ELSEVIER

Science of Computer Programming 34 (1999) 141–158

Science of Computer Programming

www.elsevier.nl/locate/scico

E3: A logic for reasoning equationally in the presence of partiality

Joseph M. Morris*, Alexander Bunkenburg

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, UK

Communicated by C.B. Jones; received 5 May 1996; received in revised form 15 May 1998

Abstract

Partiality abounds in specifications and programs. We present a three-valued typed logic for reasoning equationally about programming in the presence of partial functions. The logic in essence is a combination of the equational logic **E** and typed LPF. Of course, there are already many logics in which some classical theorems acquire the status of neither-true–nor-false. What is distinctive here is that we preserve the equational reasoning style of **E**, as well as most of its main theorems. The principal losses among the theorems are the law of the excluded middle, the anti-symmetry of implication, a small complication in the trading law for existential quantification, and the requirement to show definedness when using instantiation. The main loss among proof methods is proof by mutual implication; we present some new proof strategies that make up for this loss. Some proofs are longer than in **E**, but the heuristics commonly used in the proof methodology of **E** remain valid. We present a Hilbert-style axiomatisation of the logic in which modus ponens and generalisation are the only inference rules. The axiomatisation is easily modified to yield a classical axiomatisation of **E** itself. We suggest that the logic may be readily extended to a many-valued logic, and that this will have its uses.

Keywords: Three-valued logic; Equational reasoning; Partial expressions

1. Introduction

Partiality abounds in programs and reasoning about programs. A simple example is the following statement about the behaviour of the head and tail functions on a sequence s:

$$s = \langle\rangle \vee (\text{head}\, s)^\frown(\text{tail}\, s) = s \qquad (*)$$

Such a statement must be given a clear meaning so that we can know its meaning even when s denotes the empty sequence; for example, does $(*)$ preserve its meaning

* Corresponding author. Fax: +44 141 330 4913; e-mail: jmm@dcs.glasgow.ac.uk

0167-6423/99/$ - see front matter
PII: S0167-6423(98)00019-7

when the two sides of the disjunction are commuted. There are many approaches to handling partiality; for a discussion of them see [6]. One approach is to allow formulae with non-denoting terms to be neither-true–nor-false, and to extend the logical connectives and quantifiers to cope with this third value. For example, when s denotes the empty sequence in $(*)$, we view the term $(\text{head}\, s)^\frown(\text{tail}\, s)$ as non-denoting, and the formula $(\text{head}\, s)^\frown(\text{tail}\, s) = s$ as neither-true–nor-false. We then extend the meaning of disjunction so that $P \vee Q$ is true when one of P and Q is true, regardless of the value of the other, and hence we can say that $(*)$ holds for all s. In short, this approach yields a three-valued logic for reasoning about partiality. We present such a logic.

Our logic is to a large extent a marriage of the equational logic **E** originating with Dijkstra and Feijen [7, 8], and typed LPF ("logic of partial functions") [11]. We call the union **E3**. **E** is essentially a treatment of traditional predicate logic based on the equivalence connective. Proof construction resembles familiar manipulation of expressions, in that the development proceeds mainly by substituting equals for equals. Although the strongest evidence for the efficacy of equational proofs comes from experience, there are technical reasons why this way of proving may be genuinely advantageous; see [13] for a discussion of this. LPF [4] is a logic developed for reasoning about the specification language VDM [10]. A typed version of LPF is developed in [11], formulated as a sequent calculus for proofs in natural deduction style. For arguments in favour of the LPF approach to handling partiality see [6]. With LPF we share the definitions of $\wedge$, $\vee$, $\neg$, and the quantifiers. From **E** we have taken the central role of $\equiv$ (which in **E3** becomes so-called "strong equality") and its proof techniques. The element that bridges the two theories is an implication connective which is different from that in either **E** or LPF, as we shall see.

We present the logic as a Hilbert-style axiomatisation, with modus ponens and generalisation as the only inference rules. This presentation holds good for **E** (one need only add the axiom that says that all terms are defined), and so as a by-product we obtain a classical presentation of **E** itself. It will turn out that **E3** retains the propositional part of **E** to a great extent, and that the predicate part survives virtually intact. The main propositional losses are the law of the excluded middle, associativity of $\equiv$, the absorption law that allows us to discard "$\neg P \vee$" in $P \wedge (\neg P \vee Q)$, and the anti-symmetry of implication. These still appear in **E3**, but with an obligation to show that some or all of the constituent formulae are well defined. Virtually, the only change in the predicate part of the logic is in the law of instantiation: we are now required to show that terms used to instantiate a formula are well defined. Of the proof methods of **E**, the main loss is proof of equivalence by mutual implication.

As well as presenting the logic formally, we give a model-theoretic semantics against which soundness and completeness can be established, we show that the equational reasoning of **E** is valid in **E3**, and we present some new proof strategies that make up for the loss of proof by mutual implication.

2. Design criteria

2.1. Monotonic operators

With the three boolean values being denoted by True, False, and $\bot$ (pronounced "bottom", representing neither-true–nor-false), we adopt the following truth-tables for negation, conjunction, and disjunction:

$\neg$	
True	False
False	True
$\bot$	$\bot$

$\wedge$	True	False	$\bot$
True	True	False	$\bot$
False	False	False	False
$\bot$	$\bot$	False	$\bot$

$\vee$	True	False	$\bot$
True	True	True	True
False	True	False	$\bot$
$\bot$	True	$\bot$	$\bot$

These connectives have a long pedigree, traced in [3]. They are monotonic in all arguments with respect to the partial ordering $\sqsubseteq$ that places $\bot$ below True and False. We also introduce strict equality which we denote by $=$ (operations are said to be "strict" if they yield $\bot$ whenever $\bot$ is an argument). Strict equality is also known as "weak equality".

2.2. Non-monotonic operators

We include strong equality $\equiv$, which differs from weak equality in that $\bot \equiv X$ is true when X is replaced by $\bot$, and is otherwise false. It is convenient to have a unary operator Δ such that ΔP is True whenever P is well defined, and otherwise False. ΔP is definable in terms of existing operators, for example, as $(P \equiv \text{True}) \equiv P$. Avron [1] investigates various three-valued implications, of which the most attractive for our purposes equates $P \Rightarrow Q$ with $(P \not\equiv \text{True}) \vee Q$ (equivalently, $\neg P \vee \neg \Delta P \vee Q$). Among the nice properties of this implication is that it preserves the deduction theorem. Avron [2] attributes the earliest use of this implication to Monteiro [12]. This implication is not antisymmetric, which means that we cannot prove equivalence by mutual implication (unless we show that both arguments are defined). Note that implication is monotonic with respect to $\sqsubseteq$ in its second argument but not its first.

2.3. Quantifiers

In two-valued logic, universal quantification is a generalised conjunction, and we retain that interpretation. In $(\forall x: T \cdot P)$, where T stands for a type, x is quantified over the *proper* elements of T (i.e. the non-$\bot$ elements of T). Analogously, we retain the view of existential quantification as a generalised disjunction. A consequence is the continued equivalence of $(\exists x: T \cdot P)$ and $\neg(\forall x: T \cdot \neg P)$.

There is some subtlety in these definitions. Taking $\exists$ as an example, $(\exists x: T \cdot P)$ may be either true (if P is true for some x), false (if P is false for every x), or neither-true–nor-false (if P is neither-true–nor-false for some x and false for any remaining x). This requires us to exercise some care in encoding informal statements as quantifications,

because it is easy to mis-translate a false/true statement into a false/true/neither-true–nor-false statement. Take as an example the encoding of the statement "function f (on the integers) has at least one zero" – a statement we mean to be interpreted as either true or false. We might naively encode this as $(\exists x: \mathbb{Z} \cdot fx = 0)$, but consider: (i) if f has a zero, $(\exists x: \mathbb{Z} \cdot fx = 0)$ is true, as we would like; (ii) if f has no zero and fx is defined for all x, then $(\exists x: \mathbb{Z} \cdot fx = 0)$ is false, as we would like; (iii) if f has no zero and fx is undefined for some x then $(\exists x: \mathbb{Z} \cdot fx = 0)$ is undefined, which is *not* as we would like. We must encode the statement as, for example, $(\exists x: \mathbb{Z} \cdot fx \equiv 0)$, or $(\exists x: \mathbb{Z} \cdot (fx = 0) \equiv \text{True})$, or $(\exists x: \mathbb{Z} \cdot fx = 0) \equiv \text{True}$.

Quantifications over subtypes are written as $(\forall x: T | R \cdot P)$ and $(\exists x: T | R \cdot P)$, respectively, where in each case R is called the "range". In two-valued logic, $(\forall x: T | R \cdot P)$ encodes "P is true for all x that satisfy R", and $(\exists x: T | R \cdot P)$ encodes "P is true for some x that satisfies R". For the three-valued case, we have to decide whether an x for which R is neither-true–nor-false is deemed "in" or "out" of the quantification, or even whether the existence of such an x might render the whole quantification neither-true–nor-false. In deciding this, the properties we first look to are the so-called "trading laws", which in the case of $\forall$ is

$$(\forall x: T | R \cdot P) \equiv (\forall x: T \cdot R \Rightarrow P).$$

Such laws are important in equational reasoning, because they provide a simple calculational mechanism for manipulating P in ways that exploit the limited range of x. To retain the trading laws we interpret $(\forall x: T | R \cdot P)$ as "P is true for all x for which R is (defined and) true". Crudely expressed, in ranges, neither-true–nor-false is not distinguished from false. If we want to preserve de Morgan's laws, and we do, we must define existential quantifications over subtypes by $(\exists x: T | R \cdot P) \equiv \neg(\forall x: T | R \cdot \neg P)$. This produces a trading law for $\exists$ that is less attractive than the two-valued version – $(\exists x: T | R \cdot P) \equiv (\exists x: T \cdot \neg(R \Rightarrow \neg P))$ – but we shall have to live with this. Observe that quantifications over subtypes are not monotonic with respect to $\sqsubseteq$ in their range argument.

2.4. Caveats

Non-monotonicity has ramifications for program refinement. We extract a program from a specification by refining its constituent parts, so that gradually the non-algorithmic constructs are replaced by algorithmic substitutes. In refining term E to term F (think of E and F as a specification or program or some kind of hybrid specification/program) we do not, in general, require that E and F be equivalent (in their context), but only that they be equivalent wherever E is defined. (Actually, the story is more complex for non-flat domains and/or in the presence of nondeterminacy, but we can ignore such complications for the moment.) We write $E \sqsubseteq F$ when it is safe to replace E with F in this sense (this re-use of the symbol $\sqsubseteq$ is safe, because the two uses of $\sqsubseteq$ on the booleans coincide). For example, $x \div x \sqsubseteq (2 * x + 1) \div (2 * x + 1)$ holds for any integer, but not $x \div x \equiv (2 * x + 1) \div (2 * x + 1)$ because the right-hand

side is always defined, even when x is 0. Program refinement depends fundamentally on substitution being monotonic with respect to $\sqsubseteq$, i.e. if $E \sqsubseteq F$ then $G \sqsubseteq G'$ where G' is obtained from G by replacing one or more occurrences of E with F – this is the essential property that enables us to program by stepwise refinement. However, $E \sqsubseteq F$ does not imply $G \sqsubseteq G'$ if E occurs in G as an operand of a non-monotonic operator such as $\Rightarrow$. In that case we can only replace E with an expression that is *equivalent* to it. For example, although $x \div x \sqsubseteq (2 * x + 1) \div (2 * x + 1)$ holds for any integer x, it is not the case that

$$x \div x = 1 \;\Rightarrow\; x \neq 0 \;\sqsubseteq\; (2 * x + 1) \div (2 * x + 1) = 1 \;\Rightarrow\; x \neq 0$$

because for $x = 0$ the left-hand side yields True, while the right-hand side yields False. The practical consequences are simply that we have to exercise care when we refine terms in non-monotonic positions, only substituting equals for equals.

We also have to exercise some extra care with recursively defined functions. Every recursively defined function f depends for its meaning on the existence of the "least fixed point" of a related function F, and for the least fixed point of F to exist, occurrences of f in the definition of F must be in monotonic positions (with respect to $\sqsubseteq$). This means that we cannot write a recursive function with a shape such as, say,

$$f \triangleq \textbf{fun}\, x{:}\, \mathbb{N} \cdot \ldots f(x-1) \Rightarrow \ldots$$

In practice, this is unlikely to be limiting in any substantive way, because it can be shown that the operators $\neg$ and $\vee$, with the constants True and $\bot$, are expressively complete for the class of all monotonic truth-valued functions (see [5]).

Implementations of the three-valued $\wedge$ and $\vee$ are expensive because they require the parallel evaluation of their arguments. We therefore have to replace them in a final program with so-called "conditional operators" that evaluate their arguments left to right. For example, the conditional conjunction $\vec{\wedge}$ behaves like strict conjunction except that False $\vec{\wedge} \bot$ yields False. The replacement of monotonic operators with conditional ones is, in practice, a minor clerical exercise. If $P \wedge Q$ is well defined, at most one of P and Q is undefined, say Q, and so $P \wedge Q$ can be re-written as $P \vec{\wedge} Q$, and similarly for $\vec{\vee}$. There are counter-examples to this simple strategy; for example, in the function

$$\begin{aligned} f \triangleq \textbf{fun}\, n{:}\, \mathbb{N},\ x{:}\, \{0,1\} \cdot\ & \textbf{if}\ n = 0\ \textbf{then}\ 0 \\ & \textbf{else if}\ x \div x = 1 \vee (1-x) \div (1-x) = 1 \\ & \qquad \textbf{then}\ f(n-1, 1-x) \\ & \textbf{fi} \end{aligned}$$

the expression $x \div x = 1 \vee (1-x) \div (1-x) = 1$ is always well defined and true but it cannot be replaced either by $x \div x = 1 \vec{\vee} (1-x) \div (1-x) = 1$ or $(1-x) \div (1-x) = 1 \vec{\vee} x \div x = 1$. Such examples are probably just curiosities, and do not appear to arise in practice.

2.5. Types

It is traditional in logic to have terms of only two types: those which denote boolean entities – these are usually called "formulae" – and those which denote objects in another unspecified type. Our treatment will admit of an arbitrary number of types, one of which will be the booleans, and we will regard "formula" as a synonym for an expression of type boolean. Note that we do not present a type system as such, but merely admit a family of type symbols which denote unknown types; this is the same approach as adopted in [11].

3. The logic

3.1. Syntax

We assume a set of type symbols; the letters T and U will stand for arbitrary-type symbols. Each term is associated with a type. The letters E, F, G, t, and u stand for terms in general. For each type we are given a supply of "variable symbols" (in programming we are free to introduce variables as we go along, and to choose their names, but that is a minor syntactic issue that is not a concern at the current level of discussion.) We use the letters x, y, and z to stand for variable symbols. Variable symbols are terms, and as such have the type with which they are initially associated. For each type we are given a supply of atomic terms of the type, called "constants". We are also given a supply of "operator symbols"; compound terms are made from more elementary terms using the operator symbols. An operator symbol combines a sequence of terms, each of a type determined by the operator; the resulting combination is a term of a type fixed by the operator called the "result type". It follows that the type of each term is statically decidable. The operators $\equiv$, $\not\equiv$, $=$ and $\neq$ are exceptions in that they accept arguments of any type, but the result type is fixed in each case (and of course is boolean).

We are given one type symbol $\mathbb{B}$ (pronounced "bool"). Terms of type $\mathbb{B}$ are called "formulae"; we let P, Q, R, S, V, W, X, Y, Z stand for formulae. True and False are constants of type $\mathbb{B}$. The "boolean operators" are $\wedge$, $\vee$, $\Rightarrow$, Δ, $\neg$, $\equiv$, $\not\equiv$, $=$, $\neq$, and the quantifiers $\forall$ and $\exists$, with a syntax as explained in the preceding section. Whenever we write a quantification $(\forall x: T \cdot P)$, etc. we require that x be of type T, but we will not explicitly say so on each occasion.

Brackets may be omitted using the following operator precedence (highest first): (i) Δ (ii) $\neg$ (iii) $\wedge$ and $\vee$ (iv) $\Rightarrow$ (v) $\equiv$ and $\not\equiv$ (vi) $=$ and $\neq$. It turns out that $\wedge$ and $\vee$ are associative, and we will use this fact from the outset to omit brackets. $\neg$ and Δ bracket to the right.

We denote by $E[x := t]$ the term got by substituting each free occurrence of x in E with t, where x and t are of the same type. This is the usual substitution mechanism whereby bound variables in E are renamed as necessary to avoid free variables in

t becoming bound as a result of the substitution. Whenever we write a substitution expression $E[x := t]$ we will assume that the requirement that x and t be of the same type is understood, without explicitly mentioning it. Substitution binds tightest of all.

3.2. Model theory

Terms and type symbols are interpreted with respect to a set containing at least the values True, False, and $\perp$; such a set is called the "domain of interpretation", or simply the "domain". (Note that we are re-using the symbols True, False, and $\perp$ to denote values in the domain; context should resolve any possible ambiguity). Each type symbol T is interpreted as a subset of the domain containing $\perp$ and at least one other element. Interpretations of constants (of type T, say) are constrained to be non-$\perp$ elements in the subset corresponding to T. Every operator symbol is associated with a "matching" total function on the domain. By "matching" we mean that the operator and the function take the same number of arguments, and that when the arguments of the function are type-correct then so is the result. The arguments of an interpreting function f are type-correct if they are interpretations of corresponding type-correct arguments of the operator. Terms are interpreted by induction on their structure.

The interpretation of $\mathbb{B}$ is {True, False, $\perp$}, with True in the logic being identified with True in the domain. The interpretations of the $\wedge$, $\vee$, $\Rightarrow$, and $\neg$ must be in agreement with the definitions of the preceding section.

The interpretation of $\equiv$ is the function on the domain that yields True or False according to whether its arguments are identical or not, and analogously for $\not\equiv$. The interpretations of $=$ is a function that behaves like $\equiv$ when its arguments are proper, and otherwise yields $\perp$. The interpretation of $\Delta\mathbf{E}$ is False or True according to whether the interpretation of E is $\perp$ or not.

A "valid interpretation" is a domain D and a mapping from types to subsets of D, and from terms without free variables or quantifiers to elements of D, that respects the requirements set out above. A valid interpretation extends naturally to terms with free variables by giving each variable symbol an interpretation. Interpretations of variables (of type T, say), are constrained to be non-$\perp$ values in the subset of the domain corresponding to T. A mapping from the set of variable symbols to their respective interpretations that meets this requirement is called a "state", and we refer to the interpretation of terms "with respect to" or "in" or "for" that state. It follows that for every valid interpretation, every term without quantifiers but possibly with free variables has an interpretation for each state.

Finally, we extend interpretations to terms with quantifiers. For any valid interpretation, we say that two states are "x-equivalent" if they are similar except possibly for the assignments to variable x. We also say that a state "satisfies" ("dissatisfies") formula P iff P is interpreted as True (respectively, False) in that state. For any valid interpretation, $(\forall x: T | R \cdot P)$ is interpreted as True in a state iff P is satisfied by all x-equivalent states that satisfy R. $(\forall x: T | R \cdot P)$ is interpreted as False in a state

iff some x-equivalent state that satisfies R dissatisfies P. $(\forall x: T \cdot P)$ is interpreted as $(\forall x: T | \text{True} \cdot P)$; $(\exists x: T | R \cdot P)$ is interpreted as $\neg(\forall x: T | R \cdot \neg P)$; and $(\exists x: T \cdot P)$ is interpreted as $\neg(\forall x: T \cdot \neg P)$.

We want each formula to be classified as a theorem if and only if every valid interpretation of it is True in every state. A logic which satisfies the "if" part of the preceding statement is said to be "sound", and one that satisfies the "only if" part is said to be "complete". It is pretty routine to show soundness, by checking that each axiom given below is interpreted as True for every valid interpretation and every state, and that this is preserved by the inference rules. As regards completeness, we can prove that every theorem of LPF is also a theorem of **E3**. The essence of the proof is that every inference rule in LPF is a derived inference rule in **E3**, allowing for minor differences in the two languages. The proof is omitted for brevity.

3.3. Proof theory

The axioms are all substitution instances of the formulae of Fig. 1. The theorems are the smallest subset of the formulae such that (i) every axiom is a theorem; (ii) if P and $P \Rightarrow Q$ are theorems, then so is Q, and (iii) if P is a theorem, then so is $(\forall x: T \cdot P)$ where x is of type T. A "proof" of P, i.e. a demonstration that P is a theorem, consists of a sequence of formulae whose final member is P, and such that each member of the sequence is an axiom or follows from preceding formulae in the sequence by an application of the inference rules modus ponens, or "MP" for short, and "generalisation (over x)":

$$\text{MP } \frac{P, P \Rightarrow Q}{Q} \qquad \text{Generalisation } \frac{P}{(\forall x: T \cdot P)}$$

4. Reasoning in E3

Proofs in **E** look very different from the simple sequence of lines described above, and indeed proofs in **E** rarely make explicit use of modus ponens at all. The archetypical proof in **E** consists of a sequence of equivalent formulae beginning with the formula we are trying to prove, and ending with a known theorem. Each formula (except the first) is derived from its predecessor P by replacing P or a sub-term in P with an equivalent expression. We now show that this technique, and variations on it, are valid in both **E** and **E3**.

4.1. Equational reasoning

Equational reasoning proceeds using the following derived inference rules:

$$\text{Equanimity } \frac{P, P \equiv Q}{Q} \qquad \text{Leibniz } \frac{E \equiv F, P[x := E]}{P[x := F]}$$

Equivalence

≡-reflexivity:	$E \equiv E$
≡-symmetry:	$(E \equiv F) \equiv (F \equiv E)$
≡-truth:	$((E \equiv F) \equiv \text{True}) \equiv (E \equiv F)$

Negation

exchange:	$(\neg P \equiv Q) \equiv (\neg Q \equiv P)$
≢-definition:	$(E \not\equiv F) \equiv \neg(E \equiv F)$
False-definition:	$\text{False} \equiv \neg\text{True}$

Disjunction

∨-symmetry:	$P \vee Q \equiv Q \vee P$
∨-associativity:	$P \vee (Q \vee R) \equiv (P \vee Q) \vee R$
∨-idempotency:	$P \vee P \equiv P$
∨-zero:	$P \vee \text{True} \equiv \text{True}$
∨-truth:	$((P \vee Q) \equiv \text{True}) \equiv (P \equiv \text{True}) \vee (Q \equiv \text{True})$

Conjunction

∧-definition:	$P \wedge Q \equiv \neg(\neg P \vee \neg Q)$
∧/∨:	$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
consistency:	$(P \wedge Q \equiv P) \equiv (P \vee Q \equiv Q)$
∧-truth:	$(P \wedge Q \equiv \text{True}) \equiv (P \equiv \text{True}) \wedge (Q \equiv \text{True})$

Implication

⇒-definition:	$P \Rightarrow Q \equiv (P \not\equiv \text{True}) \vee Q$
⇒/≡:	$P \Rightarrow (Q \equiv R) \equiv ((P \Rightarrow Q) \equiv (P \Rightarrow R))$
≡-weakening:	$(P \equiv Q) \Rightarrow (P \Rightarrow Q)$
Leibniz:	$(E \equiv F) \Rightarrow (G[x:=E] \equiv G[x:=F])$

Boolean definedness

Δ-definition:	$\Delta P \equiv ((P \equiv \text{True}) \equiv P)$
one-⊥:	$\Delta P \vee \Delta Q \vee (P \equiv Q)$

Universal quantification

∀/∧:	$(\forall x{:}T \bullet P \wedge Q) \equiv (\forall x{:}T \bullet P) \wedge (\forall x{:}T \bullet Q)$
∨/∀:	$(\forall x{:}T \bullet P \vee Q) \equiv P \vee (\forall x{:}T \bullet Q)$ if x does not occur free in P.
∀/≡:	$(\forall x{:}T \bullet P \equiv Q) \Rightarrow ((\forall x{:}T \bullet P) \equiv (\forall x{:}T \bullet Q))$
∀-truth:	$((\forall x{:}T \bullet P) \equiv \text{True}) \equiv (\forall x{:}T \bullet P \equiv \text{True})$
interchange:	$(\forall x{:}T \bullet (\forall y{:}U \bullet P)) \equiv (\forall y{:}U \bullet (\forall x{:}T \bullet P))$
renaming:	$(\forall x{:}T \bullet P) \equiv (\forall y{:}T \bullet P[x:=y])$ where y is fresh
trading:	$(\forall x{:}T \mid R \bullet P) \equiv (\forall x{:}T \bullet R \Rightarrow P)$

Existential quantification

∃-definition:	$(\exists x{:}T \bullet P) \equiv \neg(\forall x{:}T \bullet \neg P)$
∃|-definition:	$(\exists x{:}T \mid R \bullet P) \equiv \neg(\forall x{:}T \mid R \bullet \neg P)$
∃-truth:	$((\exists x{:}T \bullet P) \equiv \text{True}) \equiv (\exists x{:}T \bullet P \equiv \text{True})$

Term definedness

instantiation:	$(\forall x{:}T \bullet P) \wedge \Delta t \Rightarrow P[x:=t]$
variables defined:	Δx

Weak equality

=-definedness:	$\Delta(E = F) \equiv \Delta E \wedge \Delta F$
=-definition:	$\Delta(E = F) \equiv ((E = F) \equiv (E \equiv F))$
≠-definition:	$E \neq F \equiv \neg(E = F)$

Fig. 1. Axioms of **E3**.

$$\text{Transitivity}\ \frac{E \equiv F, F \equiv G}{E \equiv G} \qquad \text{True-introduction}\ \frac{P}{P \equiv \text{True}}$$

$$\text{False-introduction}\ \frac{\neg P}{P \equiv \text{False}}$$

(The symmetry of $\equiv$ gives rise to trivial variations on each of the above.) In **E** and **E3**, a proof that P is a theorem is typically laid out as follows:

$$\begin{array}{ll} & P \\ \equiv & \text{"justification 1"} \\ & Q \\ \equiv & \text{"justification 2"} \\ & R \qquad\qquad - \text{ Theorem "}T\text{"} \end{array}$$

This is short hand for:

(i) $P \equiv Q$ – "justification 1"
(ii) $Q \equiv R$ – "justification 2"
(iii) $P \equiv R$ – (i), (ii), Transitivity
(iv) R – Theorem "T"
(v) P – (iii), (iv), Equanimity

Each "justification" is a short text explaining why the associated equivalence $X \equiv Y$ (for any X and Y) is valid. It takes one of the following forms. Firstly, it may be a reference to where $X \equiv Y$ has been established as a theorem. Secondly, if X and Y are similar in structure except that X has a subexpression E where Y has subexpression F, then $X \equiv Y$ can be cast in the form $Z[x := E] \equiv Z[x := F]$ where x stands for a fresh variable. In that case, the justification consists of a reference to where $E \equiv F$ has been established as a theorem. The truth of $Z[x := E] \equiv Z[x := F]$ follows from axiom Leibniz and an application of MP. Alternatively, if $X \equiv Y$ can be cast in the form $Z[x := P] \equiv Z[x := \text{True}]$, its justification consists of a reference to where P has been established as a theorem. The conclusion follows as before, with an additional appeal to True-introduction to infer $P \equiv \text{True}$. Similarly, if $X \equiv Y$ can be cast in the form $Z[x := P] \equiv Z[x := \text{False}]$, the justification consists of a reference to where $\neg P$ has been established as a theorem. The conclusion follows as above, except we appeal to False-introduction. Occasionally, a reference to where $E \equiv F$ has been established is packaged as a pair of references, one to where $P \Rightarrow (E \equiv F)$ has been established for some P, and another to where P has been established; $E \equiv F$ follows from an application of MP.

The foregoing describes a proof carried out in two steps; the generalisation to any number of steps is obvious. Comparing the proof presentation above with its first expansion (i.e. (i)–(v)), we see that step (iii) establishes $P \equiv R$. It follows that we may prove $P \equiv R$ using just this proof presentation, but without a justification of R in the final line. In short, we may prove an equivalence by reducing one side to the other. This is much used in **E** and continues to be valid in **E3**.

4.2. Reasoning with implication

A proof of $P \Rightarrow R$ may proceed just as described in the preceding section, or it may be laid out in a style typified by

$$
\begin{array}{ll}
 & P \\
\Rightarrow & \text{"justification 1"} \\
 & Q \\
\Rightarrow & \text{"justification 2"} \\
 & R
\end{array}
$$

This abbreviates the proof steps $P \Rightarrow Q$ followed by $Q \Rightarrow R$, the conclusion, $P \Rightarrow R$, following from an application of the derived inference rule

$$\text{Implication Transitivity } \frac{P \Rightarrow Q, Q \Rightarrow R}{P \Rightarrow R}$$

See [8] for further details and minor variations. Just as with proofs of equivalence, this style may be viewed as a macro-language for traditional derivations that use only the originally given inference rules.

4.3. Other inference rules

E occasionally employs proof by mutual implication and $\wedge$-introduction:

$$\text{Mutual Implication } \frac{P \Rightarrow Q, Q \Rightarrow P}{P \equiv Q} \qquad \wedge\text{-introduction } \frac{P, Q}{P \wedge Q}$$

Mutual Implication is not valid in **E3**, and its absence occasionally increases the proof burden; we shall introduce other inference rules later to make up for this loss. $\wedge$-introduction continues to hold in **E3**.

5. Propositional logic

5.1. Theorems

"Propositional logic" is that part of the logic in which the only type is $\mathbb{B}$, the axioms are those of equivalence, negation, disjunction, conjunction, implication, and boolean definedness, and MP is the only inference rule. The propositional part of **E** is retained in large measure in **E3**.

- Negation is an involution.
- Conjunction and disjunction are associative, symmetric, and idempotent. They retain their distribution properties with respect to one another. True and False behave as their zeros and units. De Morgan's laws hold.

- Implication is reflexive and transitive. False implies everything, and everything implies True. Implication distributes over $\wedge$, $\vee$, and $\Rightarrow$ on the right.
- The weakening and strengthening laws hold (i.e. $P \Rightarrow P \vee Q$, and $P \wedge Q \Rightarrow P$), as does the shunting law (i.e. $P \wedge Q \Rightarrow R$ is equivalent to $P \Rightarrow (Q \Rightarrow R)$).
- Almost all the monotonicity properties of operators with respect to $\Rightarrow$ continue to hold (for example, $(P \Rightarrow Q) \Rightarrow (P \vee R \Rightarrow Q \vee R)$).

The following theorems of **E** are *not* theorems of **E3**:

True is a unit of $\equiv$	$(P \equiv \text{True}) \equiv P$
Excluded middle	$P \vee \neg P$
$\vee$ distributes over $\equiv$	$(P \vee (Q \equiv R)) \equiv ((P \vee Q) \equiv (P \vee R))$
absorption	$P \wedge (\neg P \vee Q) \equiv P \wedge Q$
$\Rightarrow / \equiv / \wedge$	$(P \Rightarrow (Q \equiv R)) \equiv ((P \wedge Q) \equiv (P \wedge R))$.

In each case, they become theorems if they are prefixed with "$\Delta P \Rightarrow$". Some of them can be weakened in other ways. For example, $P \vee \neg P \not\equiv \text{False}$ is a theorem. The following weakened versions of the absorption law holds:

absorption'	$(P \wedge (\neg P \vee Q) \equiv \text{True}) \equiv (P \wedge Q \equiv \text{True})$.

Neither do the following theorems of **E** hold in **E3**, unless we show that all arguments are well defined:

$\equiv$ is associative, and $\Rightarrow$ is anti-symmetric

golden rule	$(P \wedge Q \equiv P) \equiv (P \vee Q \equiv Q)$
$\wedge$ from $\Rightarrow$	$P \Rightarrow Q \equiv (P \wedge Q \equiv P)$
contrapositive law	$(P \Rightarrow Q) \equiv (\neg Q \Rightarrow \neg P)$
$\neg$-import/export	$\neg(P \equiv Q) \equiv (\neg P \equiv Q)$.

The following substitution laws of **E** continue to hold in **E3**:

$\wedge$-substitution:	$(E \equiv F) \wedge P[x := E] \equiv (E \equiv F) \wedge P[x := F]$
$\Rightarrow$-substitution:	$(E \equiv F) \Rightarrow P[x := E] \equiv (E \equiv F) \Rightarrow P[x := F]$.

The operator Δ is not interesting in two-valued logic. In **E3** it enjoys the properties:

$\Delta(E \equiv F), \Delta(E \not\equiv F), \Delta\text{True}, \Delta\text{False}, \Delta\Delta P$

$(P \equiv \text{True}) \Rightarrow P \wedge \Delta P$

$P \vee \neg P \vee \neg \Delta P$

$\Delta P \Rightarrow P \vee \neg P$ and $P \vee \neg P \Rightarrow \Delta P$.

Although ΔP and $P \vee \neg P$ imply one another, they are not equivalent – to conclude equivalence from mutual implication we would have to show that both terms are well-defined, and we cannot prove $\Delta(P \vee \neg P)$.

5.2. Few or many $\perp$s

All of the theorems up to this point are provable without appealing to axiom one-$\perp$, but this changes when we come to what would seem to be a reasonable encoding of Δ:

$$\Delta P \equiv (P \not\equiv \neg P).$$

It turns out that without one-$\perp$ the axiomatisation of the propositional part of the logic does not exclude the possibility of many boolean bottoms, and the above equivalence does not hold in all such worlds. For example, the axiomatisation admits of two improper boolean values $\perp_0$ and $\perp_1$, the four booleans constituting a chain lattice with elements False, $\perp_0$, $\perp_1$, and True in that order, $\wedge$ and $\vee$ being the lattice operations, and $\perp_0$ and $\perp_1$ being one another's complement. Axiom one-$\perp$ excludes such possibilities. We remark that the fact that we can pass to an n-valued logic, $n>3$, with hardly any loss over the 3-valued version is interesting, because there are some settings in which it is desirable to have more than one improper value, in which case all of the theorems except the preceding one (which is not very important) are valid.

We have not included $\perp$ as a designated value in the logic; we can do so if we wish by adding the axiom

$\perp$-definition: $\quad \neg\Delta\perp.$

We observe finally, that we can reduce **E3** to **E** by adding the axiom

$$\Delta E.$$

This produces a classical two-valued logic by excluding the possibility of improper values in any type. Alternatively, we can add the axiom of the excluded middle which guarantees that the booleans are 2-valued while leaving open the possibility of assigning no meaning to integer expressions such as $5 \div 0$.

5.3. Proof strategies

Proofs of propositional theorems are in the main no more difficult in **E3** than in **E**. Surprisingly, the loss of the law of the excluded middle was not a big impediment to finding proofs, because an alternative route was usually evident. Perhaps the main loss is that of proof of equivalence by mutual implication: in **E3** we have to live with the weakened form

$$\text{Mutual Implication } \frac{P \Rightarrow Q, Q \Rightarrow P, \Delta P, \Delta Q}{P \equiv Q}$$

When we cannot guarantee the well-definedness of both sides of the equivalence, we can resort to the rule of Truth Cases:

$$\text{Truth Cases } \frac{(P \equiv \text{True}) \equiv (Q \equiv \text{True}), (P \equiv \text{False}) \equiv (Q \equiv \text{False})}{P \equiv Q}$$

The justification of Truth Cases relies on axiom one-$\bot$. It is a brute force method, to be applied as a last resort.

The following derived inference rules are useful in three-valued logic:

$$\text{True-elimination}\ \frac{P \equiv \text{True}}{P} \qquad \Rightarrow\text{-Truth}\ \frac{(P \equiv \text{True}) \Rightarrow (Q \equiv \text{True})}{P \Rightarrow Q}$$

(and, of course, they are also valid in two-valued logic.) Their primary value is that they allow us to work with well-defined formulae such as $P \equiv \text{True}$ instead of P.

6. Predicate logic

6.1. Derived inference rules

In proofs involving quantifiers and variables, **E** relies on the following set of inference rules:

$$\forall\text{-Leibniz 1}\quad \frac{t \equiv u}{(\forall x: T \mid R[y := t] \cdot P[y := t]) \equiv (\forall x: T \mid R[y := u] \cdot P[y := u])}$$

$$\forall\text{-Leibniz 2}\quad \frac{R \Rightarrow (t \equiv u)}{(\forall x: T \mid R \cdot P[y := t]) \equiv (\forall x: T \mid R \cdot P[y := u])}$$

We also have the rules $\exists$-Leibniz 1 and $\exists$-Leibniz 2 by replacing $\forall$ with $\exists$ in the above, respectively. In the preceding rules, it is not excluded that x and y are identical symbols.

$$\forall\text{-monotonicity 1}\quad \frac{R \wedge P \Rightarrow Q}{(\forall x: T \mid R \cdot P) \Rightarrow (\forall x: T \mid R \cdot Q)}$$

$$\forall\text{-monotonicity 2}\quad \frac{R \Rightarrow S}{(\forall x: T \mid S \cdot P) \Rightarrow (\forall x: T \mid R \cdot P)}$$

We also have the rules $\exists$-monotonicity 1 and $\exists$-monotonicity 2 by replacing $\forall$ with $\exists$ in the above, respectively.

$$\text{Instantiation}\quad \frac{(\forall x: T \cdot P)}{P}$$

All of these derived inference rules remain valid in **E3**.

The deduction theorem continues to hold in **E3**; standard proofs (see for example [9]) rely only on theorems which are valid in **E3**.

With the quantifiers, there are some additional ways of justifying steps in proof presentations. Briefly, when a proof step (i.e. a pair of lines in a proof, together with their connecting $\equiv$ or $\Rightarrow$) is an instance of the conclusion of one of the inference rules above, the justification will be the corresponding hypothesis (or a justification thereof). It follows that proofs in **E** or **E3** can be mechanically translated to proofs that employ MP and Generalisation as the only inference rules.

6.2. Theorems

Just about all the theorems of the predicate logic part of **E** hold in **E3**. The principle exceptions are those which arise from the axiom of instantiation, which must now be prefixed with "$\Delta t \Rightarrow$" where t stands for the instantiating term; for example:

$$\Delta t \Rightarrow (P[x := t] \;\Rightarrow\; (\exists x: T \cdot P)).$$

A second source of differences arises from the trading law for existential quantification, which in one form can be expressed as:

$$(\exists x: T | R \cdot P) \equiv (\exists x: T \cdot (R \equiv \mathrm{True}) \wedge P).$$

Weak equality behaves as in two-valued logic except that reflexivity has a definedness requirement, as have the one-point and shifting rules. For example, the one-point rule for $\forall$ is

$$(\forall x: T | x = E \cdot P) \equiv P[x := E] \vee \neg \Delta E \quad \text{where } x \text{ not free in } E$$

and the shifting rule for $\forall$ is

$$(\forall x: T | R \cdot P) \equiv (\forall x: T | R[x := E] \cdot P[x := E])$$

provided that as a function of x, E is surjective and total, i.e. $(\forall y: T \cdot (\exists x: T \cdot y = E))$ and $(\forall x: T \cdot (\exists y: T \cdot y = E))$ (or equivalently, $(\forall x: T \cdot \Delta E)$). This differs from **E** in the addition of the totality requirement.

The theorem $E = F \Rightarrow (E \equiv F)$ allows us to use weak equality in place of strong equality in the hypotheses of inference rules and the antecedents of implications. As a consequence, we can use $E = F$ as a justification in proof steps where previously we have required $E \equiv F$.

6.3. Proof strategies

Predicate logic proofs in **E3** are in the main pretty similar to proofs in **E**, although they tend to be a bit longer and on occasion more difficult. The main irritant in proofs in **E3** is the obligation to show definedness of a term prior to instantiation. We illustrate with an example. In [4, 6, 11], the theorem $(\forall i, j: \mathbb{Z} \cdot i \geqslant j \Rightarrow f(i,j) = i - j)$ is used to highlight differences among alternative logics, where

$$f \triangleq \textbf{fun}\ i, j:\ \mathbb{Z} \cdot \textbf{if}\ i = j\ \textbf{then}\ 0\ \textbf{else}\ f(i, j+1) + 1.$$

We give a proof in **E3** for comparison. It is obvious that we shall have to use induction on variable j, and so we re-write the demonstrandum as $(\forall i: \mathbb{Z} \cdot (\forall j: \mathbb{Z} \cdot i \geqslant j \Rightarrow f(i,j) = i - j))$ (nesting the induction variable inside is simpler, though not always adequate). Appealing to generalisation over i, we need only prove $(\forall j: \mathbb{Z} \cdot i \geqslant j \Rightarrow f(i,j) = i - j)$. We shall use obvious properties of the integers, indicating such use by the hint

"arithmetic". We shall also use simple properties of function application and **if**... **then**... **else**.... As far as the definedness of integer terms is concerned, we only need the fact that addition, subtraction, and the relational operators are strict, and that Δc holds for all integer constants c.

$$(\forall j: \mathbb{Z} \cdot i \geqslant j \Rightarrow f(i,j) = i - j)$$

$\equiv$ "shifting with $j := (i - j)$, noting that the surjectivity

$$- (\forall k: \mathbb{Z} \cdot (\exists j: \mathbb{Z} \cdot k = i - j)) \text{ – and totality}$$

$$- (\forall j: \mathbb{Z} \cdot \Delta(i - j)) \text{ – requirements are met"}$$

$$(\forall j: \mathbb{Z} \cdot i \geqslant i - j \Rightarrow f(i, i - j) = i - (i - j))$$

$\equiv$ "arithmetic"

$$(\forall j: \mathbb{Z} \cdot j \geqslant 0 \Rightarrow f(i, i - j) = j)$$

$\equiv$ "trading"

$$(\forall j: \mathbb{Z} \mid j \geqslant 0 \cdot f(i, i - j) = j)$$

$\equiv$ "$(\forall x: \mathbb{Z} \mid x \geqslant 0 \cdot P) \equiv (\forall x: \mathbb{N} \cdot P)$"

$$(\forall j: \mathbb{N} \cdot f(i, i - j) = j)$$

$\equiv$ "induction on $\mathbb{N}$"

$$f(i, i - 0) = 0 \wedge (\forall j: \mathbb{N} \cdot f(i, i - j) = j \Rightarrow f(i, i - (j + 1)) = j + 1)$$

$\equiv$ "arithmetic"

$$f(i,i) = 0 \wedge (\forall j: \mathbb{N} \cdot f(i, i - j) = j \Rightarrow f(i, i - (j + 1)) = j + 1)$$

By $\wedge$-introduction, we prove each conjunct separately. Firstly,

$$f(i,i) = 0$$

$\equiv$ "definition of f"

$$(\textbf{fun } i,j: \mathbb{Z} \cdot \textbf{if } i = j \textbf{ then } 0 \textbf{ else } f(i, j + 1) + 1)(i,i) = 0$$

$\equiv$ "function application, Δi (substitution requires definedness)"

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

$\equiv$ "reflexivity of $=$, Δi"

$$(\textbf{if } \text{True} \textbf{ then } 0 \textbf{ else } f(i, i + 1) + 1) = 0$$

$\equiv$ "$(\textbf{if } \text{True} \textbf{ then } E \textbf{ else } F) \equiv E$"

$$0 = 0 \text{ – reflexivity of } =, \ \Delta 0$$

For the second conjunct $(\forall j: \mathbb{N} \cdot f(i, i-j)=j \Rightarrow f(i, i-(j+1))=j+1)$, we begin by appealing to generalisation:

$$f(i, i-j)=j \Rightarrow f(i, i-(j+1))=j+1$$

$\equiv$ "assume $f(i, i-j)=j$ (here we are using the deduction theorem)"

$$f(i, i-(j+1))=j+1$$

$\equiv$ "definition of f"

$$(\textbf{fun } i, j: \mathbb{Z} \cdot \textbf{if } i=j \textbf{ then } 0 \textbf{ else } f(i, j+1)+1)(i, i-(j+1))=j+1$$

$\equiv$ "function application, $\Delta i, \Delta(i-(j+1))$"

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

$\equiv$ "arithmetic"

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

$\equiv$ "$f(i, i-j)=j$ by assumption"

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

$\equiv$ "elementary property of **if**"

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

$\equiv$ "elementary property of **if** (using $\Delta(j+1=0)$)"

$$j+1=j+1 \text{ – reflexivity of } =, \Delta(j+1)$$

7. Concluding remarks

We have presented a logic for reasoning equationally about programming in the presence of partiality. It is a combination of **E** and LPF, and is designed to cover the full gamut from programs to specifications to reasoning about specifications. The logic is presented as a classical Hilbert-style axiomatisation in which modus ponens and generalisation are the only inference rules. Almost all the proof methods of **E** continue to be valid, as well as most of the main theorems. The principal changes in the body of theorems are the loss of the law of the excluded middle and anti-symmetry of implication, a small complication in the trading law for existential quantification, and the requirement to show definedness before using instantiation. With some exceptions, proofs tend to be but marginally more difficult than proofs in **E**. Much of the added difficulty stems from the unavailability of proof by mutual implication, but there are alternative proof strategies to overcome this, albeit at the price of longer proofs on occasion. The axiomatisation is easily modified to yield a classical axiomatisation of **E** itself. As an aside, we observe that most of the theorems continue to hold without

the assumption of a single improper value, and so the logic should readily admit of extension to a many-valued logic. This could be important in some contexts.

Acknowledgements

The axiomatisation of the propositional part of the logic is based on an earlier version in which Sharon Flynn also took part. The presentation has been much improved by the suggestions of two referees. Cliff Jones suggested the strategy for proving completeness.

References

[1] A. Avron, Foundations and proof theory of 3-valued logics, LFCS Report Series ECS-LFCS-88-48, Laboratory for the Foundations of Computer Science, Edinburgh University, 1988.
[2] A. Avron, Natural 3-valued logics – characterisation and proof theory, J. Symbolic Logic 56 (1991) 276–294.
[3] A. Blikle, Three-valued predicates for software specification and development, in: R. Bloomfield et al. (Eds.), VDM – The Way Ahead, Lecture Notes in Computer Science, Vol. 328, Springer, Berlin, 1988, pp. 243–266.
[4] H. Barringer, J.H. Cheng, C.B. Jones, A logic covering undefinedness in program proofs, Acta Informatica 21 (1984) 251–269.
[5] J.H. Cheng, A logic of partial functions, Ph.D. Thesis, University of Manchester, Dept. of Computer Science Technical Report UMCS-86-7-1, 1986.
[6] J.H. Cheng, C.B. Jones, On the usability of logics which handle partial functions, in: C. Morgan, J.C.P. Woodcock (Eds.), 3rd Refinement Workshop, Workshops in Computing, Springer, London, 1991, pp. 51–69.
[7] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Springer, New York, 1990.
[8] D. Gries, F.B. Schneider, A Logical Approach to Discrete Math., Springer, New York, 1993.
[9] A.G. Hamilton, Logic for Mathematicians, Cambridge University Press, Cambridge, 1988.
[10] C.B. Jones, Systematic Software Development Using VDM, 2nd ed., Prentice-Hall International, New York, 1990.
[11] C.B. Jones, C.A. Middelburg, A typed logic of partial functions reconstructed classically, Acta Informatica 31 (1994) 399–430.
[12] A. Monteiro, Construction des algèbres de Lukasiewicz trivalentes dans les algèbres de Boole monadiques I, Mathematica Japonica 12 (1967) 1–23.
[13] L.A. Wallen, On form, formalism, and equivalence, in: W.H.J. Feijen et al. (Eds.), Beauty is our Business – A Birthday Salute to Edsger W. Dijkstra, Springer, New York, 1990.