Bunch theory: Axioms, logic, applications and model

Bill Stoddart a,∗, Steve Dunne a, Chunyan Mu b, Frank Zeyda a

a Teesside University, Middlesbrough TS1 3BX, United Kingdom
b University of Aberdeen, United Kingdom

A R T I C L E   I N F O

Keywords:
Bunch-theory
Partial-functions
Logic
Denotational-model
Backtracking
Refinement-calculus

A B S T R A C T

In his book A practical theory of programming [10,12], Eric Hehner proposes and applies a radical reformulation of set theory in which the collection and packaging of elements are seen as separate activities. This provides for unpackaged collections, referred to as “bunches”. Bunches allow us to reason about non-determinism at the level of terms, and, very remarkably, allow us to reason about the conceptual entity “nothing”, which is just an empty bunch (and very different from an empty set). This eliminates mathematical “gaps” caused by undefined terms. We have made use of bunches in a number of papers that develop a refinement calculus for backtracking programs. We formulate our bunch theory as an extension of the set theory used in the B-Method, and provide a denotational model to give this formulation a sound mathematical basis. We replace the classical logic that underpins B with a version that is still able to prove the laws of our logic toolkit, but is unable to prove the property, derivable in classical logic, that every term denotes an element, which for us is pathological since we hold that terms such as 1/0 simply denote “nothing”. This change facilitates our ability to reason about partial functions and backtracking programs. We include a section on our backtracking program calculus, showing how it is derived from WP and how bunch theory simplifies its formulation. We illustrate its use with two small case studies.

1. Introduction

We have published a number of papers which develop a prospective value formalism as a more expressive alternative to weakest preconditions, and we have developed a refinement calculus for reversible and backtracking programs [6,25,22,26,28,31]. We reformulate Abrial’s Generalised Substitution Language (GSL) to give backtracking GSL (BGSL), which differs from GSL in including preferential and probabilistic choice, in expressing non-determinism at the level of terms, and in allowing results from provisional computations (which return results without changing program state) to be included in our expression language.

We make use of the radical reformulation of set theory proposed by Hehner in “A Practical Theory of Programming” [10,12] in which collection and packaging are separate activities. This provides for unpackaged collections, referred to as “bunches”. Bunches allow us to reason about non-determinism at the level of terms, and, very remarkably, allow us to reason about the conceptual entity “nothing”, which we refer to as null. This is just an empty bunch (and is very different from an empty set).

Rather than using Hehner’s bunch theory directly, we extend the set theory of the B-Method to include bunches.

* Corresponding author.
E-mail addresses: w.j.stoddart@gmail.com (W. Stoddart), stevedunne47@gmail.com (S. Dunne), chunyan.mu@gmail.com (C. Mu), frank.zeyda@gmail.com (F. Zeyda).

https://doi.org/10.1016/j.jlamp.2024.100977
Received 10 October 2023; Received in revised form 3 May 2024; Accepted 3 May 2024
Available online 15 May 2024
© 2024 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).
In making changes to a theory we risk introducing inconsistency. B uses a classical first order theory, built by adding the axioms of set theory to a foundation of first order predicate logic (FOPL). It turns out that were we to just add our bunch theory axioms to B’s set theory axioms, we would, indeed, create an inconsistent theory.1

A major aim of the current paper is to establish the consistency of our formulation of bunch theory by providing a denotational model that covers both the inference rules and axioms of our theory.

A surprising claim of the current paper is that our formulation resolves problems associated with partial functions and undefined terms - there will be no undefined terms in our theory, instead terms will be able to express non-determinism.

Our paper is organised as follows. In section 2 we introduce bunch theory. In 3 we revisit the reversible program semantics described in [6]. We show how its laws are derived from those of B’s weakest pre-condition (wp) calculus, show how the use of bunches simplifies its semantics, and give two case studies. In section 4 we discuss the changes to FOPL required to accommodate the mathematical concept of “nothing”, provided by bunch theory, without this leading to inconsistency. We formulate a first-order predicate logic consistent with “nothing” (FOPLN), discuss its positive effect on the information that can be obtained from the application of partial functions, and compare it with Well Defined B [1]. In section 5 we state the axioms of our theory and provide a model. In section 6 we describe extensions to our basic theory, and in section 7 we list some additional properties. In sections 8 and 9 we discuss related work, give our conclusions, and suggest future work.

We provide two supporting documents: Model Validation [24] provides the complete validation of all axioms and inference rules in our theory, thus proving soundness and consistency; Logic proofs in FOPL and FOPLN [23] gives the inference rules for these logics, and by providing proofs for the laws of an extensive logic toolkit, illustrates that (unlike logics that deal with partial functions via multi-valued logic) FOPLN imposes no additional proof burden as compared to FOPL.

Our contributions are the explicit statement of the inference rules and axioms of our version of bunch theory, the provision of a model, an analysis of classical logic that shows where it leads to inconsistency in the presence of bunches, the formulation of a logic suitable for bunch theory, the description of a law for reasoning about partial function applications which dispenses with problems of undefined terms, and a fixed point theorem for bunch theory. We provide a “mini” case study in which we give a new operational interpretation for the useful application of a partial function outside its domain, and we illustrate the highly expressive nature of our backtracking semantics with a second case study that makes use of provisional computations. In response to the report of Morris et al. in [19] on a source of inconsistency in theories of non-deterministic functions, which they discovered in the model building phase of their theory construction, we show B’s model-based approach avoids the problem they describe.

The paper covers logic, bunch and set theory, denotational semantics and program semantics, and there are inevitably a large number of symbols. Details of precedence and associativity are given in an appendix. Abstract syntax is given for symbols where they are defined.

2. Bunch theory

Our theory will give a mathematical meaning to the contents of a set. We write \( \sim A \) (“unpack A”) for the contents of set \( A \). Thus \( \sim \{1, 2\} = 1, 2 \). The comma used here is now a mathematical operator called bunch union rather than just syntax. It is associative and commutative and its precedence is just below that of the standard expression connectives.

We can package any expression by enclosing it in set brackets, thus obtaining a set. Packaging is the inverse of unpacking, so that we have the rules:

\[
\sim \{E\} = E \text{ (} E \text{ any expression)}
\]

\[
\{\sim A\} = A \text{ (} A \text{ a set)}
\]

Bunch union is associated with set union by the rule

\[
\{E\} \cup \{F\} = \{E, F\} \text{ (} E, F \text{ any expressions)}
\]

Whereas we can have a set of sets, we cannot have a bunch of bunches. Bunches are self-flattening. Thus \( \sim \{1, 2\}, \sim \{2, 5\} = 1, 2, 5 \) as may be proved from the above rules.

A bunch \( E \) is a sub-bunch of \( F \) if and only if each element of \( E \) is an element of \( F \). We write this as \( E : F \) (“\( E \) is part of \( F \)”).

Sub-bunches are related to subsets by the rule:

\[
E : F \Leftrightarrow \{E\} \subseteq \{F\}
\]

and bunches \( E \) and \( F \) are equal if \( E : F \) and \( F : E \).

Bunches provide us with a formal conceptualisation of “nothing”, which we call \textit{null} and define as the contents of the empty set:

\[
\textit{null} \equiv \sim \{\}
\]

\textit{null} acts as a unit with respect to bunch union: \( E, \textit{null} = \textit{null}, E = E \)

The guarded bunch \( g \rightarrow E \) is equal to \( E \) where \( g \) is true, and otherwise is \textit{null}. One simple use is to define a conditional expression in terms of more primitive components, i.e.

---

1 See sections 4.3 and 5.1.
if g then E else F end ≡ g → E, ¬g → F

If g holds, the conditional reduces to E. If ¬g holds it reduces to F.

As another example, consider this recursive description of the factorial function, in which the large equals symbol represents a low precedence version of equality.

\[ f(\text{act}) (n) = \begin{cases} n = 0 & \rightarrow 1, \\ n > 0 & \rightarrow n \cdot f(\text{act})(n-1) \end{cases} \]

For negative \( n \) this yields \( f(\text{act}) (n) = \text{null} \), which corresponds to the non-existence of factorials of negative numbers. Non-existence, which is over-deterministic, does not equate to non-termination, which, since we take it as being maximally non-deterministic, is its opposite, and is represented by the improper bunch ∅.

Function application is lifted when applied to a plural bunch, e.g.: \( f(\lnot \{a,b\}) = f(a), f(b) \) and is strict with respect to null: \( f(\text{null}) = \text{null} \).

The pre-conditioned bunch

\[ P \parallel E \equiv P \rightarrow E, \neg P \rightarrow \bot \]

takes the value \( E \) unless \( P \) is false, in which case it becomes maximally non-deterministic.

We define bunch preference \( E \triangleright F \) (pronounced “\( E \) in preference to \( F \)”) which we will use in the definition of a preferential choice in our program semantics. It takes the value \( E \) unless \( E = \text{null} \) in which case it takes the value \( F \):

\[ E \triangleright F \equiv E \equiv F \equiv \text{null} \rightarrow F \]

3. Program semantics for reversible computing and backtracking

3.1. Prospective values

The form \( S \circ E \) expresses, in terms of the current program variables, the value expression \( E \) would take were it to be computed on the modified program variables after execution of program \( S \).

We call this the prospective value of \( E \) after \( S \) and we refer to our semantics as PV semantics. To allow for non-determinism in \( S \), \( S \circ E \) may be a bunch of values.

We give a prospective value calculus for a guarded command language and allow prospective values as part of our expression language. For deterministic \( S \), the term \( S \circ E \) is computed by running \( S \) on a reversible machine,\(^2\) computing and cloning \( E \), and then reversing the computation of \( E \) and \( S \) with all state changing side-effects being undone.

Our expression language allows terms of the form \( \{S \circ E\} \), and here \( S \) may include provisional choices that may be revised by backtracking. Each time a way is found to complete the execution of \( S \), the value of \( E \) in the resulting after-state is calculated and added to the set of results being constructed. Execution is then reversed, so that all possible choices are explored. The topology of this execution is in the form of a tree, and ends when backwards execution finds no more unexplored choices. The value of the term \( \{S \circ E\} \) is then known, and forward execution can recommence without the evaluation of \( \{S \circ E\} \) having caused any enduring change in the program state. Our second case study illustrates the use of such terms.

3.2. Derivation of prospective value primitives from the weakest pre-condition calculus of Abrial’s GSL

Following Abrial we propose a set of semantic primitives for describing program commands. These correspond closely with the primitives used by Abrial in his Guarded Substitution Language (GSL) \(^2\) from which we can derive them. In GSL, \( \{S\}P \) is the weakest pre-condition for \( S \) to establish \( P \). Let \( \langle S \rangle P \) be the weakest pre-condition that \( S \) might establish \( P \). We call this the conjugate weakest pre-condition. Now \( S \) might establish \( P \) if it is not sure to establish \( \neg P \), so WP and CWP are related by \( \langle S \rangle P \leftrightarrow \neg \{S\} \neg P \).

Now since \( S \circ E \) is the bunch of values which expression \( E \) may take after \( S \), then if \( z \) is an arbitrary element, we will have \( z : \{S \circ E\} \) exactly when it is possible that \( S \) could establish \( z : E \) and we therefore have the following basic law linking CWP and PV semantics:

\[ z : \{S \circ E\} \leftrightarrow \langle S \rangle z : E \quad \text{Basic Law} \]

this law enables us to derive the rules for our PV semantics, with the exception of pre-condition (which is not captured in the Basic Law) and preference (which has no equivalent in WP).

We derive the PV rule for bounded choice as an example. Our first step is to derive the CWP law for this construct.

\[ \langle S \cap T \rangle P = \text{“definition of CWP”} \]
\[ \neg \{S \cap T\} \neg P = \text{“by WP for bounded choice”} \]
\[ \neg \{\{S\} \neg P \land \{T\} \neg P\} = \text{“logic, de Morgan”} \]
\[ \neg \{\{S\} \neg P \lor \{T\} \neg P\} = \text{“definition of CWP”} \]
\[ \langle S \rangle P \lor \langle T \rangle P = \text{“definition of CWP”} \]

\(^2\) Hehner has also used this form and writes it as \( S \text{ result } E \).

\(^3\) Our reversible virtual machine WRVM and its documentation is available on Source Forge and is described in \(^29\).
Here \(=\) is Hehner’s large equality, a low precedence symbol used for equational reasoning. For us it represents equality when written between expressions and material equivalence when written between predicates.

Now we can proceed as follows:

\[
\begin{align*}
    z : (S \cap T \circ E) &= \text{“basic law”} \\
    (S \cap T) z : E &= \text{“CWP law for bounded choice”} \\
    (S) z : E \lor (T) z : E &= \text{“basic law”} \\
    z : (S \circ E) \lor z : (T \circ E) &= \text{“bunch theory”} \\
    z : (S \circ E), (T \circ E) &= \text{“and hence”} \\
    S \cap T \circ E &= (S \circ E), (T \circ E)
\end{align*}
\]

3.3. Prospective value semantic primitives

We now give our PV rules for the basic abstract syntax forms:

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax: (F)</th>
<th>Semantics: (F \circ E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>\texttt{skip}</td>
<td>(E)</td>
</tr>
<tr>
<td>assignment</td>
<td>(x := F)</td>
<td>((\lambda x \cdot E)F)</td>
</tr>
<tr>
<td>precondition</td>
<td>(P \mid S)</td>
<td>(P \mid (S \circ E))</td>
</tr>
<tr>
<td>bounded choice</td>
<td>(S\ \cap T)</td>
<td>((S \circ E), (T \circ E))</td>
</tr>
<tr>
<td>preference</td>
<td>(S \rangle T)</td>
<td>((S \circ E) \triangleright (T \circ E))</td>
</tr>
<tr>
<td>guard</td>
<td>(P \Longrightarrow S)</td>
<td>(P \rightarrow (S \circ E))</td>
</tr>
<tr>
<td>seq comp</td>
<td>(S : T)</td>
<td>(S \circ T \circ E)</td>
</tr>
</tbody>
</table>

The ascending order of precedence of symbols used in the above table is \(\circ := \Longrightarrow \mid \rangle \triangleright \rightarrow\), with \(\circ\) being right associative.

The use of bunch theory allows us to unify deterministic and non-deterministic cases. If we use sets instead of bunches to record the possible values of \(E\) after \(S\), then \(S \circ E\) would be a set and \(S \circ T \circ E\) a set of sets. The rules for \texttt{skip} and \texttt{seq comp} would then need to be formulated as follows, with generalised union being used to remove the build up of unwanted structure:

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax: (F)</th>
<th>Semantics: (F \circ E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>\texttt{skip}</td>
<td>{(E)}</td>
</tr>
<tr>
<td>seq comp</td>
<td>(S : T)</td>
<td>(\bigcup (S \circ T \circ E))</td>
</tr>
</tbody>
</table>

Returning to our actual calculus, we give some simple examples of prospective values. First we look at assignment.

\[
\begin{align*}
    x := x + 5 \circ 2 \cdot x + y &= \text{“assignment rule”} \\
    (\lambda x \cdot 2 \cdot x + y)(x + 5) &= 2 \cdot (x + 5) + y
\end{align*}
\]

Our next example looks at sequential composition, where to find \(S \circ T \circ E\) our first step is to find \(T \circ E\). This order of evaluation might seem counterintuitive until we recall that the same reverse evaluation takes place in WP analysis.

\[
\begin{align*}
    x := 1 : x := x + 2 \circ x + 3 &= \text{“seq comp”} \\
    x := 1 \circ x := x + 2 \circ x + 3 &= \text{“assignment”} \\
    x := 1 \circ (\lambda x \cdot x + 3)(x + 2) &= \text{“substitution”} \\
    x := 1 \circ (x + 2) + 3 &= \text{“assignment”} \\
    (\lambda x \cdot (x + 2) + 3)(1) &= \text{“substitution and arithmetic”}
\end{align*}
\]

The precondition of a command is part of the “instructions on the box”. If a command is used when its precondition is false, we assume it is maximally non-deterministic.

Bounded choice is used in specification to represent non-determinism that will be removed during refinement, and in program code to represent choice that may be revised by backtracking.

With Abrial’s GSL, the combination of guards and choice is used to express the semantics of conditional expressions and loops. For example:

\[
\text{if } g \text{ then } S \text{ else } T \text{ end} \equiv g \Longrightarrow S \cap \neg g \Longrightarrow T
\]

Hehner has used guards and choice as control primitives since 1976, initially without being able to get them accepted for publication. Fortunately, the proposal was seen and adopted by Ralph Back, “culminating in his wonderful book The Refinement Calculus [5]”.

In bGSL, in addition to selection and iteration, we use guards and choice to describe backtracking. We use the naked guarded command \(g \Longrightarrow S\) with the following operational interpretation: if \(g\) evaluates to \texttt{true}, execute \(S\), otherwise reverse computation back to the most recent choice which has an untried alternative, and recommence forward execution with an untried alternative.

\[\text{Footnote:} \quad \text{Quoted from [11], Henner’s invited presentation at the First International Symposium on Unifying Theories of Programming. [7]}\]
The following example demonstrates the use of assignment, choice, guards and sequential composition. We have a choice of assigning 1 or 2 to \( x \), but the choice is controlled by a subsequence guard.

\[
\begin{align*}
x & := 1 \land x := 2 ; x = 2 \rightarrow \text{skip} \circ 10 \ast x = \text{"seq comp"} \\
x & := 1 \land x := 2 \circ x = 2 \Rightarrow \text{skip} \circ 10 \ast x = \text{"guard and skip"} \\
x & := 1 \land x := 2 \circ x = 2 \rightarrow 10 \ast x = \text{"bounded choice"} \\
(x & := 1 \circ x = 2 \rightarrow 10 \ast x), (x := 2 \circ x = 2 \rightarrow 10 \ast x) = \text{"assignment"} \\
1 = 2 \rightarrow 10, 2 = 2 \rightarrow 20 = \text{"evaluation of bunch guards"}
\end{align*}
\]

Where \( E \) is a bunch, assigning a non-deterministic choice from \( E \) to \( x \) is expressed by \( x := E \). We do not have a bunch data type in our programming language, but bunches of values arise as the result of non-deterministic computations.

Our table of commands does not include choice from a set, which we use in our second case study. We can define it as:

\[
x \in S \stackrel{\Delta}{=} x := \sim S
\]

In Abrial’s GSL, there is a specific command to express unbounded choice, but in bGSL we can express this with assignment, e.g. \( x := \sim \mathbb{N} \circ 2 \ast x + 1 \) is the bunch of all odd natural numbers.

A refinement calculus for our theory is fully described in [6] and here we limit ourselves to some brief remarks. During the refinement of a specification to executable code, non-deterministic choice represents programmer’s choice which is removed as the programmer interprets the specification and produces code. We have the following definition for the operational refinement of \( S \) by \( T \), which simply describes this reduction of non-determinism.

\[
S \subseteq T \stackrel{\Delta}{=} (\mathcal{S} \circ E) : (S \circ E) \text{ for any expression } E.
\]

The definition of refinement given above allows over refinement to the point of infeasibility, e.g. any program \( S \) can be refined by \( \text{false} \Rightarrow \text{skip} \). To guard against this possibility, in [6] we discuss “star refinement”, which includes a feasibility caveat. We define a program as feasible where cannot backtrack from an abortive situation.

\[
\begin{align*}
\text{fis}(S) & \stackrel{\Delta}{=} (S \circ \bot) = \bot \\
S \subseteq T & \Rightarrow S \subseteq T \land \text{fis}(S) \Rightarrow \text{fis}(T)
\end{align*}
\]

Star refinement is required at the top level, but standard refinement of components is acceptable. The definitions can be used in combination, as in the following easily proved law, which eliminates one proof obligation:

\[
S \cap T \subseteq S' \cap T' \Rightarrow S \subseteq S' \land T \subseteq T' \land (S \cap T) \subseteq (S' \cap T')
\]

Both star refinement and preferential choice introduce non-monotonicity into our refinement calculus. In [6] we discuss how this may be handled and how program continuations are defined and used to analyse preferential choice.

### 3.4. Functions and operations

In bGSL (i.e. in our variant of Abrial’s GSL) we incorporate operations, which have the form \( y \leftarrow \text{Op}(x) \stackrel{\Delta}{=} P | S \). The command \( S \) may assign to the output variable(s) \( y \) of \( \text{Op} \) and reference the input variable(s) \( x \) of \( \text{Op} \). \( P \) is the operation precondition, and is part of the instructions for using \( \text{Op} \). The precondition helps implementers by relieving them of any responsibility for what happens if the operation is invoked when the precondition does not hold. Operations are defined within “abstract machines” which encapsulate state, and may access and change this state. Functions may be invoked by operations, and in a B development there would be a proof obligation to show that the precondition of an operation ensures that any function invoked by that operation is applied only to elements within its domain. This leaves us free to describe functions in a purely mathematical way, as we did with the factorial function in section 2.

With bGSL we have an additional possibility that would permit application of a function outside its domain. We first note that attempting to assign \( \text{null} \) to a variable has the same semantic effect as attempting to execute a naked guarded operation with a false guard, as we see from the following:

\[
\text{false} \Rightarrow \text{skip} \circ E = \text{null} \text{ and } x := \text{null} \circ E = \text{null}
\]

We see from this that the operational interpretation of an attempt to assign \( \text{null} \) to a variable would be to trigger backtracking.

Now consider a function whose domain is not directly known. Our example will be a perfect natural number square root function \( \text{prroo} \); yielding positive integer square roots where they exist, or otherwise \( \text{null} \), so we have \( \text{prroo}(0) = 0 \), \( \text{prroo}(1) = 1 \), \( \text{prroo}(2) = \text{null} \), \( \text{prroo}(3) = \text{null} \), \( \text{prroo}(4) = 2 \ldots \).

Now whereas we might reasonably require protection from attempting to calculate the perfect square root of a negative integer, attempting to calculate it for any \( n \geq 0 \) can yield useful information, since we can code our program to behave accordingly when the perfect square root does not exist. For example the following expression, which is our first case study, yields a set of Pythagorean triples with perpendicular sides less than \( L \).

\[
\{ a : \in 1..L \circ b : \in a + 1..L \circ c := \text{prroo}(a^2 + b^2) \circ \{ a, b, c \} \}
\]
Here we repeatedly choose values for \( a \) and \( b \), then attempt to calculate the perfect square root of \( a^2 + b^2 \) and assign it to \( c \). Where this square root exists, we add \([a, b, c]\) to our set of Pythagorean triples, otherwise we backtrack and try \( a \) with a different value of \( b \), or if all values for \( b \) have been exhausted we try with new values for both \( a \) and \( b \). Computation of the expression terminates when all combinations of values that can be chosen for \( a \) and \( b \) have been tried.

### 3.5. The knights tour search heuristic

In the Knight’s Tour problem, our task is to construct a series of moves for a knight on a chess board so that after 64 moves the knight has completed a tour which visits every square on the board, and has returned to the original square. With a backtracking language this can be achieved by programming a loop which at every iteration makes a provisional move to an unvisited square, backtracks if no such square is available, and terminates when it returns to the starting square. Eventually such an approach will find a tour, but its efficiency can be greatly increased by employing the “Wansdorf Heuristic” which requires us to choose moves that take us to the most tightly constrained squares, on the grounds that these are the squares most at risk of becoming dead ends. The implementation of this heuristic will comprise our second case study. We assume a global integer program variable \( \textit{square} \) which holds the current position of the knight, and an operation \( \textit{Move} \) which will make a provisional choice of a move to a new, previously unvisited square. The set of all possible squares the knight can move to is given by the term \( \{ \textit{Move} \circ \textit{square} \} \). The size of this set tells us how constrained the current square is. To implement our heuristic we construct a relation between the possible squares available for the knight to move to, and how constrained each of these squares is, and from this we obtain the set of most tightly constrained squares available for our next move.

// Lines starting with // are comments
// available is the set of squares we can move to
available := [move \circ \textit{square}] : card(available) > 0 \Rightarrow skip ;
// We create a function mapping available squares to the number of
// onward moves that can be made from each square
constraints := \{ \textit{square} : \textit{available} \circ \textit{square} \mapsto card(\{\textit{move} \circ \textit{square}\}) \} ;
// \textit{m} is the least number of onward moves that can be made
// from an available square
m := minset(range(constraints)) ;
// we use range restriction \( \triangleright \) to limit our function to squares having \textit{m}
// onward moves, then take its domain.
// \textit{tight} is the set of squares that have a minimal number of onward moves
tight := domain(constraints \triangleright \{m\}) ;

### 3.6. Probabilistic choice

We discuss one further command which is particularly interesting for demonstrating the descriptive power of bunch notation. For a possibly probabilistic program \( S \) and expression \( X \), the value of \( S \circ X \) will follow a probability distribution and we will be interested in its “expected value” \[9\], which we express as \( S \circ E \ X \). Where \( S \) does not involve a random calculation, \( S \circ E \ X = S \circ X \). To introduce randomness we formulate a probabilistic choice between programs, where for feasible continuations, \( S \mathop{\oplus} T \) will choose execution of \( S \) with probability \( p \) and \( T \) with probability \( 1-p \). Infeasible continuations provoke backtracking.

We define a weighted addition of values:

\[ E_p + F = \begin{cases} E = \text{null} & \rightarrow F, \ F = \text{null} & \rightarrow E, \ p \ast E + (1-p) \ast F. \end{cases} \]

Then for \( 0 < p < 1 \) we define the expected value of some expression \( X \) (of suitable type) after executing such a choice as\(^{5}\):

\[ S_p \mathop{\oplus} T \circ E \ X = (S \circ E \ X)_{p} + (T \circ E \ X) \]

Where \( S \) and/or \( T \) involve non-determinism their expectations will be a bunch of values, and the bunch lifting of \( p \) is all that is needed to integrate probabilistic and non-deterministic choice, which, as reported in \[30,25\], causes extensive difficulties in other approaches.

As noted in \[6\] and \[25\] we can construct Galois connections between the lattice of wp programs ordered by refinement and the lattices of programs which may also include preferential and probabilistic choice.

---

\(^{5}\) In \[25\] we use the notation \( E(S \circ X) \) for the expected value of \( X \) after executing \( S \). That notation resembles the \( E(X) \) of probability theory, but is not compositional, so we subsequently changed it to the notation used here.
4. Logic and partial functions in bunch theory

4.1. The set theory and underlying logic of B

In our approach we construct a theory of bunches by extending the typed set theory of B with additional axioms specific to bunches. Before we show how we define such extensions and validate them with a denotational semantics, we recall the set theory and underlying logic of B.

The mathematical foundations of B are a classical first-order theory. That is, they consist of the inference rules of first-order logic with equality, (FOPL) plus the axioms which are proper to the theory, these being the axioms of a typed set theory.

For the set theory used with B, elements are either primitive, ordered pairs, or sets. The existence of an arbitrary finite number of “given” sets is assumed, and these contain only primitive elements. These given sets include the set of natural numbers (or alternatively the set of integers). Other given sets may be declared as required. All given sets are maximal.

From the given sets we may construct additional maximal sets using cross product \( S \times T \), and power set \( \mathcal{P}(S) \).

\( S \times T \) is the set of all ordered pairs \( x \mapsto y \) that can be formed by taking an element \( x \) from \( S \) and an element \( y \) from \( T \). \( \mathcal{P}(S) \) is the set of all subsets of \( S \).

Following B, every element is a member of exactly one maximal set, which is its type. We express type information in terms of set membership, thus if \( X \) is a maximal set we can write \( x \in X \) to say \( x \) is of type \( X \). We also allow type information to be expressed implicitly.

We can construct additional sets using set comprehension. \( \{ x \mid x \in S \land P \} \) is the subset of \( S \) consisting of those elements which satisfy \( P \). Here \( S \) might be the type of \( x \), or some subset of its type.

Relations and functions are sets of ordered pairs. \( S \times T \) is the maximal relation between \( S \) and \( T \). We define \( S \leftrightarrow T \equiv \mathcal{P}(S \times T) \) which is the set of all relations between \( S \) and \( T \). We write \( S \rightarrow T \) for the set of partial functions from \( S \) to \( T \), and \( S \rightarrow T \) for the set of total functions from \( S \) to \( T \), so that \( S \rightarrow T \subseteq S \leftrightarrow T \subseteq S \rightarrow T \).

4.2. The Prize Law

FOPL is widely considered to be the standard logic for mathematical theories. For a full expression of this view see [8]. However, when used in formalisms that make extensive use of partial functions, FOPL gives us theories that are less than ideal. To illustrate the main inconvenience we face, suppose we have a partial function king_{of}. From the proposition king_{of}(Norway) = Harald we would like to infer that Norway has a king, and that king is Harald. More generally, we would very much like to have the following, which we will refer to as “the Prize Law”:

\[
f \in A \rightarrow B \land f(a) = b \Rightarrow a \mapsto b \in f
\]

However, we cannot add this law to a classical first-order formulation of set theory without introducing an inconsistency, which arises as follows:

In FOPL we have the existential introduction law: from a premiss \( P[E/x] \) we can conclude \( \exists x \bullet P \). Here, \( P[E/x] \) represents \( P \) rewritten with \( E \) replacing \( x \).

FOPL also gives us an equality axiom, \( E = E \), i.e. any expression equals itself.

Now \((E = x)[E/x] = E = E \), which holds by the equality law. Taking \((E = x)[E/x] \) as our premiss for the existential introduction law we obtain \( \exists x \bullet x = E \). We must draw the conclusion that every expression represents an element. A famous example, discussed in [13], is \( 1/0 \), about which the authors say “it never matters what the value of \( 1/0 \) is”.

Now let \( g \in \mathbb{N} \mapsto \mathbb{N} \land g = \{ \} \). So \( g \) is a function that contains no maplets. Nevertheless by the above argument \( g(1) \) will represent an element. Now we attempt to apply the Prize Law with \( g(1) \) as \( A \), \( \mathbb{N} \) as \( B \), \( g \) as \( f \), 1 as \( a \), and \( g(1) \) as \( b \). We obtain

\[
g \in \mathbb{N} \mapsto \mathbb{N} \land g(1) \equiv g(1) \Rightarrow 1 \mapsto g(1) \in g
\]

The LHS of this implication holds, and since \( 1 \mapsto g(1) \) represents an element and \( g \) was the empty set, we have proved that an element belongs to the empty set. The contradiction stems from our introduction of the Prize Law, and we must conclude that it does not in general hold.

In B, the rule for partial functions must include information about the domain of the function, and may be formulated as:

\[
f \in A \rightarrow B \land a \in dom(f) \land f(a) = b \Rightarrow a \mapsto b \in f
\]

Our inability to use the Prize Law in B is counterintuitive which can lead to errors, but we can impose mechanical checks on our texts to guard against this. It also prevents us constructing the domain of a function from its applications. Such considerations led Cliff Jones and his associates to question the suitability of classical logic as a vehicle for working with partial functions [14,15] and to construct a three-valued logic of partial functions (LPF) [3] which solves problems relating to the application of partial functions, but complicates the use of resolution provers and proof by contradiction [17]. The Rodin project found proponents of classical logic and LPF working together. They adopted an approach investigated in the thesis of Lilian Burdy [4] which used a three valued logic together with proof obligations that can be discharged to ensure well-definedness of the specification, enabling the subsequent use of classical logic. This technique is mechanised in the Rodin project. However, bunch theory allows them to be addressed directly at the level of the underlying logic. To illustrate the difference, we quote the following from [1]:

W. Stoddart, S. Dunne, C. Mu et al.
“The inference rules built into the Rodin Prover therefore preserve well-definedness. They are similar to standard predicate calculus rules, except that they require the user to additionally prove the well-definedness of all predicates or expressions that they introduce, for instance when adding a lemma, or instantiating a universally quantified hypothesis. As an example, when a user wishes to add the lemma “3/x = y” as a hypothesis to a proof, they need to prove not only the lemma itself, but also its well-definedness predicate x ≠ 0.”

In the following section we show that with bunch theory we have the Prize Law. Thus the condition x ≠ 0 would not be needed to make the lemma well defined, but is rather a consequence of the lemma.

4.3. With bunch theory, we have access to the Prize Law

When adding axioms for bunch theory to the set theory axioms of B, we were to retain FOPL we would be able to prove ∃ x • x = null, whereas null is our representation of nothing and certainly not equal to any element. We therefore propose an alternative formalism, FOPLN, which accommodates the concept of nothing. FOPLN has an additional predicate δ(E) which is true if and only if E is an element. The existential introduction rule of FOPLN has two premises, δ(E) and P[E/x], to enable the conclusion ∃ x • P.

We now show that in our new formalism, which is set theory with additional axioms for bunches and a logical underpinning provided by FOPLN, we can use the Prize Law for function application. We first define relational image, which, as in B, will be used in our formulation of function application.

Let R be a relation between types S and T, and A be a subset of S. The relational image of A under R is defined as

$$R[A] \triangleq \{ t \mid t \in T \land (\exists s \in A \land s \mapsto t \in R) \}$$

Now for a function f ∈ S ↦ T, and element s ∈ S and, we define function application f(s) by:

$$f(s) \triangleq \sim f([s])$$

now where a, f are elements and given the premiss of the Prize Law

$$f \in A \Rightarrow B \land f(a) = b$$

we have

$$f(a) \Rightarrow \text{“by definition of function application”}$$

$$\sim f([a]) \Rightarrow \text{“by definition of relational image”}$$

$$\sim \{ t \mid t \in B \land (\exists s \in \{ a \} \land s \mapsto t \in f) \} \Rightarrow \text{“one point rule”}$$

$$\sim \{ t \mid t \in B \land a \mapsto t \in f \}$$

Hence, since we have the premiss f(a) = b

$$\sim \{ t \mid t \in B \land a \mapsto t \in f \} = b$$

So, packaging both sides, by referential transparency we have:

$$\{ t \mid t \in B \land a \mapsto t \in f \} = \{ b \}$$

Hence b ∈ \{ t \mid t \in B \land a \mapsto t \in f \} and by set comprehension:

$$b \in B \land a \mapsto b \in f$$

Thus f(a) = b ⇒ a ↦ b ∈ f and we have derived the Prize Law. We can indeed, state it more succinctly in our formulation without even mentioning the source and target sets: given that a, f are elements, then we have:

$$f(a) = b \Rightarrow a ↦ b \in f \text{ The Prize Law.}$$

4.4. The logic FOPLN as a sequent calculus

In the previous section we noted that the existential introduction rule of FOPL is not suitable for use with bunch theory, and must be given an additional premis. There is a corresponding change in the Y-elim rule, but the remaining rules, including those for propositional logic, are the same as for FOPL. So, unlike LPF, FOPLN has no effect on proof by reductio ad absurdum or proofs using the law of the excluded middle.

To check that the inference rules of FOPLN along with its domain of discourse, provided by bunch theory, result in a sound and consistent formalism, we have constructed a model, which we discuss in section 5.

We present the logics FOPL and FOPLN as sequent calculi. In the sequent Γ ⊢ Δ, the antecedent Γ consists of one or more hypotheses, and the consequent Δ is a single predicate. A sequent asserts that a proof of the consequent can be obtained under the assumption of the hypotheses in the antecedent. An important monotonicity property of our logics is that if we can prove a result
under certain assumptions, adding to those assumptions can never make it impossible for us to prove that result, so if \( \Gamma \vdash \Delta \) holds so does \( \Gamma, \Gamma' \vdash \Delta \). Thus \( \Gamma, \Gamma' \vdash \Delta \) is weaker than \( \Gamma \vdash \Delta \).

Our inference rules consist of one or more sequents as premises and a single sequent as a conclusion. A rule tells us that if the premises hold we have a proof of the conclusion.

Now consider the two laws which differ between FOPL and FOPLN.

The \( \exists \)-intro rule of FOPL says that if a proposition \( P \) is true when \( x \) is replaced by some expression \( E \), then we can say that there exists some \( x \) for which \( P \) is true. The \( \exists \)-intro rule for FOPL is:

\[
\begin{align*}
\text{HYP} \vdash P[E/x] & \quad \exists \text{-intro} \\
\text{HYP} \vdash \exists x \cdot P
\end{align*}
\]

However, once we are dealing with bunches an expression may represent null, and \( P[E/x] \) may then be vacuously true,\(^6\) and provide no evidence that some \( x \) exists such that \( P \) is true. The \( \exists \)-intro rule of FOPLN has an additional premiss that the expression we are citing represents an element, thus it is weaker than the \( \exists \)-intro law of FOPL as it requires an additional premiss to establish the same conclusion. The \( \exists \)-intro rule for FOPLN is:

\[
\begin{align*}
\text{HYP} \vdash \delta(E), \; \text{HYP} \vdash P[E/x] & \quad \exists \text{-intro} \\
\text{HYP} \vdash \exists x \cdot P
\end{align*}
\]

The \( \forall \)-elim rule of FOPL says that if a proposition \( P \) is true for all values of \( x \), then \( P[E/x] \) is true for an arbitrary expression \( E \):

\[
\begin{align*}
\text{HYP} \vdash \forall x \cdot P & \quad \forall \text{-elim} \\
\text{HYP} \vdash P[E/x]
\end{align*}
\]

For FOPLN we have the additional premiss that \( E \) denotes an element, once again, that means this law is weaker than the corresponding law of FOPL:

\[
\begin{align*}
\text{HYP} \vdash \forall x \cdot P, \; \text{HYP} \vdash \delta(E) & \quad \forall \text{-elim} \\
\text{HYP} \vdash P[E/x]
\end{align*}
\]

In FOPLN we have an additional axiom that says any variable is an element.

\[
\text{HYP} \vdash \delta(v) \quad \text{a variable} \quad \text{VAR}
\]

If the expression \( E \) in \( P[E/x] \) is a variable, it will be axiomatic that \( \delta(E) \) holds. This gives us the following derived laws:

\[
\begin{align*}
\text{HYP} \vdash P[v/x] & \quad \text{a variable} \quad \exists \text{-intro-mu} \\
\text{HYP} \vdash \exists x \cdot P \\
\text{HYP} \vdash P[v/x] & \quad \text{a variable} \quad \forall \text{-elim-mu}
\end{align*}
\]

With the help of these two derived laws it is a simple matter to convert proofs within logic from FOPL to FOPLN. In the supporting document Logic proofs in FOPL and FOPLN we demonstrate this for the laws from an extensive logic toolkit, with only one-point laws requiring an extra definedness condition.

We also note that FOPLN is a weakening of FOPL: the laws for \( \exists \)-intro and \( \forall \)-elim require an extra premiss, and the additional axiom VAR holding by default in FOPL, which is a logic in which all expressions are elements.

Given that proofs of predicate calculus equivalence laws involve bound variables, and that bound variables in FOPLN range over elements, it is not surprising that the equivalence laws of FOPL are also provable in FOPLN.

Below we give a proof of the DeMorgan law \( (\exists x \cdot \neg P) \Leftrightarrow (\exists x \cdot \neg P) \) in both logics.

For RHS \( \Rightarrow \) LHS, i.e. \( \neg(\forall x \cdot P) \Rightarrow (\exists x \cdot \neg P) \) we have in FOPL:

1. \( \neg(\forall x \cdot P) \) 
   assumption
2. \( P[a/x] \) 
   assumption
3. \( \forall x \cdot P \) 
   2,\( \forall \)-intro
4. \( \neg P[a/x] \) 
   1,3,contradiction
5. \( \exists x \cdot \neg P \) 
   4,\( \exists \)-intro
6. \( \neg(\forall x \cdot P) \Rightarrow (\exists x \cdot \neg P) \) 
   5,deduction

and in FOPLN:

1. \( \neg(\forall x \cdot P) \) 
   assumption
2. \( P[a/x] \) 
   assumption
3. \( \forall x \cdot P \) 
   2,\( \forall \)-intro
4. \( \neg P[a/x] \) 
   1,3,contradiction
5. \( \exists x \cdot \neg P \) 
   4,\( \exists \)-intro-mu
6. \( \neg(\forall x \cdot P) \Rightarrow (\exists x \cdot \neg P) \) 
   5,deduction

\(^6\) For example we will see from the lifted definition of membership given in section 6.1 that \( \text{null} \in \{ \} \) is true.
In our proofs we indent after each assumption and outdent where that assumption is discharged. For example, in both proofs, under the assumption of \( \neg (\forall x \cdot P) \) at line 1, we are able to prove \( (\exists x \cdot \neg P) \) at line 5 and thus by deduction (use of the DED rule which discharges the assumption of line 1) \( \neg (\forall x \cdot P) \Rightarrow (\exists x \cdot \neg P) \) at line 6. We refer the reader to [23] for a full description of the inference rules of these logics including the rules we use for propositional logic, which they have in common. That document also presents these proofs in tree form.

The only difference between the proofs occurs at the \( \exists \)-intro step. Here the FOPLN proof requires the proposed witness \( a \) to satisfy \( \delta(a) \), which holds by the VAR axiom of FOPLN since \( a \) is a variable. These details are not seen in the proof because they are encapsulated within the derived rule \( \exists \)-intro-mu.

For the proof of \( \text{LHS} \Rightarrow \text{RHS} \) we have in FOPL:

1. \( \exists x \cdot \neg P \) assumption
2. \( \neg P[a/x] \) assumption
3. \( \forall x \cdot P \) assumption
4. \( P[a/x] \) 3, \( \forall \)-elim
5. \( \neg (\forall x \cdot P) \) 2, 4, contradiction
6. \( \neg P[a/x] \Rightarrow \neg (\forall x \cdot P) \) 5, deduction
7. \( \neg (\forall x \cdot P) \) 1, 6, \( \exists \)-elim
8. \( (\exists x \cdot \neg P) \Rightarrow \neg (\forall x \cdot P) \) 7, deduction

And in FOPLN:

1. \( \exists x \cdot \neg P \) assumption
2. \( \neg P[a/x] \) assumption
3. \( \forall x \cdot P \) assumption
4. \( P[a/x] \) 3, \( \forall \)-elim-mu
5. \( \neg (\forall x \cdot P) \) 2, 4, contradiction
6. \( \neg P[a/x] \Rightarrow \neg (\forall x \cdot P) \) 5, deduction
7. \( \neg (\forall x \cdot P) \) 1, 6, \( \exists \)-elim
8. \( (\exists x \cdot \neg P) \Rightarrow \neg (\forall x \cdot P) \) 7, deduction

Here the only difference in the proofs is the use of the derived rule \( \forall \)-elim-mu in the FOPLN proof in place of the \( \forall \)-elim rule in the FOPL proof.

5. A denotational semantics of bunch theory

We introduce a denotational model with a compositional semantics to demonstrate that our bunch theory and its associated first-order logic are sound and consistent.

5.1. The axioms of our theory

We take Abrial’s axiomatisation of set theory, as given in the B-Book, and add the axioms required for bunch theory.

Our set theory axioms are:

Name | Rule | Condition
--- | --- | ---
Ordered pair | \( E \mapsto F \in (s \times t) \iff E \in s \land F \in t \) | |
Powerset | \( s \in P(t) \iff (\forall x \cdot x \in s \Rightarrow x \in t) \) \( x \setminus (s, t) \) | |
Comprehension | \( E \in \{ x | x \in s \land P \} \iff E \in s \land P[E/x] \) \( x \setminus s \) | |
Set equality | \( (\forall x \cdot x \in s \iff x \in t) \Rightarrow s = t \) \( x \setminus (s, t) \) | |
Choice | \( (\exists x \cdot x \in s) \Rightarrow (\exists x \cdot x \in s \land x = \text{choice}(s)) \) \( x \setminus s \) | |

In the B-Book the axiom of choice is written as: \( \exists x \cdot x \in s \Rightarrow \text{choice}(s) \in s \). Once bunches are introduced this leaves open the possibility that \( \text{choice}(s) = \text{null} \) with \( \text{null} \in s \) being vacuously true. Our modified version ensures that \( \text{choice}(s) \) is an element whenever \( s \) is non-empty.

We inherit the whole of Abrial’s set theory into our bunch theory but with FOPLN as its underlying logic and function application defined as in section 4.

We follow the general mathematical convention, also used by Abrial, by which variables are implicitly universally quantified at the top level.

The additional axioms required for bunch theory are:

Name | Rule | Condition
--- | --- | ---
Packaging | \( \{ \neg A \} = A \) | \( A \) a set
Unpacking | \( \neg \{ E \} = E \) | |
Empty choice | \( \text{choice}(\{ \}) = \{ \} \) | |
Element | \( \delta(E) \iff \neg (\{ E \} = \{ \}) \land E = \text{choice}(\{ E \}) \) | |
Guard 1 | \( g \Rightarrow g \mapsto E = E \) | |
Guard 2 | \( \neg g \Rightarrow g \mapsto E = \{ \} \) | |
In these axioms, \( \sim \{ \} \) is used in place of \textit{null}, which has not yet been defined.

It is also axiomatic that variables are elements but that axiom is part of our logic so is not stated here.

The empty choice axiom is to avoid the undefined term that otherwise arises from the expression \( \text{choice}(\{1\}) \).

In our theory the Element axiom is obviously equivalent to \( \delta(E) \Leftrightarrow \text{card}(\{E\}) = 1 \). However, because use of \textit{card} involves function application whereas \textit{choice}(s) is a semantic primitive, validating the axiom in our model is simpler with the form we have given.

5.2. Model construction

Abrial’s typed set theory will act as our model language, i.e. the language in which we express our model. We refer to it as \( \mathcal{M} \), and to the bunch theory language we are validating with our model as \( \mathcal{B}_0 \). This latter will be an impoverished version of bunch theory, lacking vital operations such as bunch union and vital predicates such as bunch inclusion. However it will have two virtues: its simplicity will simplify its validation, and it will be possible to extend it to obtain an enriched language \( \mathcal{B} \) which is our full theory. \( \mathcal{M} \) is a sub-language of \( \mathcal{B}_0 \) which is a sub language of \( \mathcal{B} \).

To express our semantics we use denotational brackets \([..]\) to enclose each abstract syntax form from \( \mathcal{B}_0 \), and we give rules for the translation of each form to a matching form from \( \mathcal{M} \). The rules of our model are given in Table 1 of section 5.3. Here we provide some indications of what these rules represent and some examples of how they may be derived.

The aim of our model is to enable the translation of any predicate \( P \) from \( \mathcal{B}_0 \) to a predicate \([P]\) in \( \mathcal{M} \) such that \( P \) will be true in \( \mathcal{B}_0 \) exactly when \([P]\) is true in \( \mathcal{M} \). Since \( \mathcal{B}_0 \) is an extension of \( \mathcal{M} \) we can write this property, in \( \mathcal{B}_0 \), as \( P \Leftrightarrow [P] \). We validate the axioms of \( \mathcal{B}_0 \) by translating them to \( \mathcal{M} \) and proving them in set theory.

Before we can translate predicates, we must consider how to translate their constituent expressions. \( \mathcal{B}_0 \) differs from \( \mathcal{M} \) in allowing unpackaged collections, and for each expression \( E \) in \( \mathcal{B}_0 \) there will be some set in \( \mathcal{M} \) that has the values of \( E \) as its contents, and can act as the matching form that corresponds to \( E \). For any expression \( E \) from \( \mathcal{B}_0 \) we write its semantic representation as \([E]\), where \([E] = \{E\}\) holds in \( \mathcal{B}_0 \). We refer this as our “General Rule”.

Where the expression \( E \) is also a well defined expression in \( \mathcal{M} \), this will represent the complete semantic representation. This is the base case of our model.

One idea that is useful in defining more general translations is that of the “semantic facsimile” of an elemental expression. Let \( E \) be an expression from \( \mathcal{B}_0 \) representing an element. Then \([E] = \{E\}\) and \([E] \) will be a singleton set with \( E = \text{choice}(\{E\}) \) and hence \( E = \text{choice}(\{E\}) \). We call the term \textit{choice}([\( E \)]) the semantic facsimile of \( E \), it is a term which represents the same value as \( E \), but expressed in the language \( \mathcal{M} \).

Consider the semantics of unpacking. Recalling that we use \( s \) to represent a set, and by implication an element, the rule for unpacking could be expressed in \( \mathcal{B}_0 \) as \( \sim \{s\} = \sim \{s\} \), but this will not do as a semantic rule for our model because the expression \( \sim \{s\} \) does not exist in \( \mathcal{M} \). Nor can we have the rule \( \sim \{s\} = s \), because not every expression \( s \) that represents a set in \( \mathcal{B}_0 \) exists in \( \mathcal{M} \). However, using the semantic facsimile of \( s \) we can express our rule as:

\[
\sim \{s\} = \text{choice}(\{s\})
\]

For an ordered pair \( E \rightarrow F \), where \( E \) and \( F \) are expressions which may represent non-elemental bunches, the problem is slightly different. The form \( \{E \rightarrow F\} \) exists in set theory, but with \( E \) and \( F \) being elements. Our semantics expresses \( [E \rightarrow F] \) for cases in which \( E \) and \( F \) are general bunches, and our rule is:

\[
[E \rightarrow F] = [E] \times [F]
\]

For the semantics of \textit{choice} we must consider two cases. Choice from an empty set is \textit{null}. In our rule for \textit{choice}(s) we see the denotation of \( s \) rather than its value, and the denotation of the empty set is \( \{\} \), so when \([s] = \{\}\) we have \([\text{choice}(s)] = \{\} \). For the case where \([s] \neq \{\}\) with \( s \) being a set and thus also an element, we have:

\[
[\text{choice}(s)] = \text{“by our general rule”}
\]

\[
[\text{choice}(s)] = \text{“replacing } s \text{ by its semantic facsimile”}
\]

\[
[\text{choice}(\{s\})]
\]

We obtain the semantic rule for \textit{choice}(s) by combining the above cases in a conditional:

\[
[\text{choice}(s)] = \text{ if } [s] = \{\} \text{ then } \{\} \text{ else } [\text{choice}(\{s\})] \text{ end}
\]

Moving on now to predicates, for the semantics of equality, \( E = F \) will be true in \( \mathcal{B}_0 \) when \([E] = [F]\) is true, i.e. when \([E] = [F]\) is true in \( \mathcal{M} \).

In the Element rule of Table 1 in section 5.3 we are saying that \( E \) is an element when its denotation is a singleton set. The apparently simpler way of expressing the singleton property as \( \text{card}(\{E\}) = 1 \) would be insufficient in \( \mathcal{M} \) as it does not rule out the possibility of \([E] \) being an infinite set. We have proved earlier that \( 1/0 \) takes some value when using FOPL, and similarly so does, for example, \( \text{card}(\mathcal{Z}) \), and there is no way we can prove \( \text{card}(\mathcal{Z}) \) is not 1.

Now deriving the semantics of membership:
Table 1
Semantic model.

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax: ( P )</th>
<th>Denotation: ( [P] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Case</td>
<td>( E )</td>
<td>([E] ), ( E ) a well defined expression from ( \mathcal{M} )</td>
</tr>
<tr>
<td>Packaging</td>
<td>( {E} )</td>
<td>([{E}])</td>
</tr>
<tr>
<td>Unpacking</td>
<td>( \sim s \mapsto \text{choice}(s) )</td>
<td>([\text{choice}(s)])</td>
</tr>
<tr>
<td>Maplet</td>
<td>( E \mapsto F )</td>
<td>([E] \times [F])</td>
</tr>
<tr>
<td>Powerset</td>
<td>( \forall(s) { \text{P(choice}(s[x])) } )</td>
<td>([\text{P(choice}(s[x]))])</td>
</tr>
<tr>
<td>Cross Prod</td>
<td>( s \times t { \text{choice}(s) \times \text{choice}(t) } )</td>
<td>([\text{choice}(s) \times \text{choice}(t)])</td>
</tr>
<tr>
<td>Choice</td>
<td>\text{choice}(s) { \text{if } [s] = {{} } \text{then } {} \text{ else } \text{choice(\text{choice}(s))} } } }</td>
<td>([\text{choice(\text{choice}(s))}])</td>
</tr>
<tr>
<td>Element</td>
<td>( \delta(E) { [E] \neq {} \land [E] = {\text{choice}(E)} } } }</td>
<td>([\text{choice}(E)])</td>
</tr>
<tr>
<td>Equals</td>
<td>( E \equiv F { [E] = [F] } } }</td>
<td>([F])</td>
</tr>
<tr>
<td>Member</td>
<td>( E \in s { [E] \subseteq \text{choice}(s) } } }</td>
<td>([\text{choice}(s)])</td>
</tr>
<tr>
<td>Not</td>
<td>( \neg P { [P] } } }</td>
<td>([\neg P])</td>
</tr>
<tr>
<td>And</td>
<td>( P \land Q { [P] \land [Q] } } }</td>
<td>([P] [Q])</td>
</tr>
<tr>
<td>Or</td>
<td>( P \lor Q { [P] \lor [Q] } } }</td>
<td>([P] [Q])</td>
</tr>
<tr>
<td>Set</td>
<td>( { x \mid P } { { x \mid [P] } } }</td>
<td>([{ x \mid [P] }])</td>
</tr>
<tr>
<td>ForAll</td>
<td>( \forall x \cdot P { \forall x \cdot [P] } } }</td>
<td>([\forall x \cdot [P]])</td>
</tr>
<tr>
<td>Exists</td>
<td>( \exists x \cdot P { \exists x \cdot [P] } } }</td>
<td>([\exists x \cdot [P]])</td>
</tr>
<tr>
<td>Subst1</td>
<td>( F[E/x] { [F]{\text{choice}(E)}/x } { { \delta(E)} } } }</td>
<td>([\text{choice}(E)])</td>
</tr>
<tr>
<td>Subst2</td>
<td>( P[E/F] { [P]{[E]/[F]} } } }</td>
<td>([\text{choice}(E){[E]/[F]}])</td>
</tr>
<tr>
<td>Subst3</td>
<td>( P[E/x] { [P]{\text{choice}(E)}/x } { { \delta(E)} } } }</td>
<td>([\text{choice}(E)])</td>
</tr>
</tbody>
</table>

\( E \in s = \text{“packaging LHS”} \)
\( \{E\} \subseteq s = \text{“by our General Rule for semantics of expressions”} \)
\( [E] \subseteq s = \text{“replacing s by its semantic facsimile”} \)
\( [E] \subseteq \text{choice}(s) \)

“and thus”
\( [E \in s] \iff [E] \subseteq \text{choice}(s) \)

From the general property of our semantics that \( P \iff [P] \) we obtain our rules for the semantics of quantifiers:
\( [\forall x \cdot P] \iff [\forall x \cdot [P]] \) and \( [\exists x \cdot P] \iff [\exists x \cdot [P]] \)

Our semantics of predicates allows any predicate of \( \mathcal{B}_0 \) to be translated to a corresponding predicate of \( \mathcal{M} \). To validate our theory we perform such translations on our axioms and prove the resulting predicates. An extension of this method allows us to validate the inference rules for quantifiers of our new logic FOPLN.

We call the approach we use “symbolic semantics” since it provides rules to translate each formula in our language to a symbolic representation in our model language, where the environments in which these representations are interpreted are provided implicitly. In the working notes on bunch theory that we posted on arXiv [27] we used a different approach in which environments are provided explicitly by means of bindings, in the style adopted in [19] and [21]. We have changed because validation proofs done with symbolic semantics are simpler. An added benefit is that when, as here, the language to be validated is an extension of the model language, symbolic semantics permits simple derivation of the model’s rules.

5.3. Model

Denotations for the abstract syntax forms \( F \) of our language \( \mathcal{B}_0 \) are given in Table 1 of section 5.3, in which \( x \) is a variable, \( E \) and \( F \) are general expressions from \( \mathcal{B}_0 \), \( s \) and \( t \) are expressions from \( \mathcal{B}_0 \) representing sets, and \( P \) and \( Q \) are predicates from \( \mathcal{B}_0 \).

The model represents bunch theory in terms of set theory because a predicate \( P \) is true in bunch theory if and only if its semantic representation \( [P] \) is true in set theory. Validating a bunch theory predicate \( P \) begins by taking its denotation \( [P] \) and applying the denotational rule that corresponds to the form of \( P \). Let us take \( [\sim 4 = \text{choice(4)}] \) as an example. By the semantics of equality we have:
\( [\sim 4 = \text{choice(4)}] = [\sim 4] = [\text{choice}(4)] \)

On the RHS we now have two expressions in denotational brackets, and the corresponding rules are for unpacking and choice. Applying these rules we obtain:
\( \text{choice}([\text{choice}(4)]) = \{\text{choice}(\text{choice}([\text{choice}(4)]))\} \)

Inspecting the terms remaining within denotational brackets, we apply the semantics of packaging on both sides.
\( \text{choice}(4) = \{\text{choice}(4)\} \)

So far we have just applied denotational rules, but now we can do some housekeeping by taking choice on both sides:
\( [4] = \{\text{choice}(4)\} \)

Now applying our Base Case to the constant 4:
{4} = \{\text{choice}[4]\}

We have now eliminated all denotational brackets and arrived at a set theory predicate, which we simplify by taking choice on the RHS to give:

{4} = \{4\} \quad \text{which holds by the equality axiom of logic.}

During the validation the denotational rules applied are chosen in a completely mechanical way. Text within denotational brackets remains passive apart from application of the model rules.

We finally note two general points:

**Note 1.** In the typed set theory of B, a predicate $P$ in which $x$ occurs free would indicate the type of $x$, e.g. it could have the form $x \in s \land P$ or $x \in s \Rightarrow P$. Where we use the term $[P]$ in the model we can, if required, convert to a typed form by replacing $P$ with a predicate containing explicit type information, and then analysing this typed predicate, for example:

\[
\begin{align*}
[x \in s \land P] &= \text{"by semantics of } \land" \\
[x \in s] \land [P] &= \text{"by Base Case applied to variable } x \text{ and semantics of membership"} \\
\{x\} \subseteq \text{choice}(s[s]) \land [P] &= \text{"set theory"} \\
x \in \text{choice}(s[s]) \land [P]
\end{align*}
\]

**Note 2.** Subst2 is required to validate the referential transparency rule of our logic. To see intuitively that it holds, note that our denotational semantics is compositional: i.e. the semantics of any bunch-theory formula is entirely determined by that of its constituent sub-formulas and terms. So the parse-tree structure of such a formula will be accurately mimicked by that of the set-theory formula which is its interpretation in the model. Thus occurrences of the term $E$ in the parse tree of formula $P$ will be accurately mimicked by the occurrences of $[E]$ in the parse tree of $[P]$. We can therefore replace all those occurrences of $E$ by $F$ at the outset and then semantically interpret the resulting formula, or we can interpret the original formula $P$ and then replace all the occurrences of $[E]$ in the resulting set-theory parse tree by $[F]$. For a more mathematical argument and a proof that the rule Subst3 can be derived from Subst2 see the supplementary document ModelValidation.

### 5.4. The semantics of logical inference rules

As detailed in section 4.4, our logic is presented as a sequent calculus. In the sequent $\Gamma \vdash \Delta$, the antecedent $\Gamma$ consists of one or more hypotheses, and the consequent $\Delta$ consists of a single hypothesis. A sequent asserts that a proof of the consequent hypothesis can be obtained under the assumption of the hypothesis in the antecedent. To validate we must establish the corresponding semantic entailment $\Gamma \models \Delta$, which we do by showing $[\Gamma] \Rightarrow [\Delta]$.

In our presentation an inference rule consists of one or more sequents which are the premises of the rule, and a single sequent which is the conclusion of the rule. To validate we show the validity of the premises implies the validity of conclusion.

### 5.5. Models, validity, soundness, consistency, and completeness

A formal model provides a complete representation of a theory in terms of another, generally more established, theory. Our model of bunch theory provides a complete representation of bunch theory in terms of set theory.

Having a model allows us to talk about the validity of propositions in a theory. A proposition $P$ is valid if it is possible to prove its representation in the model. In our formulation, a proposition $P$ of our theory is "valid" if $[P]$ can be proved in $\mathcal{M}$. Showing that our representation is a model requires us to validate the axioms and inference rules of our theory.

A theory is sound if it allows the proof only of valid propositions. The existence of a model for a theory, along with an assumption that the model language is consistent, implies that the theory is necessarily sound.

A theory is consistent when it is not possible to prove some result along with its negation. The existence of a model for a theory implies its consistency, for were it to be inconsistent and admit proofs of both $P$ and $\neg P$ this would mean both $[P]$ and $[\neg P]$ could be proved in the model language. But by the semantics of negation the second of these is $\neg [P]$, allowing us to prove both $[P]$ and its negation in our model language. This is impossible under the assumption that our model language is consistent.

The complementary property to soundness is completeness. A theory is complete if it allows proof of all valid results. However, Gödel’s Incompleteness Theorem tells us that complex theories are never complete, and this will be the case for our bunch theory: we cannot hope to prove completeness. We may discover the need for a new axiom by identifying a property of our theory that we are unable to prove from our existing axioms. In this case we can use our model to check the validity of any new axiom we may wish to propose.

---

7 Here we are using the semantics of negation from our symbolic semantics, which allows the argument to be presented in a particularly concise way.
5.5.1. Example validations for our theory

To illustrate use of the model in proving soundness we validate two of our bunch theory axioms and the \( \exists \)-intro rule from our logic. A full validation of the theory is provided in the supplementary document Model Validation.

For the packing axiom we have to show that where \( A \) is a set (and hence also an element) the semantic representation of \( \{ \sim A \} \) in our model is equal to that of \( A \).

\[
[[ \sim A ]] = [A]
\]

Taking LHS

\[
[[ \sim A ]] = \text{“by semantics of packaging”}
\]

\[
[[ A ]] = \text{“by semantics of unpacking”}
\]

\[
\text{choice}([[ A ]]) = \text{“since the semantic representation of an element is a singleton set”}
\]

\[
[A] \ \square
\]

For the unpacking axiom we similarly show the semantic representation of \( \sim \{ E \} \) is equal to that of \( E \).

Again taking LHS

\[
[[ \sim \{ E \} ]] = \text{“by semantics of unpacking”}
\]

\[
\text{choice}([[\{ E \} ]]) = \text{“by semantics of packaging”}
\]

\[
\text{choice}([[\{ E \} ]]) = \text{“taking choice”}
\]

\[
[E] \ \square
\]

5.5.2. The validation of an inference rule

For the \( \exists \)-intro rule of FOPLN, our rule is:

\[
\begin{align*}
HYP & \vdash \delta(E), \ HYP \vdash \ P[E/X] \\
HYP & \vdash \exists x \cdot P
\end{align*}
\Rightarrow \exists \text{-intro}
\]

We have to prove that the validity of the premises implies the validity of the conclusion, i.e. \( [[\delta(E)]] \land [[P[E/x]]] \Rightarrow [[\exists x \cdot P]] \).

\[
\begin{align*}
[[\delta(E)]] \land [[P[E/x]]] & \Rightarrow [[\exists x \cdot P]] = \text{“by Exists rule”} \\
[[\delta(E)]] \land [[P[E/x]]] & \Rightarrow (\exists x \cdot [[P]]) = \\
& \text{“by logic } a \land b \Rightarrow c \Leftrightarrow a \Rightarrow (b \Rightarrow c)’’ \\
[\delta(E)] & \Rightarrow ([P[E/x]]) \Rightarrow \exists x \cdot [[P]]
\end{align*}
\]

Which we can discharge by proving \( [[P[E/x]]] \Rightarrow \exists x \cdot [[P]] \) under the assumption \( [[\delta(E)]] \).

Noting that the assumption together with the Subst3 law tells us that \( [P[E/x]] \) is equivalent to \( [P][\text{choice}[E]/x] \) it will be sufficient to prove:

\[
[P][\text{choice}[E]/x] \Rightarrow (\exists x \cdot [[P]])
\]

which follows from the \( \exists \)-intro rule of FOPL. \( \square \)

6. Extending the basic theory

We have given an axiomatisation and model for a minimal theory of bunches, which we now extend.\(^8\)

The descriptions of these new definitions may be explicit or implicit. For example suppose we want to define \( \text{nat} \), the bunch of natural numbers. Since we are building bunch theory by extending set theory we have \( \mathbb{N} \), the set of natural numbers, at our disposal, and we can explicitly define \( \text{nat} \equiv \sim \mathbb{N} \). Such definitions are conservative extensions to our theory that may be eliminated during proof by replacement, and do not affect the validity of our formalism. Alternatively, we could introduce \( \text{nat} \) by characterising it with the recursive equation \( \text{nat} = 0, \text{nat} + 1 \). This form of implicit definition is generally supported by a least fixed point argument.

Finally, an extension could be so radical as to require a revision of our denotational model. This is the case when we introduce the improper bunch.

6.1. Conservative extensions

These definitions will serve two purposes. Firstly we will define various forms not included in our model, namely \( \text{null} \), bunch union and intersection, sub-bunch inclusion, the conditional expression, and bunch comprehension. Secondly we will lift some definitions which currently apply to elements to apply to arbitrary bunch expressions.

\(^8\) This corresponds to the presentation of set theory, where \( U, \cap, \setminus, \subseteq \) etc. are not described by axioms but subsequently added by definitions. We assume the usual set theory extensions.
Our bunch.

We etc.

Since $x$ which

we abbreviate

apply

In this
definition

membership

of $a$ and $b$ which

may

first

right

now

these definitions which can be extended using bunch comprehension. For example if $e$ is an element and a set, the existing definition of unpacking $e$ yields a bunch which consists of the members of $e$. Now we want to define $\sim E$ for the case where $E$ may be a bunch of sets, which we do as follows

$\sim E \triangleq \{ x \mid P \cdot E \}$

We apply this idea of lifting in a quite general way, as in the following re-definitions:

$P(S) \triangleq \{ s \mid s \cdot S \}$

$A \cup B \triangleq \{ a, b \mid a \cdot A \land b \mid B \}$

$a \triangleleft \beta \triangleq \{ a, b \mid a \cdot a \land b \mid \beta \}$

$choice(S) \triangleq \{ s \mid s \cdot S \}$

etc.

We now extend function application, which we discussed in section 4. We first modify our previous definition, simply by removing the restriction that the argument of a function must be an element. That gives us the following definition, in which $E$ can be any bunch. To allow non-deterministic functions we also allow $f$ to be any relation:

$f(E) \triangleq \sim f(\{ E \})$

Our Prize Law still needs the argument of $f$ to be an element, but may now be generalised to $f(a) = E \Rightarrow a \Rightarrow E \in f$, where $E$ need not be an element, and indeed may be null!

We lift our definition of function application to encompass the application of a bunch of functions to a bunch of arguments.

$F(E) \triangleq \{ f \mid f \cdot F \}$

Our semantic model for the formation of ordered pairs already gives us a lifted definition, so no redefinition is needed in this case.

The set membership predicate is lifted, so for bunches $X, S$

$X \in S \triangleq \forall x, s \cdot x : X \land s : S \Rightarrow x \in s$

Thus lifted membership is robust with respect to a reduction of non-determinism by refinement, i.e. $X \in S \land X' : X \land S' : S \Rightarrow X' \in S'$.

The negation of the lifted membership predicate does not have this robustness, but we can define definitive non-membership, which does. For a set $s$ of type $f$ define the complement of $s$ as $s^c \triangleq i \setminus s$ and $x \not\in s \triangleq x \not\in s^c$ Then where $S$ is a bunch of sets, $S^c$ is the bunch of complements of sets from $S$ and $X \not\in S$ is equivalent to $X \in S^c$. 
We define relational predicates, $<, \leq, \subseteq$ etc. in terms of set membership, and these are lifted in the same way. The equality predicate cannot sensibly be lifted since to do so would contradict the equality law $E = E$ which we inherit from our logic.

6.2. Least fixed points

Recursive function definitions are a form of implicit definition and may be defined formally as the least fixed solution of a fixed-point equation on their functional. Since we have set theory at our disposal and functions are defined as sets of ordered pairs, these arguments are available to us unchanged. However, we also want to use recursive equations of bunch expressions such as $\text{nat} = 0, \text{nat} + 1$. Bunches are partially ordered by bunch inclusion, so we wonder if we can take advantage of this in some kind of fixed-point argument. The problem is that classical fixed-point theory is based on partial orders (PO). A PO is a set with a partial order relation, e.g. $(P(S), \subseteq)$. However, we cannot place bunches in a set without them losing their identity, for example $\{\{1, 2\}, \{2, 3\}\} = \{1, 2, 3\}$, and thus we cannot call on classical fixed-point theory directly. However, we will be able to formulate a fixed-point theory for bunches and prove it by reference to the classical theory, which we now recall.

A PO is a complete lattice if each subset of its elements has a least upper bound (lub) and a greatest lower bound (glb) in $S$.

Given a po $(S, \subseteq)$ and a function $f \in S \rightarrow S$ we say $f$ is monotonic if for any $s, s' \in S, s \subseteq s' \Rightarrow f(s) \subseteq f(s')$.

Tarski’s theorem for complete lattices tells us that given a complete lattice $(S, \subseteq)$ a monotonic function $f \in S \rightarrow S$ will have a least fixed point. That is a least value satisfying the equation $x = f(x)$.

$(P(S), \subseteq)$ qualifies as a complete lattice because any set $T \subseteq P(S)$ has lub $\bigcup T$ and glb $\bigcap S$. This allows us to state the following:

Corollary to Tarski’s theorem. A monotonic set transformer $f \in P(S) \rightarrow P(S)$ has a least fixed point.

In bunch theory, function application may be applied to a bunch and may return a bunch of values. However it does not provide a general way of mapping bunches to bunches. For example it is strict wrt null. To obtain a completely general form of bunch transformer we define “wholistic function application” which applies to the whole of a bunch, and which we represent by an infix period which has the same precedence as function application. Where the function $f$ is a set transformer we define:

\[ f.X \equiv f(\{x\}) \]

We say wholistic application of $f$ is monotonic wrt bunch inclusion if $E_1 : E_2 \Rightarrow f.E_1 : f.E_2$.

We introduce two lemmas before stating our fixed-point theorem:

**Lemma 1.** Wholistic application of $f$ is monotonic wrt bunch inclusion exactly when $f$ is a monotonic set transformer.

**Proof.** We begin with the defining property of bunch monotonicity of wholistic application. Retaining equivalence at each step, we convert this to the defining property for monotonicity of a set transformer. In this proof the contents of an arbitrary set $x_i$ are written as $X_i$, i.e. $X_i = \sim x_i$.

\[
\begin{align*}
X_1 : X_2 &\Rightarrow f.X_1 : f.X_2 = \text{"applying definition of wholistic application"} \\
X_1 : X_2 &\Rightarrow \sim f(\{X_1\}) : \sim f(\{X_2\}) = \text{"writing $X_i$ as $\sim x_i$"} \\
\sim x_1 : \sim x_2 &\Rightarrow \sim f(\{x_1\}) : \sim f(\{x_2\}) = \text{"by bunch theory packing axiom"} \\
\sim x_1 : \sim x_2 &\Rightarrow \sim f(x_1) : \sim f(x_2) = \text{"since $E : F \Rightarrow \{E\} \subseteq \{F\}$"} \\
\{x_1\} : \{x_2\} &\Rightarrow \{f(x_1)\} : \{f(x_2)\} = \text{"by bunch theory packing axiom"} \\
x_1 : x_2 &\Rightarrow f(x_1) : f(x_2) \quad \Box
\end{align*}
\]

**Lemma 2.** If $x$ is a fixed point of set transformer $f$ iff $X = \sim x$ is a fixed point for the wholistic application of $f$.

**Proof.**

\[
\begin{align*}
x = f(x) &\Rightarrow \text{"unpacking both sides"} \\
\sim x = \sim f(x) &\Rightarrow \text{"since $X = \sim x$ and $\{X\} = x$"} \\
X = \sim f(\{x\}) &\Rightarrow \text{"by definition of wholistic application"} \\
X = f.X &\quad \Box
\end{align*}
\]

We now state our fixed-point theorem for bunches.

**Theorem 1.** If wholistic application of $f$ is bunch monotonic, then wholistic application of $f$ has a least fixed point, that is there is a minimal bunch $X$ which satisfies $X = f.X$.

**Proof.** By Lemma 1 if wholistic application of $f$ is bunch monotonic then $f$ is monotonic, and by the corollary to Tarski’s theorem given above, $f$ has a least fixed point $x$ (say).
By Lemma 2 $X = \sim x$ is a fixed point for the wholistic application of $f$. We now show it is the least fixed point. Let $x'$ be a fixed point of $f$. Then since $x$ is the least fixed point, $x \subseteq x'$, and since $f$ is monotonic, $f(x) \subseteq f(x')$. We now proceed as follows:

$$f(x) \subseteq f(x') \iff \text{“unpacking both sides”}$$
$$\sim f(x) : \sim f(x') \iff \text{“since } x = \{X\} \text{ and } x' = \{X'\}\text{”}$$
$$\sim f([X]) : \sim f([X']) \iff \text{“by definition of wholistic application”}$$

$$f.X : f.X'$$

Hence $X$ is the least fixed point for wholistic application of $f$.  

As a simple example we can use the equation $nat = 0, nat + 1$ mentioned above. Let $f$ be the function whose wholistic application is given by $f.X = 0, X + 1$ and suppose $X : X'$. Then wholistic application of $f$ is monotonic since:

$$f.X : f.X' = 0, X + 1 : X' + 1 = X + 1 : X' + 1 = X : X'$$

Theorem 1 tells us $X = f.X$ has a least solution, but doesn’t indicate what this solution is. However, we can use monotonicity to derive a sequence of approximations $X_i$ with $X_0 = \text{null}$ and $X_{i+1} = 0, X_i + 1$. This gives us $X_0 = \text{null}$, $X_1 = 0$, $X_2 = 0, 1$, $X_3 = 0, 1, 2$, $X_4 = ...$.

6.3. Improper bunch theory

We introduce the improper bunch to allow us to define a total-correctness semantics, as described in section 3. This requires us to have a level of non-determinism that goes beyond “anything goes”.

We refer to bunch theory which includes the improper bunch $\bot$ as “Improper bunch theory”.

To model the improper bunch, each given type $T$ in our theory is supported by a type $T'$ in our model language which includes the elements of $T$ plus an additional “improper element”. For example, let $T = \{a, b\}$ be a two element type in our language. This will be supported by a three element type $T' = \{a, b, \kappa_T\}$ in our model.

For use with our total correctness approach, the denotations of bunches of type $T$ are:

$$[\text{null}] = \{\}$$
$$[a] = \{a\}$$
$$[b] = \{b\}$$
$$[a, b] = \{a, b\}$$
$$[\bot_T] = \{a, b, \kappa_T\}$$

We extend this approach to constructed types. Writing the improper elements of two general types $S$ and $T$ as $\kappa_S$ and $\kappa_T$ the constructed type $S \times T$ is supported by the set $S \times T \cup \{\kappa_S \mapsto \kappa_T\}$, and the constructed type $\mathcal{P}(T)$ is supported by the set $\mathcal{P}(T) \cup \{\kappa_T\}$.

The reader will recall that we have provided a denotational model for bunch theory by extending language $\mathcal{M}$ to become $B$ and providing, directly or indirectly, denotations for all abstract syntax forms of $B$ in the model language $\mathcal{M}$. For improper bunch theory we use a more elaborate scenario that allows us to restrict improper elements to our model language. In addition to the extension of $\mathcal{M}$ to $B$ we provide a separate extension of $\mathcal{M}$ to $\mathcal{M}'$. This second extension includes the definitions of supporting types and provides the model language we need to provide a semantics of improper bunch theory.

Our semantic brackets now enclose abstract syntax forms from improper bunch theory and convert them to the language $\mathcal{M}'$.

For improper bunch theory we have the following additional axioms, in which $T$ is a general type, and $E$ and $F$ are expressions of type $T$.

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax $F$</th>
<th>Semantics $[F]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improper bunch</td>
<td>$\bot_T$</td>
<td>$T \cup {\kappa_T}$</td>
</tr>
<tr>
<td>Improper power</td>
<td>$\bot_{\mathcal{P}(T)}$</td>
<td>$\mathcal{P}(T) \cup {\kappa_T}$</td>
</tr>
<tr>
<td>Improper pair</td>
<td>$\bot_{S \times T}$</td>
<td>$S \times T \cup {\kappa_S \mapsto \kappa_T}$</td>
</tr>
</tbody>
</table>

Validation of the axioms of improper bunch theory against the model is straightforward, making use of the property that for a general type $T$, $\kappa_T \in [E] \iff [E] = [\bot_T]$, and noting that since bunch containment and bunch union are used in these axioms we need to establish their implicit semantics as part of the validation process.
7. Further properties of bunch theory

Here we list further properties. Some of these are proved in [27]

Bunch Comprehension properties

\[ \neg \{ x \mid P \} = \emptyset \quad \neg \{ x \mid P \cdot E \} = \emptyset \]

\[ \emptyset \cdot x : P = E \quad E = \emptyset \cdot x : E \quad \text{equivalent forms} \]

Equality properties

\[ E = F \Leftrightarrow \{ E \} = \{ F \} \]

\[ \delta(S) \land \delta(T) \Rightarrow (S = T \Leftrightarrow \neg S = \neg T) \]

Properties of bunch union and bunch intersection.

\[ (E \cdot F)^* = E \cdot F \]

\[ (E \cdot F)^* = (E \cdot G)^* \quad (E \cdot F)^* = (E \cdot F)^*(E \cdot G) \]

\[ E : E, E \rightleftarrows E \cdot E \]

\[ E, E = E \rightleftarrows E \cdot E = E \]

Distributive properties of guarded bunches.

\[ g \rightarrow (E, F) = (g \rightarrow E), (g \rightarrow F) \]

\[ (E \cdot F)^* = (g \rightarrow E)^*(g \rightarrow F) \]

Generalisation of certain laws

We have the following “part of” and “member of” laws:

\[ x : E, F \Leftrightarrow x : E \lor x : F \quad (x \text{ an element}) \]

\[ x \in E, F \Leftrightarrow x \in E \land x \in F \]

which generalise to the following bunch comprehension laws:

\[ x : \emptyset \Leftrightarrow y : Y \rightarrow E \Leftrightarrow \exists y : Y \land x : E \quad (x \text{ an element}) \]

\[ x \in \emptyset \Leftrightarrow y : Y \rightarrow E \Leftrightarrow \forall y : Y \Rightarrow x \in E \]

Some of the set theory axioms can be generalised. The ordered-pair axiom \(E \mapsto F \in (s \times t)\) \(\Leftrightarrow E \in s \land F \in t\) holds for non-elemental values, and this has already been validated in our model.

The power-set axiom \(s \in \mathbb{P}(t) \Leftrightarrow (\forall x : x \in s \Rightarrow x \in t)\) where \(x\) is not free in \(s\) or \(t\), still holds when \(t\) is non-elemental.

8. Related work

The main exposition on Bunch Theory and its applications is found in Hehner’s book [10,12], which uses a predicative style for describing sequential programs. This approach is applied to functional programs in the paper “Logical specifications for functional programs” [20], which introduces the important concept of the bunch guard. Bunches are used to incorporate non-determinism in functions by Morris and Bunchenburg [18], who axiomatise boolean and function types in the presence of non-determinism.

An important difference between the above work and our approach is that there is no separate logical layer, predicates just being boolean functions. Thus, predicates can take values such as “null” or “true, false”. Hehner generally considers such non-elementary predicate values as uninteresting in the context of program specification, whereas in [18] we consider considerable attention paid to the optimum adaptation of logic to bunches of truth values.

Morris and Bunchenburg provide a model. Like our original approach [27], it links values in the language and its model by means of bindings. In [19] they report on inconsistencies with their initial theory discovered during the model building process, and report related problems in the work of other authors. To discuss the problem in terms of our formalism, let \(f\) and \(g\) be distinct total functions. Define \(h_1 \equiv f \land g\) and \(h_2 \equiv f \lor g\). Both \(h_1\) and \(h_2\) are non-deterministic functions, with the same applicative effect, but we see a difference when treating functions as first class objects, i.e. objects used to instantiate variables. The assignment \(F := h_1\) is equivalent to \(F := f \cap F := g\), with non-determinism resolved at the point of the assignment (early resolution). No such resolution occurs in the assignment \(F := h_2\), because \(h_2\) is an element, and non-determinism remains to be resolved at the point where \(F\) is subsequently applied (late resolution). Inconsistency will occur in a theory that does not distinguish these cases, and attributes to non-deterministic late-resolution functions the properties of an early resolution function. The B Formalism helps us avoid this kind of problem by being “model-based”.\(^{10}\) We model functions as sets of ordered pairs, and non-deterministic functions as either a bunch

---

10 Here the word “model” is used in a different but not completely unrelated sense to its use in “denotational model”. In both cases we are presented with a mathematical model which we can interrogate.
of deterministic functions or as a relation. Then, rather than attributing properties to them, we can enquire as to their properties via a mathematical investigation of their definitions.

An early application of bunches was to formal grammars, with a language being a bunch of strings. In “The functional treatment of parsing” by René Leermakers [16] (1993, second edition 2012) non-deterministic functional parsing algorithms are used with the aim of bridging the gap between the parsing techniques of computer science and of natural language analysis. The author notes that the use of bunches smooths the transition from deterministic to non-deterministic algorithms, and allows the simple expression of formulae which “blow up” if expressed in set theory.

9. Conclusions

We have extended the underlying mathematical formalism of B to include bunches, which are unpackaged collections, i.e. the contents of sets. We have been able to provide an expressive semantics for a refinement calculus with simple formal rules to describe results provided by backtracking implementation programs. Placing our extended formalism on a sound mathematical basis has required us not only to provide a denotational semantics, but also to adjust the classical first-order predicate logic which underpins B, to render it incapable of proving results such as $3 \times \cdot x = 1/0$, which for us are pathological. We have been able to do this whilst still being able to prove the laws of an extensive logic toolkit in the same number of steps as are required using classical logic (with the requirement of a definedness premise for one-point laws being the only change required). The most striking feature of our approach is a mathematical formalisation of the concept of “nothing”, which, following Hehner, we call null and define as the contents of the empty set. Terms that would be undefined in a classical treatment of partial functions now literally define “nothing”, e.g. $1/0 = \text{null}$, and is perfectly well defined. This allows us to draw the same conclusions from partial function applications as we can from the application of total functions. For example from $\text{king.of}(\text{Norway}) = \text{Harald}$ we can conclude both that Norway has a king and that king is Harald. This contrasts with the use of partial functions in B, where to draw this conclusion we would also need to know that Norway belonged to the domain of the function $\text{king.of}$.

The null bunch, which conceptualises nothing, is used again and again in our formulations: for results from computations which do not occur (e.g. the else branch of a conditional statement when the if branch is selected), for infeasible continuations during backtracking search, for the parts of conditional expressions which are eliminated when the expression guard is evaluated, and for nonsensical descriptions (the present King of France) arising when partial functions are applied outside their domains. Our experience working with bunches makes a theory that is unable to conceptualise “nothing” seem as defective as a theory of arithmetic that omits zero. An intriguing possibility is that, with a suitable denotational model, the concept of “nothing” and it associated logic could be adapted for use with standard set theory.

CRediT authorship contribution statement


Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Two supporting documents covering formal semantics and proof have been submitted.

Acknowledgements

We thank the anonymous referees, whose comments have enabled us to greatly improve the presentation of this paper. We warmly thank Eric Hehner for extended discussions and colleagues from the BCS Formal Aspects special interest group for their interest and comments.

Appendix A. Precedence and associativity

In descending order, with connectives of equal precedence included within brackets, e.g. ($\ast / \text{mod}$), the order of precedence is:
function application (symbol elided). wholistic application
(* / mod) (+ −)
∪ ∩ set intersection and union
▷ ◀ range restriction and domain restriction
⇒ ⇔ total function, partial function and relation
× set cross product
(‘ ‘▷) bunch union, intersection and preference
(▷ < ≤ ≥) (≠ ≠) (∶ ∈ ∈ ∈ ∈) ∨ ∀ ⇒ ⇔
• as used in quantifications
⇒ bunch guard and bunch precondition
⊕ ⊕ assignment and program guard
∪ ∪ non-deterministic, preferential and probabilistic choice
∋ ∈ sequential composition of operations
∈ ∈ expectation and prospective value
∈ ∈ refinement
(∈ = ) definition and low precedence equality/equivalence

We have the following unary prefix symbols, whose precedence is above that of binary connectives.

− unary minus
¬ set unpacking
℘ ⊇ powerset and bunch comprehension
¬∀ ∀ expectation and existential
δ elementhood

The precedence level of •, together with the definition of ∀ and ∃ as unary prefixes, defines the scope of quantified variables in predicates.

The following binary connectives are right associative:

• ⇒ (logical implication) •

all others are left associative.

References