Acta Informatica 37, 541–561 (2001)

A theory of bunches

Joseph M. Morris[1], **Alexander Bunkenburg**[2,⋆]

[1] School of Computer Applications, Dublin City University, Dublin, Ireland (e-mail: jmorris@compapp.dcu.ie)

[2] Department of Computing Science, University of Glasgow, Glasgow G12 8RZ, United Kingdom

Received: 23 December 1998 / 2 November 1999

Abstract. A *bunch* is a simple data structure, similar in many respects to a set. However, bunches differ from sets in that the data is not packaged up or encapsulated, and in particular in that a bunch consisting of one element is the same as that element. Bunches are attractive for handling nondeterminacy and underspecification, by which is meant that for any particular input to the program or specification, the associated output is not fully determined. The acceptable outputs for any given input can be described by a bunch. This approach nicely generalises traditional single-output programs and specifications. We present a formal theory of bunches. It includes an axiomatisations of boolean and function types whose behaviour is well-known to be complicated by the presence of nondeterminacy. The axiomatisation of the booleans preserves most of the laws of classical predicate calculus. The axiomatisation of functions accommodates higher-order functions in all their generality, while avoiding the dangers of inconsistency when functions and nondeterminacy intermix. Our theory is presented as a Hilbert-style system of axioms and inference rules for a small specification language. We prove consistency.

1 Introduction

A *bunch* is a simple data structure, similar in many respects to a set. As with sets, we can ask whether an element is *in* the data structure or not. However, bunches differ from sets in that the data is not packaged up or encapsulated, and in particular in that a bunch consisting of one element is the same

⋆ Supported by the Leverhulme Trust

as that element. While a set is many entities packaged into *one* entity, a bunch is a many seen as a many. A bunch doesn't *contain* its elements, it *consists* of its elements. Bunches are attractive for handling nondeterminacy and underspecification, by which is meant that for any particular input to a program or specification, the associated output is not fully determined. A simple example is a program which returns an e-mail address corresponding to a supplied user name, without any commitment as to which of the user's e-mail addresses is returned in the case that he/she has more than one. We can describe such nondeterminacy in programs by specifying the output to be any of the values constituting a certain bunch. This approach nicely generalises traditional single-output programs and specifications, because there is no semantic or syntactic distinction between a single value and the bunch consisting of just that value.

We present a formal theory of bunches. Actually, most of the technical difficulties come from the intermixing of bunches and the data types of programming and specification languages, most fundamentally booleans and function types. (This is nothing to do with bunches per se, but rather is a consequence of the nondeterminacy which bunches provide.) We therefore axiomatise a small specification language which includes booleans and functions. Our theory is presented as a classical Hilbert-style system of axioms and inference rules, the first time this has been done for bunch theory as far as we are aware.

Our treatment of booleans and functions departs significantly from previous approaches [NH93, Heh93]. A boolean term can be identified with any of four bunches: as well as the familiar *true* and *false* we have two *strange* values, namely the empty boolean bunch and the bunch consisting of *true* and *false*. It is important that the boolean operators should retain as many as possible of their algebraic properties when they are extended to accommodate the strange boolean values. We propose just such an extension and argue that it has better algebraic properties than the booleans of previous approaches. We also propose an axiomatisation of functions that, in our view, fits more comfortably with bunch theory. It has the advantage of accommodating higher-order functions in all their generality, while avoiding the well-known dangers of inconsistency when functions and nondeterminacy intermix (see for example [Mee86]). Finally, we present a model by which we can show that bunch theory (or at least our version of it) is consistent.

Bunches were invented by Hehner ([Heh81]) as a mechanism for incorporating non-determinism and underspecification into formal programming. Although these topics can be approached via set theory, set theory is too powerful and the resulting calculus is too unwieldy. Its great disadvantage is that in place of traditional single-valued expressions, we have singleton sets with all the accompanying notational overheads of packaging and unpackaging.

The theory that is ‘just right’ is bunch theory because there is no distinction between a value and the singleton bunch consisting of that value.

The short paper [Heh81] first introduces bunches and argues their usefulness in computer science. The textbook [Heh84] uses bunches as the basis of specification and programming; it is superseded by the more formal [Heh93]. The workshop paper [NH93] examines in detail the important combination of bunches and functions. For an example of the practical application of bunch theory, see the textbook [Lee93] which uses bunches for a unified presentation of parsing algorithms. It exploits the fact that since a singleton bunch *is* its only element, there is no notational gap between deterministic and non-deterministic parsing algorithms.

The present paper can be understood without reference to the literature on bunch theory, but the reader who wants further motivation and examples should consult the above references.

2 Bunches

The fact that bunches, unlike sets, are not packaged up or encapsulated is reflected in their notation: there are no bunch-brackets, but rather the elements of a bunch are simply listed, separated by commas. For example $1, 2, 3$ is the bunch consisting of 1 and 2 and 3. The comma is called “bunch union”. (Since “,” is also a punctuation mark of English, we will construct our sentences so that punctuation-commas are easily distinguishable from bunch-commas, although this will sometimes lead to seemingly clumsy phrasing.) Unlike [NH93], our bunches are strongly typed; for example, we do not admit the bunch $1,$ *true*. We adopt strong typing partly because it is standard in just about all commonly used specification and programming languages, and partly because it facilitates the establishment of consistency. The operational idea underlying bunches is that the evaluation of a bunch yields one of its elements. We do not know which element is delivered, and there is no guarantee that several evaluations of the same bunch will yield the same element. For example, an application of the function $\lambda x{:}\mathbf{N} \cdot x, x+1$ to 7 yields the outcome 7 or 8, and it may well yield 7 when applied in the morning, and 8 when applied in the afternoon.

The *prescription* $\S x{:}T \mid P$ describes a bunch consisting of those elements x of type T that satisfy the predicate P. For example $\S n{:}\mathbf{N} \mid n \leq 3$ is equivalent to $0, 1, 2, 3$. For a simple example of its role in providing underspecification, consider the specification $\lambda f{:}\mathbf{N}{\to}\mathbf{N} \cdot (\S n{:}\mathbf{N} \mid f\ n \equiv 0)$. This takes a function on the naturals and returns one of its zeros. When applied to $\lambda x{:}\mathbf{N} \cdot x^2 - 3x + 2$ for example, it yields either 1 or 2. If the argument function has no zero, then the outcome is $null_{\mathbf{N}}$, where $null_T$ denotes the empty bunch of type T, i.e. the bunch consisting of zero elements. The bunch all_T

consists of all elements of type T. We may sometimes omit the subscripts in $null_T$ and all_T when they can be inferred from context, or when their value is not significant.

A bunch E is said to be *elementary* or an *element* and we assert ΔE, iff conceptually it consists of a single entity. For example, both *true* and 1 are elementary. (We use the symbol Δ for compatibility with [MB99a, MB98]; in [Heh93] it is used for an entirely different purpose.) All bound variables range over *elements* only. For example $\forall n{:}\mathbf{N} \cdot P$ implies that $P[n\backslash 0]$ and $P[n\backslash 12]$ and $P[n\backslash 35]$ hold, but not $P[n\backslash null_{\mathbf{N}}]$ or $P[n\backslash 0,1]$ or $P[n\backslash all_{\mathbf{N}}]$.

We say "E is a *subbunch* of F" and assert $E : F$ iff every element of E is also an element of F. For example, $2,3 : 1,2,3,4$ holds. The empty bunch $null_T$ is a subbunch of every bunch of type T, while every bunch of type T is a subbunch of all_T. As one might expect by analogy with subsetting, the subbunch relation : is a partial order whose antisymmetric closure we denote by the symbol $\equiv$. A type T is said to be *flat* iff for all *elements* of the type, the subbunch relation is identical with equivalence, i.e. $\forall x, y{:}T \cdot x : y \equiv (x \equiv y)$. For example, the natural numbers and the booleans are both flat, but not function types as we shall see later. We also write $E{:}T$ to assert that expression E has type T, and similarly for $E, F{:}T$ etc.; context will resolve any ambiguity between this use of : and its use to denote subbunching (and in the latter case, we surround : with a little extra space).

The bunch intersection $E'F$ consists of those elements that are common to both E and F. If a bunch E has finitely many elements, we say it is *finite* and assert $\mathbf{fin}\, E$. The *cardinality* of a finite bunch E is defined to be the number of elements it has, and is denoted by $\cent E$. For non-finite bunches, $\cent E$ is $null_{\mathbf{N}}$. If an expression E has no elements other than itself, we say it is *atomic* and assert ∇E. On the face of it, it would seem that elementhood and atomicity coincide, and indeed this is the case for booleans and naturals. However, they differ subtly for functions, as we shall see later.

We use lower-case x, y, f, g etc. to stand for variable names; variables always denote elements. We use E, F, G to stand for (bunch) terms of arbitrary type, and P, Q, R to stand for (bunch) terms of boolean type. We write $G[X\backslash E]$ to denote the expression G with E substituted for the placeholder X, with the usual caveat that bound variables might have to be renamed to avoid capture. An expression G is said to be *monotone in* X iff $E : F \Rightarrow G[X\backslash E] : G[X\backslash F]$.

3 The specification language

We describe the language we treat. As far as types are concerned, we restrict ourselves to booleans ($\mathbf{B}$), natural numbers ($\mathbf{N}$), and function types:

$$T ::= \mathbf{B} \mid \mathbf{N} \mid T{\to}T.$$

Note that there is no type '$Bunch\ T$'. Every expression of type T is a bunch of type T, regardless of whether it is a singleton bunch or not.

We give separate grammars for boolean expressions ($E_{\mathbf{B}}$), natural expressions ($E_{\mathbf{N}}$), and function expressions, say of type $T{\to}U$ ($E_{T\to U}$). Some language constructs are common to every type; these are factored out as the syntactic class C_T for type T.

The grammar of boolean expressions is:

$$\begin{aligned}
E_{\mathbf{B}} ::= {} & \mathit{true} \mid \mathit{false} \\
\mid \ & \neg E_{\mathbf{B}} \mid E_{\mathbf{B}} \wedge E_{\mathbf{B}} \mid E_{\mathbf{B}} \vee E_{\mathbf{B}} \mid E_{\mathbf{B}} \Rightarrow E_{\mathbf{B}} \mid E_{\mathbf{B}} \Leftrightarrow E_{\mathbf{B}} \\
\mid \ & \forall x{:}T \cdot E_{\mathbf{B}} \mid \exists x{:}T \cdot E_{\mathbf{B}} \\
\mid \ & \varDelta E_T \mid \nabla E_T \mid \mathbf{fin}\, E_T \\
\mid \ & E_T : E_T \mid E_T \equiv E_T \mid E_T = E_T \\
\mid \ & C_{\mathbf{B}}.
\end{aligned}$$

We will axiomatise conjunction and disjunction so that they behave as minimisation operations with respect to the total orders $<_\wedge$ and $<_\vee$, respectively:

$$\mathit{false} <_\wedge \mathit{null}_{\mathbf{B}} <_\wedge \mathit{true}, \mathit{false} <_\wedge \mathit{true}$$

$$\mathit{true} <_\wedge \mathit{null}_{\mathbf{B}} <_\wedge \mathit{true}, \mathit{false} <_\wedge \mathit{false}.$$

Observe that this preserves the meaning of $\wedge$ and $\vee$ when restricted to the two classical boolean values. These orderings of the booleans are not arbitrary, but have been empirically determined in order to yield an elegant algebra. For example, the reader may easily verify that conjunction is symmetric, associative, idempotent, has *false* as zero and *true* as unit. Negation will be extended to the strange booleans by postulating that it distributes over bunch union and is strict with respect to $\mathit{null}_{\mathbf{B}}$. The implication $P \Rightarrow Q$ will be defined to be equivalent to $(P \not\equiv \mathit{true}) \vee Q$. Universal and existential quantification will be treated as generalised conjunction and disjunction, respectively. As a shorthand, bunches are allowed to occur as the types of the bound variables of universal and existential quantifications (in the following, $E{:}T$):

$$\begin{aligned}
(\forall x{:}E \cdot P) &\mathrel{\hat{=}} (\forall x{:}T \cdot x{:}E \Rightarrow P) \\
(\exists x{:}E \cdot P) &\mathrel{\hat{=}} (\exists x{:}T \cdot x{:}E \wedge P)
\end{aligned}$$

We comment further on the booleans below.

The grammar of natural expressions is:

$$\begin{aligned}
E_{\mathbf{N}} ::= {} & 0 \\
\mid \ & \mathit{succ}\ E_{\mathbf{N}} \\
\mid \ & E_{\mathbf{N}} + E_{\mathbf{N}} \\
\mid \ & \cent E_T \\
\mid \ & C_{\mathbf{N}}.
\end{aligned}$$

As usual, we write 1, 2, etc. in place of $succ\ 0$, $succ\ (succ\ 0)$, etc.

The grammar of functions of type $T{\to}U$ is:

$$\begin{aligned} E_{T\to U} ::= {} & \lambda x{:}T \cdot E_U \\ \mid {} & \mu f{:}T{\to}U \cdot \lambda x{:}T \cdot E_U \\ \mid {} & C_{T\to U}. \end{aligned}$$

The function that maps x of type T to E is written as the λ-abstraction $\lambda x{:}T \cdot E$. In programming languages, recursive functions are usually defined like so: $f(x) \mathrel{\hat{=}} E$ where f and x may occur free in E. In contrast, we define f (without naming it) as $\mu f{:}T{\to}U \cdot \lambda x{:}T \cdot E$. One may omit the type annotation on f.

The grammars for booleans, naturals, and functions share a common part:

$$\begin{aligned} C_T ::= {} & E_T, E_T \mid E_T{}' E_T \\ \mid {} & x_T \\ \mid {} & \textbf{if } E_{\mathbf{B}} \textbf{ then } E_T \textbf{ else } E_T \\ \mid {} & (\S x{:}T \mid E_{\mathbf{B}}) \\ \mid {} & E_{U\to T}\ E_U. \end{aligned}$$

Operator precedence is as follows, running from highest to lowest (operators grouped together have the same precedence):

$$\Delta\ \neg\ \nabla\ \not\text{c}\ \mathbf{fin} \qquad + \qquad = \qquad ,\ {}' \qquad : \qquad \wedge\ \vee \qquad \Rightarrow\ \Leftarrow\!\!\!\Rightarrow \qquad \equiv$$

Substitution binds tightest of all, followed immediately by function application.

We have to give a meaning to conditionals and prescriptions when their constituent booleans take on a strange value, as in $(\S x{:}\mathbf{N} \mid even\,(x, 1))$, or **if** $even\,(0, 1)$ **then** 0 **else** 1? How we do this has little practical importance (because they should not occur in any respectable specification or program) and is of entirely local significance. We make our choice so that it gives rise to the most pleasant set of laws to work with, and that choice is to define $\S x{:}T \mid P$ to be equivalent to $\S x{:}T \mid (P \equiv \mathit{true})$, and **if** P **then** E **else** F to be equivalent to all whenever P is not elementary. The reader may well prefer to make different design decisions, but we have found these to be comfortable to work with. Function application is written by juxtaposition. To give meaning to the application of functions to arbitrary bunches, we will postulate that application distributes over bunch union on both sides, and that it is strict with respect to $null$ on both sides.

The language contains many non-algorithmic constructs. This makes it convenient for writing specifications, but these cannot in general be run on a machine. The associated programing langauge is a subset containing algorithmic components only. To implement a specification, we have to re-write it in the limited language of the programming langauge. The non-algorithmic operators are $\Rightarrow$, $\forall$, $\exists$, Δ, ∇, **fin**, ‘:’, $\equiv$, $\not\text{c}$, and $\S$.

The extra boolean values seem on the face of it to entail a complexity out of proportion to their benefits, and it might seem attractive to try and exclude them. To see that that would not be easy, observe that we can define a boolean-valued function $even$ on the naturals as $\lambda x{:}\mathbf{N} \cdot (\exists y{:}\mathbf{N} \cdot x \equiv y + y)$. Assume that applications of $even$ distribute over bunch union and are strict with respect to $null_{\mathbf{N}}$. (This holds in our theory and in that of [NH93]; [Heh93] gives us the choice, and our choice seems to be an entirely reasonable one). Now observe that the boolean term $even\,(0,1)$ reduces to *true*, *false*, and $even\ null_{\mathbf{N}}$ reduces to $null_{\mathbf{B}}$. We might try to insist that the arguments of boolean-valued functions be elementary, but this is at best inconvenient and at worst unenforceable because elementhood is not in general decidable. It seems that we have to live with the four boolean values.

It seems entirely natural to postulate that the boolean operators should behave like functions in distributing over bunch union and being strict with respect to $null_{\mathbf{B}}$. However, this leaves them with poor algebraic properties (one can easily check by truth tables that, for example, implication is not reflexive, transitive, or antisymmetric, and *false* is not a zero of conjunction nor a unit of disjunction). Instead we have proceeded differently, as described above. Our approach is closely related to the four-valued logic E4 of [MB98]. It differs somewhat because E4 has a value corresponding to “undefined” in place of $null_{\mathbf{B}}$. For brevity we will not repeat here the motivation behind the design decisions underlying our approach to four-valued logic, and instead refer the interested reader to [MB98].

4 Proof theory

We axiomatise the language in five steps. First we give a partial axiomatisation of the boolean type. In essence, this sets down that part of classical two-valued logic which continues to hold in the presence of bunches, but says nothing about the strange truth values $null_{\mathbf{B}}$ and *true*, *false*. Second, we formalise bunches and bunch operations. We can then apply the theory of bunches to deduce that the truth values are precisely *true* and *false* and *true*, *false* and $null_{\mathbf{B}}$. Third, we complete the axiomatisation of the booleans by describing how the boolean operators interact with the bunch operators. Fourth, we axiomatise equality and the naturals. Finally, we define (possibly recursive) functions. A subsection is devoted to each step.

4.1 The core boolean axioms

The core axioms of the booleans are given in Fig. 1. Further theorems are generated by applying Modus Ponens and Generalization:

$$\frac{P \qquad P \Rightarrow Q}{Q} \qquad \frac{P}{\forall x{:}T \cdot P}$$

In practice, we do not use these inference rules directly, but employ textual substitution mechanisms that reduce proving to an algebraic style of “replacing equals with equals”; see [DS90] and [GS93]. In case the reader is momentarily taken aback by seeing $P \equiv true$ (rather than simply P) in some formulae, we point out that the axioms do not imply the truth of $(P \equiv true) \equiv P$. All the axioms of Fig. 1 hold in classical two-valued logic, the only point of difference being the occasional appearance of ΔE for some term E. In traditional logics, ΔE is universally *true*, and indeed it can be shown that the axioms of Fig. 1 generate classical two-valued logic if we postulate the additional axiom ΔE. The deduction theorem holds. Observe that the core axioms make no mention of bunch theory. We may deduce that the truth values include *true* and *false*, and that these are distinct, but we can infer nothing about the presence or absence of other truth values. Nevertheless, the core axioms give rise to a host of theorems. A small selection is listed in Fig. 2, and a comprehensive list is given in [MB99d]. For brevity, only a few theorems about the quantifiers are listed, because other than those listed and the instantiation axiom, all the familiar properties of quantified terms continue to hold. A comprehensive account of the family of many-valued logics which includes the present one can be found in [MB99b]. [MB99b] gives formal proofs of all the logic theorems referred to in this paper, and many more.

We will complete the axiomatisation of the logic to accommodate $null_{\mathbf{B}}$ and *true*, *false* when we have axiomatised bunches in general.

4.2 Bunches

Bunch theory has six basic connectives. We formalise each one with a single axiom, and add a further one to assert the antisymmetry of the subbunch relation:

:-definition	$E : F \equiv (\forall x{:}T \cdot x : E \Rightarrow x : F)$, where $E, F{:}T$
antisymmetry	$(E \equiv F) \equiv (E : F) \wedge (F : E)$
,-definition	$x : E, F \equiv x : E \vee x : F$
′-definition	$x : E'F \equiv x : E \wedge x : F$
§ definition	$y : (\S x{:}T \mid P) \equiv (\exists x{:}T \cdot (P \equiv \textit{true}) \wedge y : x)$
conditional	$E : (\textbf{if } P \textbf{ then } F \textbf{ else } G) \equiv (P \Rightarrow E : F) \wedge (\neg P \Rightarrow E : G)$

Equivalence

$\equiv$-reflexivity	$E \equiv E$
$\equiv$-symmetry	$(E \equiv F) \equiv (F \equiv E)$
$\equiv$-truth	$((E \equiv F) \equiv \mathit{true}) \equiv (E \equiv F)$

Negation

exchange	$(\neg P \equiv Q) \equiv (\neg Q \equiv P)$
false-definition	$\mathit{false} \equiv \neg \mathit{true}$
$\not\equiv$-definition	$(E \not\equiv F) \equiv \neg(E \equiv F)$

Disjunction and conjunction

$\vee$-symmetry	$P \vee Q \equiv Q \vee P$
$\vee$-associativity	$P \vee (Q \vee R) \equiv (P \vee Q) \vee R$
$\vee$-idempotency	$P \vee P \equiv P$
$\vee$-zero	$P \vee \mathit{true} \equiv \mathit{true}$
$\vee$-unit	$P \vee \mathit{false} \equiv P$
$\wedge$-definition	$P \wedge Q \equiv \neg(\neg P \vee \neg Q)$

Implication

$\Rightarrow$-definition	$P \Rightarrow Q \equiv (P \not\equiv \mathit{true}) \vee Q$
$\Rightarrow/\equiv$	$P \Rightarrow (Q \equiv R) \equiv (P \Rightarrow Q \equiv P \Rightarrow R)$
$\equiv$-weakening	$(P \equiv Q) \Rightarrow (P \Rightarrow Q)$
Leibniz	$(E \equiv F) \Rightarrow (G[X \backslash E] \equiv G[X \backslash F])$
shunting	$P \wedge Q \Rightarrow R \equiv P \Rightarrow (Q \Rightarrow R)$
$\Rightarrow/\wedge$	$(P \Rightarrow Q \wedge R) \equiv (P \Rightarrow Q) \wedge (P \Rightarrow R)$
$\vee/\Rightarrow$	$(P \vee Q \Rightarrow R) \equiv (P \Rightarrow R) \wedge (Q \Rightarrow R)$
$\Leftrightarrow$-definition	$P \Leftrightarrow Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

Universal and existential quantification

$\forall/\wedge$	$(\forall x{:}T \cdot P \wedge Q) \equiv (\forall x{:}T \cdot P) \wedge (\forall x{:}T \cdot Q)$
$\Rightarrow/\forall$	$P \Rightarrow (\forall x{:}T \cdot Q) \equiv (\forall x{:}T \cdot P \Rightarrow Q)$, where x not free in P
$\forall/\equiv$	$(\forall x{:}T \cdot P \equiv Q) \Rightarrow ((\forall x{:}T \cdot P) \equiv (\forall x{:}T \cdot Q))$
$\forall$-truth	$((\forall x{:}T \cdot P) \equiv \mathit{true}) \equiv (\forall x{:}T \cdot P \equiv \mathit{true})$
interchange	$(\forall x{:}T \cdot (\forall y{:}U \cdot P)) \equiv (\forall y{:}U \cdot (\forall x{:}T \cdot P))$
renaming	$(\forall x{:}T \cdot P) \equiv (\forall y{:}T \cdot P[x \backslash y])$, where y is fresh
$\exists$-definition	$(\exists x{:}T \cdot P) \equiv \neg(\forall x{:}T \cdot \neg P)$
$\exists/\Rightarrow$	$(\exists x{:}T \cdot P) \Rightarrow Q \equiv (\forall x{:}T \cdot P \Rightarrow Q)$, where x not free in Q
$\exists$-truth	$((\exists x{:}T \cdot P) \equiv \mathit{true}) \equiv (\exists x{:}T \cdot P \equiv \mathit{true})$

Elementhood and instantiation

Δ-definition	$\Delta P \equiv ((P \equiv \mathit{true}) \equiv P)$
instantiation	$(\forall x{:}T \cdot P) \wedge \Delta t \Rightarrow P[x \backslash t]$
variables	$\forall x{:}T \cdot \Delta x$

Fig. 1. Core axioms of the booleans

Core theorems

$\equiv$ is transitive
$\neg$ is an involution
$\wedge$ is symmetric, associative, and idempotent with zero *false* and unit $true$
$\vee$ is symmetric, associative, and idempotent with zero *true* and unit $false$
de Morgan's laws for $\wedge$ and $\vee$
$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$
$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$
$(E \equiv F) \Rightarrow P[X \backslash E] \equiv (E \equiv F) \Rightarrow P[X \backslash F]$
$(E \equiv F) \wedge P[X \backslash E] \equiv (E \equiv F) \wedge P[X \backslash F]$
false $\not\equiv$ *true*
$\Rightarrow$ is reflexive and transitive with *true* as a left unit and right zero
$\Rightarrow$ distributes on the right over $\vee$ and $\Rightarrow$
$P \wedge (P \Rightarrow Q) \Rightarrow Q$
$P \wedge (Q \Rightarrow P) \equiv P$
$P \Rightarrow P \vee Q$ and $P \wedge Q \Rightarrow P$
$\Rightarrow$ is $\Rightarrow$-monotone in its 2nd argument, and $\Rightarrow$-antimonotone in its 1st.
$\wedge$ and $\vee$ are $\Rightarrow$-monotone in both arguments
$P \wedge (\neg P \vee Q) \equiv P \wedge Q$, if ΔP
$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$, if ΔP
$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$, if ΔP
$P \vee (Q \equiv R) \equiv (P \vee Q \equiv P \vee R)$, if ΔP
$P \Rightarrow (Q \equiv R) \equiv (P \wedge Q \equiv P \wedge R)$, if ΔP
true, *false*, $E \equiv F$, and ΔE are elementary
$P \vee \neg P \vee \neg \Delta P$
$\Delta P \equiv (P \equiv \textit{true}) \vee (P \equiv \textit{false})$
$\Delta E \equiv (\exists x{:}T \cdot E \equiv x)$
$P \vee (\forall x{:}T \cdot Q) \equiv (\forall x{:}T \cdot P \vee Q)$, if ΔP and x not free in Q
$P \wedge (\exists x{:}T \cdot Q) \equiv (\exists x{:}T \cdot P \wedge Q)$, if ΔP and x not free in Q
$(\forall x{:}T \cdot (x \equiv t) \Rightarrow P) \equiv P[x \backslash t]$, if Δt and x not free in t

Fig. 2. Some core theorems

The axiom defining prescriptions appears in the literature as $y : (\S x{:}T \mid P) \equiv P[x \backslash y]$. However, that version accommodates neither strange booleans nor non-flat types. The version above reduces to the original version under the assumption that T is flat and P is elementary. The operational interpretation of conditionals as explained earlier was motivated by the simplicity of the axiom given above. The remaining bunch notation can be understood as shorthand (we explain $One2One[E; n]$ below):

$null$	$null_T \equiv (\S x{:}T \mid \textit{false})$
all	$all_T \equiv (\S x{:}T \mid \textit{true})$
∇-definition	$\nabla E \equiv (E \not\equiv null) \wedge (\forall x{:}T \cdot (x : E) \equiv (x \equiv E))$
fin-definition	$\mathbf{fin}\, E \equiv (\exists n{:}\mathbf{N} \cdot One2One[E; n])$
$¢$-definition	$¢E \equiv (\S n{:}\mathbf{N} \mid One2One[E; n])$

It can now be proven that $null_{\mathbf{B}}$ and *true* and *false* and *true*, *false* account for all the booleans, and that they each differ from one another. From the

	Algebraic connectives
$\equiv$:	$(E \equiv F) \equiv (\forall x{:}T \cdot x : E \equiv x : F)$ where $E, F{:}T$
Δ:	$\Delta(E : F)$
: reflexive	$E : E$
: transitive	$E : F \wedge F : G \Rightarrow E : G$
Δ,	$\Delta(E, F) \equiv (\Delta E \wedge F : E) \vee (\Delta F \wedge E : F)$
, zero	$E, all \equiv all$
, unit	$E, null \equiv E$
, idem.	$E, E \equiv E$
, comm.	$E, F \equiv F, E$
, assoc.	$E, (F, G) \equiv (E, F), G$
, least upper bound	$E, F : G \equiv E : G \wedge F : G$
	Extremes
least	$null : E$
highest	$E : all$
four truth values	$(P \equiv null) \vee (P \equiv \mathit{true}) \vee (P \equiv \mathit{false}) \vee (P \equiv \mathit{true}, \mathit{false})$
	Prescriptions
*	$(\S x{:}T \mid x : E) \equiv E$

Fig. 3. Some bunch theorems

core axioms and the bunch axioms many bunch theorems can be proven. Some are listed in Fig. 3. They include all the algebraic properties that one would expect. See [MB99b] for proofs of these and related theorems.

Since the language has functions and natural numbers, $\mathbf{fin}\, E$ and ¢E can be defined by the usual counting tricks. First, define a predicate $One2One[E; n]$ which holds iff the bunch E (of type T) has n elements. $One2One[E; n]$ can be defined like this:

$$\begin{aligned} \exists f{:}\mathbf{N} \to T \cdot \quad & (\forall i : 0..n \cdot \Delta(f\ i)) \\ & \wedge (\forall i, j : 0..n \cdot i \neq j \Rightarrow (f\ i \not\equiv f\ j)) \\ & \wedge (\forall e{:}T \cdot e : E \equiv (\exists i : 0..n \cdot f\ i \equiv e)) \end{aligned}$$

We have used $0..n$ above for the bunch consisting of the natural numbers i such that $0 \leq i < n$. It can be proven that for all naturals m and n, $One2One[E; n] \wedge One2One[E; m] \Rightarrow n = m$, as we should expect.

Some theorems about cardinality are listed in Fig. 4. As expected, only $null$ has cardinality 0, and only atomic bunches have cardinality 1.

4.3 Bunches and booleans

We can now complete the axiomatisation of the booleans by supplying the axioms that fix the behaviour of the boolean operators in relation to the two strange booleans *true*, *false* and $null_{\mathbf{B}}$. There are just three of them:

	Cardinality
$null$	$(\cent E \equiv 0) \equiv (E \equiv null)$
∇	$(\cent E \equiv 1) \equiv \nabla E$
$\Delta\cent$	$\Delta(\cent E) \equiv \mathbf{fin}\, E$

Fig. 4. Some cardinality theorems

	Boolean operators
$\wedge null$	$\mathit{true}, \mathit{false} \wedge null_{\mathbf{B}} \equiv null_{\mathbf{B}}$
$\neg/,$	$\neg(P, Q) \equiv \neg P, \neg Q$
$\forall null$	$((\forall x{:}T \cdot P) \equiv null_{\mathbf{B}}) \equiv (\forall x{:}T \cdot P \not\equiv \mathit{false}) \wedge (\exists x{:}T \cdot P \equiv null_{\mathbf{B}})$
$\Rightarrow$ second mon.	$P : Q \Rightarrow (R \Rightarrow P) : (R \Rightarrow Q)$
$\Rightarrow$ second distr.	$P \Rightarrow (Q, R) \equiv (P \Rightarrow Q), (P \Rightarrow R)$

Fig. 5. Some theorems about bunches and booleans

	Connectives
$\neg null$	$\neg null_{\mathbf{B}} \equiv null_{\mathbf{B}}$
$\vee null$	$\mathit{true}, \mathit{false} \vee null_{\mathbf{B}} \equiv null_{\mathbf{B}}$
$\exists null$	$((\exists x{:}T \cdot P) \equiv null_{\mathbf{B}}) \equiv (\forall x{:}T \cdot P \not\equiv \mathit{true})$
	$\wedge(\exists x{:}T \cdot P \equiv null_{\mathbf{B}})$

The above axioms and those of Fig. 1, with the two inference rules, constitute a four-valued logic which we call **EB** (“equational bunch” logic). Figure 5 lists further theorems that can now be proven; see [MB99b] for proofs of these. Naturally, compromises have to be made to accommodate the strange values, and so not all properties hold that one might have wished for. Among the more regretable losses are the fact that implication is not antisymmetric, or distributive in its first argument. Neither does the axiom of the excluded middle hold, and instantiation may only be carried out with respect to elements. Not all losses are significant, however. For example, our disjunction and conjunction do not preserve $null_{\mathbf{B}}$, and they are not distributive or monotone with respect to the subbunch relation, but we are not too bothered by this because monotonicity with respect to *implication* is more important for these connectives and indeed holds.

4.4 The naturals and equality

So far, the only type that has been axiomatised is the boolean type $\mathbf{B}$. Clearly, any usable theory will have other base types, typically some number types, characters, and enumerated types. Axiomatising such types is standard, and orthogonal to bunch theory; we concern ourselves only with the added axioms necessary to describe their properties in the presence of bunches. In the case of $\mathbf{N}$:

	Equality
$=$ refl.	$\nabla E \Rightarrow E = E$
$=$ sym.	$E = F \equiv F = E$
$=$ trans.	$E = F \wedge F = G \Rightarrow E = G$
$=/\Rightarrow$	$E{=}F \Rightarrow (E \equiv F)$
$= null$	$E{=}null \equiv null_{\mathbf{B}}$
$=/,$	$E{=}(F, G) \equiv (E = F), (E = G)$
$= \mathbf{B}$	$P{=}Q \equiv (P \wedge Q) \vee (\neg P \wedge \neg Q)$

Fig. 6. Some theorems about equality

	N
N flat	$\forall x, y{:}\mathbf{N} \cdot x : y \Rightarrow (x \equiv y)$
constants	$\Delta 0$
$\Delta succ$	$\forall n{:}\mathbf{N} \cdot \Delta(succ\ n)$
$succ$:	$x : succ\ E \equiv (\exists e : E \cdot x \equiv succ\ e)$
$+$ dist	$x : (E + F) \equiv (\exists e : E \cdot \exists f : F \cdot x \equiv e + f)$

First, the type of naturals is flat. Second, 0 and all outcomes of applying $succ$ are elementary. Axiom $succ$: states that x is a posssible outcome of applying $succ$ to a non-elementary bunch iff it is an outcome of applying it to one of the constituents of the bunch. Similar remarks hold for axiom $+$ dist. It follows that $succ$ and $+$ distribute over bunch union and are strict with respect to $null_{\mathbf{N}}$.

Equality is axiomatised as follows:

	Equality
true :=	*true* : $(E = F) \equiv (\exists e : E \cdot \exists f : F \cdot e \equiv f)$
false :=	*false* : $(E = F) \equiv (\exists e : E \cdot \exists f : F \cdot e \not\equiv f)$

Equality is an implementable weakening of equivalence (which is not monotone in either argument and is therefore unimplementable). Some theorems concerning equality are listed in Fig. 6; see [MB99b] for proofs. As expected, equality is symmetric and transitive. However, since it distributes over bunch union, it is reflexive only for atomic arguments (recall that elements of base types are atomic, but not necessarily those of function type as we see shortly). For example, $(1, 2) = (1, 2)$ is *true*, *false*. Although equality is defined on all types its behaviour on non-flat types is uninteresting because it nearly always yields *true*, *false* in those cases.

4.5 Functions and recursion

We present below the axioms for functions and recursion (recall that lower case meta-variables stand for variables, and these can only be instantiated with elements):

$\lambda/:$	$(\lambda x{:}T \cdot E), (\lambda x{:}T \cdot F) : (\lambda x{:}T \cdot E, F)$
λ mon.	$(\forall x{:}T \cdot E : F) \equiv (\lambda x{:}T \cdot E) : (\lambda x{:}T \cdot F)$
ap./$null$	$E\ null \equiv null$
ap./$null$	$null\ E \equiv null$
ap./,	$E\,(F,G) \equiv (E\ F), (E\ G)$
ap./,	$(E,F)\,G \equiv (E\ G), (F\ G)$
ap. mon.	$E : F \Rightarrow E\ G : F\ G$
ap. mon.	$E : F \Rightarrow G\ E : G\ F$
fun. $\equiv$	$(f \equiv g) \equiv (\forall x{:}T \cdot f\ x \equiv g\ x)$, where $f, g{:}T \to U$, some U.
η	$f \equiv (\lambda x{:}T \cdot f\ x)$ where $f{:}T \to U$, some U

Fig. 7. Some theorems about functions

Functions

$\Delta\lambda$	$\Delta(\lambda x{:}T \cdot E)$
$\beta\equiv$	$(\lambda x{:}T \cdot E)\ y \equiv E[x\backslash y]$
ap.:	$x : E\ F \equiv (\exists e : E \cdot (\exists f : F \cdot x : e\ f))$
:-func	$f : g \equiv (\forall x{:}T \cdot f\ x : g\ x)$, where $f, g{:}T \to U$, some U.

Axiom $\Delta\lambda$ encapsulates our decision to regard all λ-terms as elementary, even those whose bodies may be non-elementary for some values of the parameter. An alternative approach is to view only traditional partial (deterministic) functions as elementary; see [MB99c]. Axiom $\beta\equiv$ is familiar β-substitution, but note that it is only applicable to elementary functions and arguments. Axiom ap.: states that the outcomes of applying a bunch of functions to a bunch argument, are the outcomes obtained by applying each constituent function to each constituent of the argument. Axiom :-func defines the subbunch relation on functions in the obvious pointwise way.

In order to avoid inconsistency (see Section 6), we require that the body of every λ-abstraction $\lambda x{:}T \cdot E$ be monotone in the parameter for elements, i.e. $x : y \Rightarrow E : E[x\backslash y]$ for fresh y. This trivially holds for functions on flat types. For non-flat types, monotonicity can be ensured syntactically by forbidding the occurrence of formal parameters in non-monotone positions (for our language, these are the arguments of $\equiv$, ':', Δ, ∇, $\mathbf{fin}$, and $\not\subset$, and the left-hand argument of $\Rightarrow$). The axioms for function types are closely related to those in [MB99c], to which we refer the reader for motivation.

Figure 7 lists some of the theorems about functions. It may be surprising that 'distributing' a λ over bunch union is *not* an equivalence. To see this, consider that the function $even$ described earlier is an element of $\lambda n{:}\mathbf{N} \cdot$ *true*, *false*, but not of $(\lambda n{:}\mathbf{N} \cdot \textit{true}), (\lambda n{:}\mathbf{N} \cdot \textit{false})$. However, λ-abstractions are monotone in their bodies. Application is monotone, preserves $null$, and distributes over bunch union on both sides. Theorem fun. $\equiv$ is just the traditional extensionality rule, but note that it applies to elementary functions only. To prove the equivalence of non-elementary functions we must start out from the bunch axioms, and in particular Theorem $\equiv$:. Note that Theorem η is also only applicable to elementary functions.

For the flat types $\mathbf{B}$ and $\mathbf{N}$, we can deduce that elementhood and atomicity coincide, but not so for functions. For example, like every λ-abstraction, $\lambda n{:}\mathbf{N} \cdot \textit{true}, \textit{false}$ is itself elementary, but it has more than one element. Its elements include the constant functions $\lambda n{:}\mathbf{N} \cdot \textit{true}$ and $\lambda n{:}\mathbf{N} \cdot \textit{false}$, the function $even$, and infinitely many others. The only function of type $T{\to}U$ that is atomic is $\lambda x{:}T \cdot null_U$. Prefixing "$\lambda x{:}T\cdot$" does create an element, but not necessarily an atom.

We axiomatise recursion in the usual way — as a fixpoint, and the greatest postfixpoint with respect to the subbunch relation. The body $\lambda x{:}T \cdot E$ of the recursive function must be monotone in f to ensure that the fixpoint exists.

Recursion

unfold	$(\mu f \cdot \lambda x{:}T \cdot E) \equiv (\lambda x{:}T \cdot E[f \backslash \mu f \cdot \lambda x{:}T \cdot E])$
postfix	$F : (\lambda x{:}T \cdot E[f \backslash F]) \Rightarrow F : (\mu f \cdot \lambda x{:}T \cdot E)$

4.6 Reasoning about specifications

We have described a theory which supports all the every-day reasoning about program correctness that a formal programmer is likely to engage in. The theory is presented as a set of axioms, together with the inference rules modus ponens and generalisation. The axioms are organised as a set of axioms for each type, including the boolean type. We regard the axioms as boolean terms. This is possible because the booleans of specification languages support a rich notation, including arbitrary quantifications. Even the subbunch relation (which corresponds precisely to the refinement relation of [Bac80, Mor88, Mor87]) and equivalence are regarded as booean operators. It follows that we can reason about specifications within the language without having to resort to a logic at the meta-level. This approach is also taken in [NH93, Heh93], and seems to us to be an economic way to organise a theory of programming from specifications.

5 Model

5.1 Mathematical preliminaries

We provide a denotational semantics for the language and show that our theory is consistent. The model is derived from that in [MB98, MB99c], and is expressed in terms of sets and functions on sets.

Let S, P and E stand for a set, predicate and expression, respectively, in the language of the model. We will write $\{x \in S \mid P \cdot E\}$ to denote

T	$[T]$	$t \leq^T u$
$\mathbf{B}$	$\{tt, ff\}$	$t = u$
$\mathbf{N}$	$\{0, 1, \ldots\}$	$t = u$
$U \to V$	$[U] \overset{mon}{\to} \mathcal{P}_{\downarrow}[V]$	$\forall v \in [U] \cdot t\ v \subseteq u\ v$

Fig. 8. Interpretations of the types

the set containing the values denoted by E for each binding of the dummy variable x to those elements of S that satisfy P. This notation generalises in the obvious way to more than one dummy variable. If E is just x, we write $\{x \in S \mid P\}$ for short, and if P is *true* we write $\{x \in S \cdot E\}$. We will re-use some language notation in the model, such as 0, $+$, and function application; in exchange for this concession, the reader is spared some needless syntactic clutter. We assume a basic familiarity with the notions of a complete partially ordered set (*cpo*) and the least fixpoint of a function on a *cpo*.

The notion of downclosed sets is central in the model. Define the *downclosure* $S{\downarrow_{\leq}}$ of a set S that is a subset of partially ordered set $(W, \leq)$ as follows:

$$S{\downarrow_{\leq}} \mathrel{\hat{=}} \{p \in W \mid (\exists s \in S \cdot p \leq s)\}.$$

A set is said to be *downclosed* if it is equivalent to its downclosure. We call a downclosed set a *downset*. Let $(W, \leq)$ be a partially ordered set; we define $\mathcal{P}_{\downarrow}W$ to be the set of downclosed subsets of W. It is routine to show that $(\mathcal{P}_{\downarrow}W, \subseteq)$ is a cpo (and indeed a complete lattice), if $(W, \leq)$ is a partially ordered set. Its lub and glb operators are $\bigcup$ and $\bigcap$, respectively. The only thing to check in the proof is that arbitrary unions and intersections of downsets are downsets, and indeed they are.

5.2 Types and environments

We associate with each type T a set $[T]$, partially ordered by $\leq^T$. See Fig. 8. $[U] \overset{mon}{\to} \mathcal{P}_{\downarrow}[V]$ denotes the monotone functions from $[U]$ to $\mathcal{P}_{\downarrow}[V]$; they are monotone in the sense $t \leq^U u \Rightarrow f\ t \subseteq f\ t'$. It is easy to show that $([T], \leq^T)$ is a partially ordered set in each case. We abbreviate $S{\downarrow_{\leq^T}}$ by $S{\downarrow}$ when $\leq^T$ can be inferred from context.

An environment ρ is a mapping from each variable of type T to an element of $[T]$. In addition, it maps each recursion dummy f of type $T \to U$ to an element of $\mathcal{P}_{\downarrow}[T \to U]$. We denote by $\rho[x \mapsto v]$ the environment which is the same as ρ except that it associates v with variable x. Every language term E of type T is interpreted as a set with respect to an environment ρ; we denote the set by $[E]\rho$. Intuitively, $[E]\rho$ is the element of $\mathcal{P}_{\downarrow}[T]$ consisting of all the elements (in the bunch-theoretic sense) of E. We give some examples to aid

Boolean X	has interpretation $[X]\rho$
true	$\{tt\}$
false	$\{ff\}$
$\neg P$	$\{p \in [P]\rho \cdot negate(p)\}$
$P \wedge Q$	$Min_\wedge\{[P]\rho, [Q]\rho\}$
$P \vee Q$	$Min_\vee\{[P]\rho, [Q]\rho\}$
$P \Rightarrow Q$	if $[P]\rho = \{tt\}$ then $[Q]\rho$ else $\{tt\}$
$\forall x{:}T \cdot P$	$Min_\wedge\{t \in [T] \cdot [P]\rho[x \mapsto t]\}$
$\exists x{:}T \cdot P$	$Min_\vee\{t \in [T] \cdot [P]\rho[x \mapsto t]\}$
$\varDelta E$	if $[E]\rho$ has a maximum then $\{tt\}$ else $\{ff\}$
∇E	if $\lvert[E]\rho\rvert = 1$ then $\{tt\}$ else $\{ff\}$
$\mathbf{fin}\, E$	if $\lvert[E]\rho\rvert \in \{0, 1, \ldots\}$ then $\{tt\}$ else $\{ff\}$
$E : F$	if $[E]\rho \subseteq [F]\rho$ then $\{tt\}$ else $\{ff\}$
$E \equiv F$	if $[E]\rho = [F]\rho$ then $\{tt\}$ else $\{ff\}$
$E = F$	$\{e \in [E]\rho, f \in [F]\rho \cdot equal(e, f)\}$

Fig. 9. Interpretation of the booleans

intuition. The term 2 is interpreted in environment ρ as $[2]\rho$, which reduces to $\{2\}$ for any ρ. The bunch $2, x$ is interpred as $[2, x]\rho$ in environment ρ, which redues to $\{2, 3\}$ when ρ associates x with 3. In all environments, $\lambda x{:}\mathbf{N} \cdot 2, 3$ is interpreted as the set containing all the (infinitely many) functions that map each element of $[\mathbf{N}]$ to a subset of $\{2, 3\}$. Not surprisingly perhaps, the subbunch relation will be interpreted as $\subseteq$.

5.3 Base types

The interpretations of boolean terms is given in Fig. 9. In defining conjunction and disjunction, we use the total orders $<_\wedge$ and $<_\vee$, defined by:

$$\{ff\} <_\wedge \{\} <_\wedge \{tt, ff\} <_\wedge \{tt\}$$

$$\{tt\} <_\vee \{\} <_\vee \{tt, ff\} <_\vee \{ff\}.$$

Taking the minimum with respect to $<_\wedge$ ($<_\vee$) will be written $Min_\wedge$ ($Min_\vee$). The auxiliary function $negate$ swaps *tt* and *ff*. The auxiliary function $equal$ delivers *tt* if its two arguments are equal, and *ff* otherwise. The phrase "$[E]\rho$ has a maximum" is formally expressed as $\exists m \in [E]\rho \cdot (\forall x \in [E]\rho \cdot x \leq^T m)$, where $E{:}T$.

The semantics of natural expressions is without surprises; Fig. 10 gives the details.

5.4 Function types

The semantics of function expressions is given in Fig. 11. (The downclosure arrow in the semantics of $\lambda x{:}T \cdot E$ is in fact redundant, as it can be shown

Natural X	has interpretation $[X]\rho$
0	$\{0\}$
$succ\ E$	$\{e \in [E]\rho \cdot e + 1\}$
$E + F$	$\{e \in [E]\rho, f \in [F]\rho \cdot e + f\}$
$¢E$	if $\|[E]\rho\| \in \{0, 1, \ldots\}$ then $\{\|[E]\rho\|\}$ else $\{\}$

Fig. 10. Interpretation of the naturals

Function X	has interpretation $[X]\rho$
$\lambda x{:}T \cdot E$	$\{f \in [T{\to}U] \mid (\forall v \in [T] \cdot f\ v \subseteq [E]\rho[x \mapsto v])\}{\downarrow}$
rec. dummy f	ρf
$\mu f \cdot \lambda x{:}T \cdot E$	μF where
	$F{:}\mathcal{P}_{\downarrow}[T{\to}U] \overset{mon}{\to} \mathcal{P}_{\downarrow}[T{\to}U]$
	$F\ S \mathrel{\hat{=}} [\lambda x{:}T \cdot E]\rho[f \mapsto S]$

Fig. 11. Interpretation of functions of type $T{\to}U$

Generic X	has interpretation $[X]\rho$
E, F	$[E]\rho \cup [F]\rho$
$E'F$	$[E]\rho \cap [F]\rho$
$\S x{:}T \mid P$	$\{t \in [T] \mid [P]\rho[x \mapsto t] = \{tt\}\}{\downarrow}$
if P **then** E **else** F	if $[P]\rho = \{tt\}$ then $[E]\rho$ else
	(if $[P]\rho = \{ff\}$ then $[F]\rho$ else $[T]$)
	where $E, F{:}T$
$E\ F$	$\bigcup\{e \in [E]\rho, f \in [F]\rho \cdot e\ f\}$
x	$\{\rho\ x\}{\downarrow}$

Fig. 12. Interpretation of generic expressions

that the set is downclosed anyhow.) Notice that the recursive expression $\mu f \cdot \lambda x{:}T \cdot E$ is interpreted as the fixpoint of the functional F as defined in Fig. 11. F acts on $\mathcal{P}_{\downarrow}[T{\to}U]$ which is a *cpo* under the order $\subseteq$. By the language restriction that $\lambda x{:}T \cdot E$ must be refinement-monotone in f, we ensure that F is $\subseteq$-monotone, and therefore the generalized limit theorem [HP72, Nel89] ensures that the least fixpoint exists.

5.5 Generic expressions

The semantics of the generic expression forms is given in Fig. 12.

It can be routinely verified that for every expression E of type T and every ρ, $[E]\rho$ is indeed an element of $\mathcal{P}_{\downarrow}[T]$. In the case of function application, note that $[E\ \ F]\rho$ is downclosed because unions of downclosed sets are downclosed and functions are interpreted by functions to downclosed sets.

5.6 Consistency

All of the above establishes the machinery for tackling our main task. We have to show that for every axiom P, we have $[P]\rho = \{tt\}$ for arbitrary environment ρ, and that the inference rules preserve this property for the theorems they generate. These proofs are largely mechanical, and for brevity we omit them. Consistency can now be easily inferred, i.e. that for no formula P is both P and $\neg P$ a theorem.

6 Concluding remarks

We have made a Hilbert-style axiomatic presentation of a small functional language which uses bunches to support nondeterminacy and underspecification. Our approach to axiomatisation, based on giving primacy to subbunching rather than bunch union, is different in style from anything previously proposed, and fills the known gaps in previous axiomatisations. The language includes boolean and functional types whose behaviour, as is well known, is seriously complicated by the presence of nondeterminacy. Our approach to these types departs from previous practice in order to avoid restrictions that lessen the expressive power of the language. Inconsistency can arise in many ways [Mee86, MB00], and in our experience theories have managed to avoid them only by imposing serious restrictions on how nondeterminacy and functions may interact. No previous theory of bunches has been accompanied by a proof of consistency.

Bunch theory leads to four truth-values. In the published literature, the boolean operators are extended to the four-valued case by postulating that they distribute over bunch union and are strict with respect to $null_{\mathbf{B}}$. We have proposed an axiomatisation that departs significantly from this approach, and which in our view better preserves the familiar algebraic properties of the booleans. Just about all of the workhorse theorems of classical two-valued logic are preserved, and rather few of the laws require that constituent terms be elementary. Of course, not everything in our garden is rosy: for example, our implication is not subbunch-monotone in the first argument, and therefore cannot be considered a part of the programming sublanguage.

In [NH93] a *predicate* is defined to be a boolean expression which is necessarily either *true* or *false* (although the exact language used for predicates is not specified). In order to avoid certain technical problems, only predicates (as opposed to boolean expressions in general) may appear in certain sensitive positions, such as the first position (the guard) in a conditional expression. The inconvenience of this is recognised by the authors of [NH93] who indicate they are looking at a allowing nondeterministic predicates without complicating the laws. We surmise that our logic is just what is required. In any event, it does not require any formal distinction between boolean expressions and predicates.

Our theory of booleans covers the universal and existential quantifiers and is presented in classical style as a set of axioms and inference rules. This has the advantage that we can formally reason about the terms of the languageb within the language itself. This seems to us to support the preferred style of reasoning in [NH93, Heh93], where the non-monotone subbunch operator, and hence term equivalence, is part of the specification language rather than the metalanguage. Of course, our theory does not prevent one having a separate logic for reasoning at the meta-level if that is preferred.

Acknowledgements. We thank three anonymous referees and E. C. R. Hehner for criticism which led to a much improved presentation of the material of the paper.

References

[Bac80] R.-J. R. Back: Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980

[DS90] E. W. Dijkstra, C. S. Scholten: Predicate Calculus and Program Semantics. New York: Springer 1990

[GS93] D. Gries, F. B. Schneider: A Logical Approach to Discrete Math. New York: Springer 1993

[Heh81] E. C. R. Hehner: Bunch theory: a simple set theory for computer science. Inf Process Lett 12: 26–31 (1981)

[Heh84] E. C. R. Hehner: The Logic of Programming. ISBN 0135399661. Englewood Cliffs, NJ: Prentice Hall 1984

[Heh93] E. Hehner: A Practical Theory of Programming. ISBN 0387941061. New York, London: Springer 1993

[HP72] P. Hitchcock, D. Park: Induction Rules and Termination Proofs. In: IRIA Conference on Automata, Languages, and Programming Theory, 1972

[Lee93] René Leermakers: The Functional Treatment of Parsing. ISBN 0-7923-9376-7. Dordrecht: Kluwer 1993

[MB98] J. M. Morris, A. Bunkenburg: Partiality and nondeterminacy in program proofs. Formal Asp Comput Sci 10: 76–96 (1998)

[MB99a] J. M. Morris, A. Bunkenburg: E3: A logic for reasoning equationally in the presence of partiality. Sci Comput Programm 34: 141–158 (1999)

[MB99b] J. M. Morris, A. Bunkenburg: Many-valued Logics for Programming: Theorems and Proofs. Report R-1999-49, University of Glasgow, Department of Computing Science, 1999.
Available from http://www.compapp.dcu.ie/∼jmorris/pubs/manyv.pdf

[MB99c] J. M. Morris, A. Bunkenburg: Specificational functions. ACM Trans Programm Lang Syst 21: 677–701 (1999)

[MB99d] J. M. Morris, A. Bunkenburg: Theorems of EB.
http://www.compapp.dcu.ie/∼jmorris/pubs/eblaws.pdf, University of Glasgow, Department of Computing Science, 1999

[MB00] J. M. Morris, A. Bunkenburg. A source of inconsistency in theories of nondeterministic functions. 2000. Submitted for publication

[Mee86] L. Meertens: Algorithmics – towards programming as a mathematical activity. Math Comput Sci 1: 1–46 (1986). CWI Monographs (J.W. de Bakker, M. Hazewinkel, J. K. Lenstra, eds.) North Holland, Publ. Co

[Mor87] J. M. Morris: A theoretical basis for stepwise refinement and the programming calculus. Sci Comput Program 9: 287–306 (1987)

[Mor88] C. Morgan: The specification statement. ACM Trans Program Lang Syst 10: 403–419 (1988)

[Nel89] G. Nelson: A generalization of Dijkstra's calculus. ACM Trans Program Lang Syst 11(4): 517–561 (1989)

[NH93] T. S. Norvell, E. C. R. Hehner: Logical specifications for functional programs. In: Proceedings of the 2nd International Conference on Mathematics of Program Construction, Oxford, 29 June–3 July 1992, Vol 669 of Lecture Notes in Computer Science, pp 269–290. Berlin Heidelberg New York: Springer 1993