2)
5.4	$== subp(i, i-n) + 1$	$\mathbb{Z}$, 5.3
5.5	$== n + 1$	$==$ -$subs$(h5,5.4)
5.6	$subp(i, i-(n+1))$ $== \textbf{if } i = i - (n+1) \textbf{ then } 0 \textbf{ else } subp(i, i-(n+1)+1) + 1$	$subp$-$defn$
	infer $subp(i, i-(n+1)) == n+1$	$==$ -$subs$(5.6,5.5)
6	$\forall n \in \mathbb{N} \cdot subp(i, i-n) == n$	$\forall$-I($\mathbb{N}$-ind(4,5))
7	**from** $i \geq j$	
7.1	$(i - j) \in \mathbb{N}$	$\mathbb{N}$, h7
	infer $subp(i,j) == i - j$	$\forall$-E(6,7.1), $\mathbb{Z}$
infer $i \geq j \;\Rightarrow\; subp(i,j) == i - j$		$\Rightarrow$ -I(7)

Figure 1: Proof about $subp$ using strong equality

While it is true that the overall structure of this proof is similar to that which might be used for total functions, there do remain some difficulties for the authors of such proofs. The principal difficulty is that both writers and readers of such proofs are forced to be aware of two notions of equality. The (computable) weak equality needed within the definition of a function must not be confused with the strong equality needed to reason about functions. This burden is unacceptable to a reader who views the implication as guarding the evaluation: it is after all true that, for appropriate values, $subp(i,j) = i - j$ (with the weak equality).

Nor does the problem end here, intuitively clear properties such as:

$$\begin{array}{l} \forall i, j \in \mathbb{Z} \cdot i > j \;\Rightarrow\; subp(i,j) > 0 \\ \forall i, j \in \mathbb{Z} \cdot i \geq j \;\Rightarrow\; subp(i,j) \in \mathbb{N} \\ \forall i, j \in \mathbb{Z} \cdot i \geq j \;\Rightarrow\; subp(i+1,j) \neq subp(i,j) \text{ etc.} \end{array}$$

would all need to be reformulated before proof could even be considered. Such reformulation could be handled by introducing strong versions of their operators. Alternatively it can be done systematically by reading any term t, for which a Boolean value is required, as though it were t == **true**. But this process of redefining 'innermost' terms must be handled with care. One might expect $subp(i,j) = i - j \vee subp(i,j) \neq i - j$ to be true but an interpretation using $(subp(i,j) \neq i - j) == \mathbf{true}$ would yield false for some values. (The expected result requires that $\neq$ is carefully interpreted such that the above expression becomes $subp(i,j) = i-j \vee \neg\,(subp(i,j) = i-j)$ before the other rewriting is handled.) Furthermore, the rewriting in the case of $subp(i,j) < i-j \vee subp(i,j) \geq i-j$ is even less clear. Another problem with applying the rewriting to innermost terms is that $\mathbf{hd}\,[\,] = 5 \;\Rightarrow\; \mathbf{hd}\,[\,] < 3$ becomes true!

The preservation of the classical logical operators has been purchased at the cost of more complex relational operators. As an alternative, it is possible to view the 're-reading' as being applied only where a truth value is required by another context (e.g. the test of a conditional). Unfortunately this re-writing of 'outermost' terms leaves the problem of a collection of Boolean operators which handle undefined.

It was pointed out to one of the current authors (CBJ) by Oliver Schoett that strong equality is not the only operator which could be used to preserve the classical two-valued calculus. *Existential equality* yields **false** if either operand is undefined; it is again not strict; nor is it monotonic; it differs from strong equality only in that it is false if both operands are undefined. Existential equality is used in PROSPECTRA [Kri88] and would yield proofs similar to those in Figure 1 and difficulties similar to those for ==. In fact, it is possible to define:

$$e_1 == e_2 \qquad \text{iff} \qquad e_1 =_\exists e_2 \vee (\neg\,(e_1 =_\exists e_1) \wedge \neg\,(e_2 =_\exists e_2))$$

Also:

$$(e_1 = e_2) == \mathbf{true} \qquad \text{iff} \qquad e_1 =_\exists e_2$$

As can be seen, existential equality has the additional disconcerting property that it is not reflexive and $\neg\,(e_1 =_\exists e_2)$ is true if either operand is undefined.

5 Conditional operators

The difficulties seen in the preceding sections suggest that it is at least worth considering a change to the meaning of the logical operators so that they handle undefined values. Section 2 has already mentioned the idea of using the conditional expression form as an operator. This non-strict (in its first argument) form can be used to reformulate properties such as:

$$\begin{array}{l} \forall i, j \in \mathbb{Z} \cdot \mathbf{if}\ i \geq j\ \mathbf{then}\ subp(i,j) = i - j\ \mathbf{else\ true} \\ \forall t \in \mathbb{N}^* \cdot \mathbf{if}\ t \neq [\,]\ \mathbf{then}\ t = append(\mathbf{hd}\,t, \mathbf{tl}\,t)\ \mathbf{else\ true} \end{array}$$

But this clearly presents a significant step away from normal predicate calculus proofs. Already in the 1960's, McCarthy suggested [McC67] staying close to predicate calculus by redefining all of the classical logical operators in terms of conditional expressions. Conditional forms of or/and/implies can be defined:

$$E_1 \textbf{ cor } E_2 \quad \text{iff} \quad \textbf{if } E_1 \textbf{ then true else } E_2$$
$$E_1 \textbf{ cand } E_2 \quad \text{iff} \quad \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else false}$$
$$E_1 \textbf{ cimpl } E_2 \quad \text{iff} \quad \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else true}$$

The obvious reformulations of the properties of sequences, maps and *subp* all become valid. These operators are monotone in the ordering shown in Figure 2 which has the pleasing computational intuition that a result can be computed if the first operand provides enough information even if the computation of the second operand is incomplete; furthermore, the latter's completion cannot result in a contradiction to the computed value. Here, however (cf. next section), a left-to-right evaluation order is implied (this is sometimes referred to as a 'sequential interpretation').

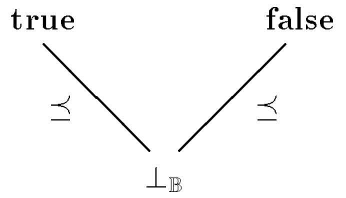

Figure 2: Ordering for Truth Values

With so many many pleasing attributes, why should one seek further? For any non-classical version of a logic, one has to ask which properties no longer hold. Here, the conditional forms of and/or do not enjoy one of the most intuitive properties expected of such operators: their operands can not be arbitrarily commuted. (Similarly, the contrapositive of an implication can fail.) In order to facilitate proofs, one of the current authors used both classical ($\wedge$) and conditional (&) forms in [Jon72]; a similar distinction is used (**and** vs. **cand**) in [Dij76]. (The programming language 'Standard ML' [HMT88] uses **andalso**.) None of these references, unfortunately, provides a full proof theory for the combined logic. What gives rise to greater concern is that there are some surprises in the identities. Whilst forms of de Morgan's law hold:

$$\neg(E_1 \textbf{ or } (E_2 \textbf{ cand } E_3)) \quad \text{iff} \quad \neg E_1 \textbf{ and } (\neg E_2 \textbf{ cor} \neg E_3)$$

and **cand** left distributes over **cor**:

$$E_1 \textbf{ cand } (E_2 \textbf{ cor } E_3) \quad \text{iff} \quad E_1 \textbf{ cand } E_2 \textbf{ cor } E_1 \textbf{ cand } E_3$$

more care is needed to see that the following property – used without comment in a program development in [Gri81] – holds:

$$E_1 \textbf{ and } (\neg E_1 \textbf{ cor } E_2) \quad \text{iff} \quad E_1 \textbf{ cand } E_2$$

(Also, for example, **cand** does *not* right distribute over **cor**.) There is also a difficult decision to be made about the interpretation of quantifiers since there is no linear text to suggest the order of evaluation (this issue is scrutinized in [Bli88]).

On balance, this first attempt to adopt a logic which handles partial functions appears no easier to use than approaches which attempt to preserve classical logic. (This comment is made in the light of the experience in writing [Jon72].)

6 LPF

It is not difficult to discover other logics which handle partial functions but it should be clear from the preceding section that their acceptance will be decided on their algebraic properties. The particular 'Logic of Partial Functions' (hereafter, 'LPF') presented in [BCJ84] has a distinguished parentage – much of which was made known to the current authors after the publication of [BCJ84]. Kleene [Kle52] attributes a collection of symmetrical 'three-valued' operators to Łukasiewicz; Blikle [Bli88] traces this reference back to [Luk20]; this logic was further studied in [Kol76] which was brought to the current authors' attention by Peter Aczel. But [Dri88] attributes such logics to MacColl and [Urq86] finds foreshadowings in the work of Boole, Pierce and Vasiliev but concedes that the work was focussed by Post and Łukasiewicz. One of the current authors developed the logic of [BCJ84] in his doctoral thesis ([Che86] formalises LPF as a first-order predicate calculus with equality in a sequent calculus with a set-theoretical semantics, upon which completeness and consistency are established. A cut-elimination theorem is proven.) The logic is used in VDM in [Jon86, Jon90, JS90]. This work has evoked some further interest [Gib88, Ten87, Avr88, Jer88, Hoo87] (Hoogewijs had studied related logics in [Hoo77]).

The basic idea is very simple: the propositional operators are given the strongest monotone (with respect to the ordering in Figure 2) extension of their two-valued interpretations. This removes the left-to-right order required for the conditional expression interpretation of the operators. Having abandoned order, it is natural to also give a 'generous' interpretation to the quantifiers (in terms of [Bli88], Kleene's quantifiers rather than McCarthy's); bound variables in quantified expressions range only over the 'proper' values of their bound set. The resulting logic has no positive surprises for the user since it employs only the normal collection of operators and all truths of LPF hold in classical first-order predicate calculus. Thus, for example, and/or are commutative and distribute over each other in the normal way. The negative surprises – which derive from the fact that the so-called 'law of the excluded middle' (*Tertium non datur*) does not hold – are discussed below.

Having elevated the *subp* result to the level of a challenge problem, the next step is to show its proof using LPF. This is done in Figure 3.[2] As in Section 3, the definition of *subp* is most conveniently handled by inference rules:

$$subp\text{-}b\ \frac{i \in \mathbb{Z}}{subp(i, i) = 0}$$

$$subp\text{-}i\ \frac{i_1, i_2 \in \mathbb{Z};\ i_1 \neq i_2;\ subp(i_1, i_2 + 1) = i_3}{subp(i_1, i_2) = i_3 + 1}$$

A simpler set of inference rules can be used for total functions. For partial functions, the rôle of the weak equality in the hypothesis of *subp-i* is important but one could envisage mechanical generation of such rules from recursive definitions. Notice that this proof only uses the weak equality required by the original statement of the property to be proved. Its only blemish appears to be the need (in line 7) to prove that the hypothesis is defined

[2] Slightly different approaches – each using LPF – are given in [BCJ84] and [Jon86]. One other approach would be to proceed as in Figure 1 and then to use the inference rule which relates strong and weak equality.

before the implication introduction rule can be used. Because the deduction theorem does not hold in general in LPF, the inference rule is:

$$\Rightarrow\text{-}I\ \frac{E_1 \vdash E_2;\ \delta(E_1)}{E_1 \Rightarrow E_2}$$

Where $\delta(E)$ asserts that E is defined (e.g. $E \vee \neg E$). The expressive completeness of the (re-interpreted) logical connectives for all possible monotone truth-valued functions (i.e. for $n \geq 0$, functions of type $\{\mathbf{true}, \mathbf{false}, \perp\}^n \rightarrow \{\mathbf{true}, \mathbf{false}, \perp\}$) is proved in [Kol76]. Thus this part of the logic is both formally and pragmatically expressive. See also [Kol81]. There are proofs – particularly of a meta-nature – which require non-monotonic truth-valued functions. Obviously, some non-monotonic connective has to be added in order to facilitate such expressions: LPF has a definedness connective (Δ), which is first formalised in [Hoo77] and is sufficiently expressive (see also [Che86]; Blamey [Bla80] has pointed out the rôle of conditional expressions in giving a point, in a hierarchy of expressiveness, between strict and monotone operators). The interpretation of sequents in LPF is important: *models* are interpreted as binding the individual free variables in both the hypothesis and conclusion. A sequent is valid if its conclusion is true whenever its hypothesis is. Notice that, when the hypothesis is undefined, there is no constraint on the conclusion: it can be true, false or undefined (see Section 3.2 of [Avr87] for a discussion of 'consequence relations'). This differs from an implication and is the reason why the implication introduction rule has an additional hypothesis.

from $i, j \in \mathbb{Z}$		
1	$i - 0 = i$	h, $\mathbb{Z}$
2	$subp(i, i) = 0$	h, *subp-b*
3	$subp(i, i - 0) = 0$	$=$-*subs*(1,2)
4	**from** $n \in \mathbb{N}; subp(i, i - n) = n$	
4.1	$\quad i - (n + 1) \in \mathbb{Z}$	h, h4, $\mathbb{Z}$
4.2	$\quad i \neq i - (n + 1)$	h, h4, $\mathbb{Z}$
	infer $subp(i, i - (n + 1)) = n + 1$	h, 4.1, 4.2, h4, *subp-i*
5	$\forall n \in \mathbb{N} \cdot subp(i, i - n) = n$	$\forall$-I($\mathbb{N}$-ind(3, 4))
6	**from** $i \geq j$	
6.1	$\quad (i - j) \in \mathbb{N}$	$\mathbb{N}$, h6
	infer $subp(i, j) = i - j$	$\forall$-E(5, 6.1), $\mathbb{Z}$
7	$\delta(i \geq j)$	h, $\mathbb{Z}$
infer $i \geq j \Rightarrow subp(i, j) = i - j$		$\Rightarrow$-I(6,7)

Figure 3: Proof about *subp* in LPF

Does this difference bring any surprises to the user of LPF? Consider:

$evenp : \mathbb{Z} \rightarrow \mathbb{B}$

$evenp(i) \ \triangleq\ \textbf{if } i = 0 \textbf{ then true else } evenp(i - 2)$

With the obvious modulus operator:

$$evenp(i) \ \vdash \ i \ \mathbf{mod} \ 2 = 0$$

is a valid sequent. If the inference rules are generated for *evenp* in the same way as for *subp* above, they are:

$$evenp\text{-}b \ \frac{}{evenp(0)}$$

$$evenp\text{-}i \ \frac{i \neq 0; \ evenp(i-2)}{evenp(i)}$$

But these are *not* sufficient to prove the sequent. They are, in fact, only requiring a fixed point of the definition rather than the least fixed point.[3] Thus $evenp(i) \ \Leftrightarrow \ \mathbf{true}$ satisfies *evenp-b* and *evenp-i*. What is needed to prove a sequent whose hypothesis is $evenp(i)$ is an upper bound and this can be provided by an induction rule. It is not however obvious how such rules can be mechanically generated from the recursive function definitions. Such leastness inference rules are needed in other cases: for example, to show:

$$subp(i,j) \neq i - j \ \vdash \ i < j$$

But notice that the implication $\forall i,j \in \mathbb{Z} \cdot subp(i,j) \neq i - j \ \Rightarrow \ i < j$ is the contrapositive of the 'challenge' result and follows therefrom without difficulty.

7 Owe's 'weak logic'

Owe's work [Owe85] is aimed at providing a logic for reasoning in program development: partial functions are identified with non-terminating subprograms or ones which terminate exceptionally. Motivated by the concept of data abstraction and functional abstraction, three requirements are proposed for such a logic:

1. reasoning about defined formulae should follow the rules of traditional first-order logic;
2. theorems proved at an abstract level should (at least mostly) be valid on a less abstract level; and
3. on an abstract level, specification or identification of so-called implementation dependent errors can be avoided.

With these requirements, 'weak logic' is formalised. There are strictly total connectives such as *and, or, implies* and *not*, as well as non-strict and non-commutative ones like *and then, or else, implies then* and *if-construct*. With such connectives, more traditional (non-strict) connectives such as $\wedge$, $\vee$ and $\Rightarrow$ can be defined.

The propositional formulae and universal quantifications are interpreted in a 'weak semantics'; informally, this means:

> A formula is valid if it is true whenever it is defined.

[3] Notice that, just as with the distinction between weak and strong equality, it is not correct to provide an axiom $evenp(i) \ \Leftrightarrow \ i = 0 \vee evenp(i-2)$.

As a result, if a formula q is undefined, then the formula **false** *and* q is undefined and therefore trivially valid, whereas **false** $\wedge\, q$ is defined and evaluated to **false**.

In contrast, the existential quantifier is interpreted in a 'strong semantics':

$\exists x \in T \cdot q$ is valid if there is a value of x in T such that q is defined and true.

($\forall$ is non-monotonic!) To cope with partial functions a *domain predicate*, denoted as *in f*, is associated with each function f and is interpreted as follows:

x *in* f is true if $f(x)$ is defined, and false otherwise.

The definedness of an expression e can also be specified by a formula, denoted as $DEF[e]$, which is evaluated to **true** when e is defined and **false** otherwise. Proof rules are developed in conjunction with the semantics, and completeness and consistency established.

Application of Weak Logic to the *subp* example would seem, at first sight, to yield a short proof, as no special attention need be given to the undefinedness, which in Weak Logic can be treated as **true**. This is akin to a so-called 'partial correctness' proof in program proofs. However, a separate proof is needed to establish the definedness, as proving termination for 'total correctness' of programs.

8 LCF

This section looks at the approach adopted by LCF – a very well-known and influential system. Based on Scott's domain theory, LCF [MMN75, GMW79] adopts layered-logic approach: separating the computable objects from the logic for reasoning about the objects.

The deductive calculus used in LCF is called *PP-lambda* [MMN75, GMW79]. Terms of *PP-lambda* are those of a typed λ-calculus with a fixed-point combinator, and formulae are:

atomic formulae an equality (==) and a partial ordering (<<) between terms,

conjunction normal (classical) $\wedge$,

implication normal (classical) $\Rightarrow$, and

Universal Quantification $\forall$ ranges over a *domain* (including a bottom element UU, which is a term).

All formulae are supposed to be two-valued, and in particular, $UU == UU$ is evaluated to $TRUTH$.

As to the inference rules of *PP-lambda*, logical rules are basically those classical rules concerning $\wedge$, $\Rightarrow$ and $\forall$. Non-logical rules deal with the equality, the partial ordering, λ-conversions and the fix-point combinator.

Because of the range of quantifiers, we would need, in the *subp* example, to establish in development of the theory of natural numbers an induction principle as follows:

$$\mathbb{N}\text{-}ind \quad \frac{\begin{array}{c} p(UU); \\ p(0); \\ x > 0, p(x-1) \;\vdash\; p(x) \end{array}}{\forall x \in \mathbb{N} \cdot p(x)}$$

assuming $\mathbb{N}$ is the domain of natural numbers with UU embedded. Once this principle has been established (possibly using fixed-point induction), a proof of *subp* in *PP-lambda* is similar to the one in LPF. However, the equality must be strong, and an extra basis step for the induction performed (i.e. we need to prove $subp(i, i - UU) == UU$). This can be established with the non-logical rules. Nevertheless, all proofs appear to involve lines of deduction dealing with the undefined value UU.

Blikle in [Bli88] also considers a 'layered approach', where the 'superpredicates' are used as a bridge between three-valued predicates and the classical logic. Four conceptual levels are distinguished:

1. the level of a software system;
2. the level of a logic of a software system;
3. the level of a definitional metalanguage; and
4. the level of a logic of a definitional metalanguage.

9 Plotkin's 'PFL'

Similar to *PP-lambda* of LCF, Plotkin's PFL (Partial Function Logic) [Plo84] is strongly influenced by Scott's domain theory and is intended to be used to reason about computable functions. But unlike *PP-lambda*, the logic itself is a free logic similar to Scott's [Sco79] (see also [Fef82] and the discussion of 'LPT' in [Bee86]; 'COLD-K' [Jon88, FJO+87, FJ87] also uses a logic based on Scott's ideas). The propositional inference rules are normal. However, because in PFL free variables may not denote but bound variables always do, we have the following rules for the quantifiers:

$$\forall\text{-}I\ \frac{x\downarrow\ \vdash\ p(x)}{\forall x \cdot p(x)}$$

$$\forall\text{-}E\ \frac{\forall x \cdot p(x);\ e\downarrow}{p(e)}$$

$$\exists\text{-}I\ \frac{p(e);\ e\downarrow}{\exists x \cdot p(x)}$$

$$\exists\text{-}E\ \frac{p(x),\ x\downarrow\ \vdash\ q;\ \exists x \cdot p(x)}{q}$$

where $x\downarrow$ means 'x is defined', and similarly for $e\downarrow$.

Due to this treatment of free and bound variables, many references to the undefined can be avoided. For example, in Plotkin's axiomatization of natural numbers, with Scott Induction, the normal induction schema for natural numbers can be derived.

This implies that, for the *subp* example, a proof in PFL would be similar to the one in LPF. The similarity may be just superficial. Underneath, PFL is based on a fairly sophisticated semantics handling ordering, existence, a partial typed λ-calculus, and many other notions.

10 Related papers

In [KTB88] the authors present a three-valued logic, using McCarthy's *CAND* and *COR* and Kleene's quantifiers. Different notions of validity (strong *always true* and weak *never false*) are distinguished; a classification of consequence relations in model-theoretic terms according to different treatments of models and theorems is given (e.g. a logic can have strong models and weak theorems). Several logics are compared with these notions. The proof theory is developed with Beth's semantic tableaux and a transformation to sequent and natural deduction systems. (Similar ground is covered in [GFLC89].)

As is mentioned in Section 3, various approaches to handling partial functions are included in the literature on 'Z'. The main semantics in [Spi88] actually results in an existential equality interpretation (cf. Section 4 above) but Section 4.3 of [Spi88] also contains an excellent discussion of alternatives and a conclusion which indicates why LPF might be thought of as more suitable for VDM whilst 'Z' requires a different approach. He also draws attention to [Abr84a] and its use of 'intentionally incomplete definition of the semantics of predicates'. It would be interesting to see a comparison of this approach with the 'supervaluations' which [Urq86] traces to [vF66] ([Ben86] lists some problems of this approach).

A thorough discussion of logics which cope with undefined terms is contained in [Bla80] or the more accessible [Bla86]. In [Hoo87], a comparison of LPF and the logic PPC (partial-predicated calculus) of [Hoo77] is given; the usefulness of the non-monotone connective is demonstrated. A discussion – in a linguistic setting – of 'free logics' is given in [Ben86]; 'L_3' is proposed in [Sch87] for the study of natural language semantics. The review of approaches in [Avr88] is far easier to relate to the problems of program proofs.

Broy [Bro87a] discusses equational specification of partial (and higher-order) algebras; the paper both formalises 'partial interpretations' and presents a proof theory. In [Bro88], Broy argues for the use of partial algebras because of the messiness of handling error values at the specification level.

A paper which discusses the problems of importing expressions from a programming language into the proof logic (as happens, for example, in Hoare-logic) is [Ten87]. The approach uses quantifiers which range over all (including $\perp$ values); so-called 'strict interpretations' (equivalent to what has been called above 'existential equality'); and a distinguished equality ('strong' in the terms used here) for axioms. Special rules are provided for the manipulation of assertions like $(p \text{and} q) = \textbf{false}$. Jervis also considers partial terms in Hoare-like logics. In particular, [Jer88] investigates the use of LPF. Amongst other interesting results, his Theorem 2.3.8 shows the existence of normal forms for expressions of LPF and his distinctions on sequent notions versus implications are illuminating. Tucker and Zucker argue that weak (strict) logical operators are appropriate for their exhaustive treatment [TZ88] of error-state semantics. Further references (for which the authors are grateful to Professor Ito – whose, as yet unpublished, 'Logic of Execution' is also relevant) are [Sat87, HN88].

11 Conclusions

It would be dangerous for the reader to expect the proponents of one of the approaches to provide a completely balanced consumer survey of the alternatives discussed in Sections 2–9. It is then fairer to state at the outset that this concluding section describes the reasons that the current authors believe that their own approach (cf. Section 6) satisfies the main criteria for reasoning about partial functions in software development.

For any logic, it is clearly desirable to have both model and proof theories; the latter must be consistent with the former and should preferably also be complete (for LPF, these results are covered in [Che86]); other non-standard logics frequently lack a proof theory. The difficulty of using a non-classical logic can be minimised in two ways. The valid sequents of LPF are all consistent with the classical (two-valued) model. Furthermore, any classical tautology can be made true in LPF simply by introducing definedness assumptions for each of its propositions. Conservation in LPF of properties like symmetry of conjunctions and disjunctions and the appropriate distributivity laws could also be regarded as a benefit. Additionally, there would appear to be a strong computational intuition in requiring that the non-strict operators are the strongest monotone (with respect to the ordering in Figure 2) extension of the classical operators.

LPF preserves the (classical) link between $p \Rightarrow q$ and $\neg p \vee q$. Lukasiewicz's truth table for implication gives true for the $\bot \Rightarrow \bot$ case and Wang [Wan61] argues that $p \Rightarrow p$ must be an identity which, in a logic without the law of the excluded middle, is incompatible with the usual definition of implication. The decision in this area is independent of most of the others in LPF and could be changed with little impact.

One objection to LPF is that it is non-standard and unfamiliar ([Fef82] argues against 'three-valued and other restrictive logics due to their debilitating effects on ordinary reasoning'!). In fact, the key difference from classical logic is the lack of the law of the excluded middle. In itself, this is unlikely to cause problems: it is quite valid to introduce (from knowledge of the data structures) a property like: $i > j \vee i \leq j$ (where both i and j are integers); banning disjunctions like: $i/0 = 5 \vee i/0 \neq 5$ is unlikely to be limiting. The indirect consequences (e.g. the inability to reason by *reductio ad absurdum*) are more difficult to assess. It is likely that fully formal proofs will only come about with the availability of usable Theorem Proving Assistants (see, for example, [JL88]). In this case, the unfamiliarity of a particular set of proof rules is likely to matter much less than other factors in the design of the User Interface.

Those approaches which avoid the need for a non-classical logic either bring their own surprises into proofs or force the separation of validity proofs from new ones for definedness. It has been a major goal in the development of LPF itself, and ways of using it in program development, that the normal user (as distinct from someone proving meta-results) should not have to reason about undefined values nor use an additional notion of strong equality.

One last objection which is levelled at LPF (mentioned for example in [Bli88]) is that it is 'unimplementable'. This point, in spite of its pragmatic tone, appears to be of minor practical significance. The property of *subp* which has been a *leitmotiv* of this paper arises out of the desire to prove that the recursive definition of *subp* satisfies a specification given by pre- and post-conditions. Those logical operators which do occur within, say, post-conditions are normally eliminated during design. (Consider, for example, a specification of *sort* which requires $\textit{is-ordered}(\textit{outl}) \wedge \textit{is-permutation}(\textit{inl}, \textit{outl})$: the conjunction is a specification trick and not intended to be part of the implementation; similar cases arise with the use of invariants). The rather small percentage of logical operators which might end up being translated into tests of conditionals in a programming language, should require only simple checking to avoid reliance on LPF's generous treatment of undefined (see **if**-I and **while**-I in [Jon90]).

Acknowledgements

The authors are grateful to Peter Lindsay for his comments on a draft of this paper. Thanks also go to members of IFIP's WG2.2 and WG2.3 working groups (especially Broy and Loeckx) with whom the evolving LPF has been discussed. One of the authors (CBJ) would like to thank Professors Burstall and Ito for arranging the Sendai workshop (where this material was first presented) and the Japanese attendees for their incredible hospitality. Thanks are also due to SERC who funded both the trip to Japan and the related research (by a Research Grant and a Senior Fellowship) and to the Wolfson Foundation for their research grant. The other author (JHC) would like to thank his employer for support.

References

[Abr84a] J-R. Abrial. The mathematical construction of a program and its application to the construction of mathematics. *Science of Computer Programming*, 4:45–86, 1984.

[Abr84b] J-R. Abrial. *Programming as a Mathematical Exercise*, pages 113–137. Prentice-Hall International, 1984.

[Abr87] J.R. Abrial. Private communication. Letter dated May 31st., 1987.

[Avr87] A. Avron. Simple consequence relations. Technical Report ECS-LFCS-87-30, LFCS, Department of Computer Science, University of Edinburgh, June 1987.

[Avr88] A. Avron. Foundations and proof theory of 3-valued logics. Technical Report ECS-LFCS-88-48, LFCS, Department of Computer Science, University of Edinburgh, April 1988.

[BCJ84] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[Bee86] M.J. Beeson. Proving programs and programming proofs. In B. Marcus et al., editors, *Logic, Methodology and Philosophy of Science*, volume VII, pages 51–82. Elsevier, 1986.

[Ben86] E. Bencivenga. Free logics. In D. Gabbay and F. Guenthuer, editors, *Handbook of Philosophical Logic, Volume III*, chapter 6. Reidel, 1986.

[BJ82] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice Hall International, 1982.

[Bla80] S.R. Blamey. *Partial Valued Logic.* PhD thesis, Oxford University, 1980.

[Bla86] S. Blamey. Partial logic. In D. Gabbay and F. Guenthuer, editors, *Handbook of Philosophical Logic, Volume III*, chapter 1. Reidel, 1986.

[Bli88] A. Blikle. Three-valued predicates for software specification and validation. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM—The Way Ahead*, pages 243–266. Springer-Verlag, 1988. Lecture Notes in Computer Science, Vol. 328.

[Bro87a] M. Broy. Equational specification of partial higher order algebras. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design — NATO ASI Series F: Computer and Systems Sciences, Vol. 36*, pages 185–241. Springer-Verlag, 1987.

[Bro87b] M. Broy. Predicative specifications for functional programs describing communicating networks. *Information Processing Letters*, 25:93–101, 1987.

[Bro88] M. Broy. Views of queues. *Science of Computer Programming*, 11:65–86, 1988.

[Che86] J.H. Cheng. *A Logic for Partial Functions*. PhD thesis, University of Manchester, 1986.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. In Series in Automatic Computation.

[Dri88] D. Driankov. Towards a many-valued logic of quantified belief. Technical Report 192, Linköping Studies in Science and Technology, 1988.

[Fef82] S. Feferman. Toward useful type-free theories, I. *Journal of Symbolic Logic*, pages 75–111, 1982.

[FJ87] L.M.G. Feijs and H.B.M. Jonkers. Formal definition of the design language COLD-K. Technical report, Philips Research Labs, The Netherlands, April 1987. Preliminary Edition.

[FJO+87] L.M.G. Feijs, H.B.M. Jonkers, J.H. Obbink, C.P.J. Koymans, G.R. R.de Lavaletter, and P.H. Rodenburg. A survey of the design language COLD. In *ESPRIT '86: Results and Achievements*, pages 631–644. Elsevier Science Publishers B.V., 1987.

[GFLC89] A. Gavilanes-Franco and F. Lucio-Carrasco. A first order logic for partial functions. Technical report, Universidad Complutense, Madrid, 1989.

[Gib88] P.F. Gibbins. VDM: Axiomatising its propositional logic. *BCS Computer Journal*, 31:510–516, 1988.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Gri81] D. Gries. *The Science of Computer Programming*. Springer-Verlag, 1981.

[Hay87] I. Hayes, editor. *Specification Case Studies*. Prentice-Hall International, 1987.

[HMT88] R. Harper, R. Milner, and M. Tofte. The definition of standard ML: Version 2. Technical report, Laboratory for Foundations of Computer Science, Edinburgh University, 1988.

[HN88] S. Hayashi and H. Nakano. *PX: A Computational Logic*. MIT Press, 1988.

[Hoo77] A. Hoogewijs. A calculus of partially defined predicates. Technical report, Rijksuniversiteit, Gent, 1977.

[Hoo87] A. Hoogewijs. Partial-predicate logic in computer science. *Acta Informatica*, 24:381–393, 1987.

[Jer88] C.A. Jervis. *A Theory of Program Correctness with Three Valued Logic*. PhD thesis, Leeds University, 1988.

[JL88] C.B. Jones and P.A. Lindsay. A support system for formal reasoning: Requirements and status. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88: VDM—The Way Ahead*, pages 139–152. Springer-Verlag, 1988. Lecture Notes in Computer Science, Vol. 328.

[Jon72] C.B. Jones. Formal development of correct algorithms: an example based on Earley's recogniser. In *SIGPLAN Notices, Volume 7 Number 1*, pages 150–169. ACM, January 1972.

[Jon80] C.B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 1986.

[Jon88] H.B.M. Jonkers. An introduction to COLD-K. Technical Report METEOR/t8/PRLE/8, Philips Research Labs, Eindhoven, July 1988.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[JS90] C.B. Jones and R.C.F. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrad, 1952.

[Kol76] G. Koletsos. Sequent calculus and partial logic. Master's thesis, Manchester University, 1976.

[Kol81] G. Koletsos. Notational and logical completeness in three-valued logic. *Bull. of the Greek Mathematical Society*, 22:121–141, 1981.

[Kri88] B. Krieg-Brückner. The PROSPECTRA methodology of program development. In Zalewski, editor, *IFIP/IFAC Working Conference on Hardware and Software for Real-Time Process Control*, page ?? ??, 1988.

[KTB88] B. Konikowska, A. Tarlecki, and A. Blikle. A three-valued logic for software specification and validation. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM—The Way Ahead*, pages 218–242. Springer-Verlag, 1988. Lecture Notes in Computer Science, Vol. 328.

[Lin87] P.A. Lindsay. A formal system with inclusion polymorphism. Technical Report IPSE Document 060/pal014/2.3, Manchester University, December 1987.

[Luk20] J. Łukasiewicz. O logice trójwartościowej. *Ruch Filozoficzny*, pages 169–171, 1920. Translated as (On three-valued logic) in Polish Logic 1920–39, S. McCall (ed.), Oxford U.P., 1967.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[McC67] J. McCarthy. A basis for a mathematical theory for computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland Publishing Company, 1967.

[MMN75] R. Milner, L. Morris, and M. Newey. A logic for computable functions with reflexive and polymorphic types. In *Proceedings of Conference on Proving and Improving Programs*, 1975.

[Owe85] O. Owe. An approach to program reasoning based on a first order logic for partial functions. Technical Report 89, Institute of Informatics, University of Oslo, February 1985.

[Plo84] G.D. Plotkin. Types and partial functions. Seminar at University of Manchester., 1984.

[Sat87] M. Sato. QJ. In *Fourth International Conference on Logic Programming*, 1987.

[Sch86] D.A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon, 1986.

[Sch87] P.H. Schmidt. Computational aspects of three-valued logic. Technical Report LILOG 26, IBM Deutschland, 1987.

[Sco79] D. S. Scott. Identity and existence in intuitionistic logic. In *Lecture Notes in Mathematics (Volume 735).* Springer-Verlag, 1979.

[Spi88] J.M. Spivey. *Understanding Z—A Specification Language and its Formal Semantics.* Cambridge Tracts in Computer Science 3. Cambridge University Press, 1988.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Ten87] R.D. Tennent. A note on undefined expression values in programming logic. *Information Processing Letters*, 24(5), March 1987.

[TZ88] J.V. Tucker and J.I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics.* CWI Monograph. North-Holland, 1988.

[Urq86] A. Urquhart. Many-valued logic. In D. Gabbay and F. Guenthuer, editors, *Handbook of Philosophical Logic, Volume III*, chapter 2. Reidel, 1986.

[vF66] B.C. van Fraasen. Singular terms, truth-value gaps and free logic. *J. Philosophy*, 63:481–495, 1966.

[Wan61] H. Wang. The calculus of partial predicates and its extension to set theory. *Math. Logic*, 7:283–288, 1961.