Bunch theory: applications, axioms and models

Bill Stoddart, Frank Zeyda, Steve Dunne

April 26, 2022

Abstract

In his book *A practical theory of programming* [Heh93], Eric Hehner proposes and applies a remarkably 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 compare the use of bunches with other approaches to this problem, and we illustrate the use of Bunch Theory in formulating program semantics combining non-deterministic, preferential, and probabilistic choice to provide a guarded command language whose exceptional expressivity we illustrate with a short case study. We show how an existing axiomatisation of set theory can be extended to incorporate bunches, and we provide and validate a model.

Contents

1 Introduction 3

2 The structure of the rest of the paper 6

3 Bunch theory in use 6

3.1 Bunch theory and descriptions 7

3.2 Bunch theory and programming language semantics 11

3.3 Programming language syntax 12

3.4 Programming language semantics 13

3.4.1 Weakest pre-condition and conjugate weakest pre-condition semantics 13

3.4.2 Prospective value semantics 14

3.4.3 A basic law linking prospective values and CWP 14
3.4.4 Derivation of the rules for $S \circ E$ .............................. 14
3.5 Bunches, homogeneity and the avoidance of complexity .... 15
3.6 Selection and iteration .................................................. 16
3.7 Provisional choice and backtracking ............................... 17
3.8 Case study ................................................................. 17
3.9 Preference as a refinement of provisional choice ............... 18
3.10 Using bunch properties in the formulation of probabilistic choice. 20

4 Extending axiomatic set theory to incorporate bunch theory 21
4.1 Our axioms ................................................................. 22
4.2 Weakening classical logic to deal with undefinedness ......... 23

5 Model 24
5.1 The rules of our model .................................................... 25

6 Proving the existence of a model 29
6.1 Example validations of set theory axioms ......................... 29
   6.1.1 Validation of the ordered pair axiom .......................... 30
   6.1.2 Validation of set comprehension axiom ....................... 30
6.2 Validation of the axioms specific to bunch theory ............. 32
   6.2.1 Bunch theory axiom, packaging 1 ............................. 32
   6.2.2 Bunch theory axiom, packaging 2 ............................. 32
   6.2.3 Bunch theory element axiom 1 ................................. 33
   6.2.4 Bunch theory element axiom 2 ................................. 33
   6.2.5 First guarded bunch axiom ..................................... 34
   6.2.6 Second guarded bunch axiom .................................. 34
6.3 Establishing the soundness of our logic .......................... 35
   6.3.1 The soundness of the exists-intro rule of FOPLN ........... 35
   6.3.2 The soundness of the exists-elimination rule of FOPLN ... 36

7 Conservative extensions to bunch notation 38

8 Some results in bunch theory 40
1 Introduction

Set theory deals with elements as singular entities. A set may contain elements, and is itself an element that can belong to another, more deeply structured, set. When forming a set from its elements we simultaneously perform two actions: collection and packaging. However, it is beneficial to think of these as two orthogonal activities. The theory that results from this approach we will refer to as bunch theory, following Hehner, who first suggested it.

Bunch theory provides benefits in terms of preserving homogeneity, accommodating what would otherwise be undefined terms, and describing non-determinism - a key concept in software development.

Hehner provides an axiomatisation for Bunch Theory and builds set theory on top of this. Our approach will be to take an existing axiomatisation of set theory and add additional axioms to incorporate bunches.

The base formalism we build on is that used in the B-Method of J-R Abrial [Abr96]. This gives us first order predicate logic with equality (FOPL) and a typed set theory. The B-Method provides us with the ability to construct basic sets, including the integers. Such sets are termed “maximal” since they are not subsets of any larger set. From any existing maximal sets we are able to produce new maximal sets by the use of set product and power-set operations. Each maximal set corresponds to a type.

Following Hehner [Heh93], we give a mathematical meaning to the contents of...
a set, which we call a bunch. We write ∼A (“unpack A”) for the contents of set A, thus ∼{1, 2} = 1, 2. The comma used in a bunch extension is now a mathematical operator, called bunch union. It is associative and commutative, and its precedence is just below that of the expression connectives.It is associated with set union via the rule

\{A\} ∪ \{B\} = \{A, B\}

A bunch A is a sub-bunch of B if each element of A is an element of B. We write this as A : B (“A is part of B”). Sub-bunches are related to subsets by the rule:

A : B ⇔ \{A\} ⊆ \{B\}

We define the bunch null to be the contents of the empty set. null literally represents nothing, a concept completely missing from other theories but which for us plays an essential rôle.

The introduction of bunches affects our logic. In FOPL all terms represent values, and it is possible to prove ∃x • x = 1/0, whereas in bunch theory 1/0 = null. In [SDM22] we propose a weakening of FOPL which we call FOPLN. FOPLN introduces a definedness predicate δ(E) to say that E is an element. We retain propositional logic and can still prove almost all predicate logic rules from our logic toolbox, the only exceptions being that one point rules require a definedness premise.

Standard function application is lifted when applied to a bunch, thus if A = x, y, then f(A) = f(x), f(y). We take the infix syntax of binary operations as a sugared form of standard function application, so these are similarly lifted to apply pairwise, e.g. adding the bunches 0, 1 and 2, 4 yields 0+2, 0+4, 1+2, 1+4. Using round brackets exclusively to express precedence (rather than sometimes using them to describe ordered pairs) and noting that bunch union (comma) has a lower precedence than +, we can write this as: (0, 1) + (2, 4) = 2, 3, 4, 5. Thus we sacrifice the use of standard brackets to represent tuples: (x, y) is no longer an ordered pair, for which we use maplet notation, i.e. x ↦→ y or bold brackets: ((x, y)). However we still use f(x, y) for application of function f to a pair of arguments, and when we apply a function f to a bunch of arguments x, y, we write this as f((x, y)).

Lifting similarly applies to tuple construction, so that, e.g. (1, 2) ↦→ (3, 4) = 1 ↦→ 3, 1 ↦→ 4, 2 ↦→ 3, 2 ↦→ 4

Lifting does not, however, apply to “packaging”, which is the inverse of the unpackaging operator described above. We package the bunch 1, 2 by enclosing it in set brackets, obtaining \{1, 2\}.

For us, as in B, a predicate is totally different from an expression. A predicate is a formula that may be subject to proof, whereas expressions denote values. In particular our predicates do not denote boolean truth values. Accordingly, predicates are not lifted in the way described above.

For use with bunches the two fundamental predicates we inherit from B, set

\footnote{A full list of precedences is given in Appendix A.}
membership, and equality, must be treated in different ways.

\( E \in S \) will be true when each element which is part of \( E \) is a member of each element which is part of \( S \). Thus \( 1, 2 \in \{1, 2, 3\}, \{2, 3, 4\} \) For elements \( x \) and \( y \) a predicate such as \( x < y \) is equivalent to \( x \mapsto y \in \_ < \_ \) where \( \_ < \_ \) is the relation containing all ordered pairs \( x \mapsto y \) for which \( x < y \). Thus we have \( 1, 2 < 3, 4, \) but \( \neg (1, 2 < 2, 3) \).

We cannot apply this treatment to equality since we inherit from first order predicate logic with equality the axiom \( E = E \). Therefore, for example, \( 1, 2 = 1, 2 \).

The guarded bunch \([NH92], p \rightarrow E\) is equal null, if predicate \( p \) is false, and is otherwise equal to the expression \( E \).

We will see many uses of guarded bunches. One we can give immediately is to define the conditional expression, for predicate \( p \) and expressions \( E, F \).

\[ \text{if } p \text{ then } E \text{ else } F \text{ end } \equiv (p \rightarrow E), (\neg p \rightarrow F) \]

The improper bunch \( \bot \) is used to describe “non-termination”. It represents the complete ignorance associated with a value provided by a computation that may have crashed. Such a failed computation might produce any value, or no value at all, and this is reflected in the improper bunch being strictly larger than any other bunch of its type, i.e. for any proper bunch \( A \), we have \( A : \bot \) and \( \neg (A = \bot) \).

The preconditioned bunch \( p \mid E \) is equal to the improper bunch \( \bot \) if \( p \) is false, and otherwise equal to \( E \).

From any set \( X \) we can construct a smaller set by set comprehensions of the form

\( \{x \mid x \in X \wedge P\} \)

All sets we consider are necessarily homogeneous.

We similarly restrict our attention to bunches which are homogeneous; these have the same type as their elements, thus \( 1, 2 \) is of type integer.

Analogously to set comprehension, we can write the bunch comprehension:

\[ \oint x \bullet E \]

Here, although the type of \( x \) is not given explicitly it has to be deducible from an examination of the expression \( E \). The result consists of all values that \( E \) can take as \( x \) ranges over the elements of its type. For example the bunch \( 10, 20 \) can be written as the bunch comprehension \( \oint x \bullet x : 1, 2 \rightarrow 10 \ast x \).

When we extend set theory to include bunches, a reasonable ambition is for every closed formula of the original theory to have the same meaning in the extended theory. This requires bound variables to denote elements, a property of FOPLN

\[ ^2 \text{There is a different null bunch and a different improper bunch for each type. Where we need to specify the type we do so by means of a suffix, e.g. } \bot_\{\text{integer}\} \text{ for the bottom bunch of type integer, but usually the type of any null or improper bunch we are considering will be obvious from context.} \]
which we prove in [SDM22]. For example in $\forall n \in \mathbb{Z} \Rightarrow \text{card}\{n\} = 1$ is true in our original set theory, and remains true in our extended theory since the bound variable $n$ denotes an element.

2 The structure of the rest of the paper

In section 3 we look at two application areas of bunch theory. Firstly we analyse the contribution it can make to theories of description due to its ability to eliminate undefined terms by representing them as null. Secondly, we show how use of bunches can facilitate the formulation of programming semantics, including combinations of non-deterministic, provisional and preferential and probabilistic choice within a backtracking context. These are illustrated with a case study that implements a search heuristic.

Since the introduction of bunches into set theory is a radical departure from mathematical tradition, we justify its soundness, along with the soundness of its logic FOPLN, by providing a model. In section 4 we extend the axiomatisation of set theory given in [Abr96] to incorporate bunch theory. In section 5 we define a model, which we validate in section 6. To simplify our validation, we keep our notation as sparse as possible, and in section 7 we formally extend our theory to include notations previously described in the informal introduction to bunch theory. In section 8 we report some results from bunch theory. In section 9 we extend our model to incorporate “improper bunches”, which are used to capture the occurrence and spread of error conditions, such as non-termination, which must be presumed when operations are executed outside their pre-conditions. In section 10 we report related work and in section 11 we give our conclusions.

Appendix A provides a fixed point treatment of loops and appendix B gives details of operator precedence and associativity.

The original contributions of this paper include an analysis of the contribution of bunch theory to the problem of undefined terms, a case study which illustrates the most novel aspects of program semantics, an axiomatisation based on extending set theory by giving a formal meaning to the contents of a set, the provision and validation of a model for this approach and for our logic FO-PLN which demonstrates the soundness of our theories, and a semantics for the “improper bunch” which our program semantics uses to model error conditions using the total correctness approach of R Back [Bac81], and used in the B-Method [Abr96], and in the Integrated Theories approach of Hoare and He [HJ98].

3 Bunch theory in use

We examine some contributions made by Bunch Theory, concentrating on three main areas: the resolution of problems associated with undefined terms, the expression of choice and randomness in programs, and the preservation of homogeneity to simplify formal descriptions.
3.1 Bunch theory and descriptions

Bunch theory solves many problems relating to “descriptions” that include undefined values and non-denoting terms and also values described in terms of a computation which does not terminate. For example the term $1/0$ will represent null, and this has well defined mathematical behaviour. We can prove $1/0 = 1/0$, which conforms with the rule for equality $E = E$ in First Order Predicate Logic with Equality. However, before we discuss the full impact of bunch theory on descriptions, let us briefly survey how troublesome the area of undefined terms has been.

In our formal notation, as in natural language, many objects are referred to by means of a description rather than being named directly. The problems which arise when such descriptions do not refer to anything (for example $1/0$ or “the present king of France”), or are ambiguous, have been the source of extensive debate in mathematics and philosophy at least since the emergence of Russell’s Theory of Descriptions (RTD) \cite{Rus05, WR10}. In response to the proposals of Meinong and Frege, Russell rejected approaches that attempted to directly assign some form of meaning or denotation to an invalid description (also referred to as “undefined term”), but showed how statements that involved such terms could nevertheless be handled in a consistent way. In RTD, undefined terms are non-denoting. They have no significance of their own account, but any assertion in which they have a primary occurrence entails an existential claim which results in the assertion being false. For example “the present king of France is bald” is false, since it entails a claim that there is a king of France; “the King of France is non-bald” is also false, but, by negation of our first example, “it is false that the King of France is bald” is true.

In formal notations, the definite description $\iota x \bullet P$ commonly represents the supposedly unique value of $x$ described by predicate $P$. A common form of definite description is function application, noting that the term $f(x)$ can be expressed as $\iota y \bullet x \mapsto y \in f$. In a formalism in which partial and total functions are a particular form of binary relation (an approach which RTD shares with a number of current formalisms including Z, B, and VDM) the term $f(x)$ is taken to be an invalid description when there is no unique $y$ such that $x \mapsto y \in f$.

In Abrial’s approach, instead of undefined terms being non-denoting (as in RTD), we hold that they do denote an object of the correct type, but we do not know what it is. This provides the advantage of a formal system compatible with predicate logic with equality. In particular we have the simple equality law $E = E$. For example we can prove $1/0 = 1/0$. This is not possible in RTD, because it entails the existential claim that there is a number that is $1/0$. However, Abrial’s approach means that stating $f(x) = y$ is not enough for us to conclude $x \in \text{dom}(f)$. To see why suppose we have the following rule for function application:

$$f(a) = b \Rightarrow a \mapsto b \in f$$

This is the function application rule in our formalism, which includes bunches, but does not work in Z and B. Using the empty relation $\{\}$ as a function
to deliberately provoke an undefined term, we can show, using this rule, that something belongs to the empty set. From our reflexive equality rule we have $\{}(a) = \{}(a)$. Hence, from the function application rule, we immediately obtain $a \mapsto \{}(a) \in \{}$. Now since $\{}a$ denotes some value (though we do not know what this is) so also must $a \mapsto \{}(a) \in \{}$: we have constructed an element that belongs to the empty set! To avoid this problem we must formulate a rule for function application that includes information about the domain and functionality of $f$ e.g.

$$f \in A \mapsto B \land a \in \text{dom}(f) \land f(a) = b \Rightarrow a \mapsto b \in f$$

and when writing a formal text the user has to be careful to give all the information required, which is not always easy, as in a recursive definition the domain of the function might not be explicitly given [SDG99].

Such considerations have led C B Jones and his associates to construct a three valued logic [BCJ84], in which predicates can be true, false, or undefined. In this approach we cannot fall back on classical logic, and we lose the law of the excluded middle as well as the simple law for equality. However, $f(x) = y$ does tell us that $y$ is the unique value associated with $x$ in $f$.

In the specification language Larch, and in the Isabelle/HOL proof assistant, partial functions are not available, and applications requiring functions that change according to events are modelled using total functions but giving only limited information about them. Asserting $f(a) = b$ then provides information about $f$ without any need for other conditions to hold. To model a birthday book we might have a set that associates people with dates, and rather than the set representing a partial function, it would contain information about a total function birthday (say) over part of its domain. Because birthday is a total function, mapping, say from PERSON to DATE we can say for any person $n$ that $\exists d \cdot d \in \text{DATA} \land \text{birthday}(n) = d$. We might not know the date of a person’s birthday, but we have no objection to the fact that such a date exists. However, this does not fit so well with a function king_of mapping from COUNTRY to PERSON, as it allows $\exists p \cdot p \in \text{PERSON} \land p = \text{king_of}(\text{france})$. I.e. there is an entailment that France has a king, though we don’t know who it is.

There are also approaches which reject predicates with undefined terms as possible candidates for proof. This may be done by requiring the discharge of some preliminary well definedness proof obligations. Alternatively, it may be achieved by syntactic restrictions that require predicates to be written in a way that ensures well definedness [AM02].

Some of the many approaches, are described in [Gri96, SDG99], and in [AM02] where the authors consider the treatment of the propositions $2 + 3/0 = 2$ and $2 + 3/0 \neq 2$ under 6 different regimes of undefinedness, none of which agree over the two results.

Let us now examine them in bunch theory. For the first we have:

$$2 + 3/0 = 2 \equiv 2 + \text{null} = 3 \equiv \text{null} = 3 \equiv \text{false}$$

In our formalism, the second is syntactic sugar for a set membership predicate,
where \( \neq \) is the relation consisting of pairs of unequal integer values, and where the type \( \mathbb{Z} \leftrightarrow \mathbb{Z} \) of the relation is given by context. We have:

\[
2 + 3/0 \neq 2 \equiv 2 + 3/0 \mapsto 2 \in \_\neq \_ \equiv \text{null} \mapsto 2 \in \_\neq \_ \equiv \text{true}
\]

Now consider the proposition:

\[
lives\_in(\text{son\_of}(\text{Charles})) = \text{uk}
\]

where Charles has two sons. We think of the relation \( \text{son\_of} \) as a non-deterministic function, which in this case returns a bunch of two values. The proposition will be true iff both sons live in the UK. The argument for this is that the bunch of two elements represent a non-deterministic choice, and as with non-deterministic choice in abstract command languages, we require that either choice must satisfy the given condition.

Returning to Russell’s example, we represent this as

\[
\text{king\_of}(\text{france}) \in \text{bald} \equiv \text{null} \in \text{bald} \equiv \text{true}
\]

thus in this case we disagree with RTD, but the truth we obtain is vacuous, like the predicate established by an infeasible operation [Mor90], or a universal quantification over an empty set.

It is possible to avoid undefined terms by using relational image rather than function application. The relational image of set \( S \) under relation \( R \) is defined as the set of these values \( y \) taken from pairs \( x \mapsto y \in R \) for which \( x \in S \), which we write as:

\[
R[S] \equiv \{ x, y \mid x \mapsto y \in R \land x \in S \bullet y \}
\]

and there is a close relationship with function application, since if \( f(a) = b \) then \( f[\{ a \}] = \{ b \} \). Relational image is always well defined, and allows us to capture non-existent objects via use of the empty set, e.g. \( \text{king\_of}[\{ \text{france} \}] = \{ \} \).

An interesting example of its use can be found in Version 1 of the Z Notation Standard [Nie95], a document which throughout uses relational image rather than function application. However, this technique is inevitably clumsy and was abandoned in later versions of the Standard.[4]

Finally we note that the underlying logic for our theory is Predicate Logic with Equality. Thus we have a classical two valued logic and the simple equality rule holds, in which every expression is equal to itself. We do not require special requirements on the domain of a function to draw conclusions from a function application, as for elements \( f, x, y \) we have the rules:

\[
f(x) = y \Rightarrow x \mapsto y \in f
\]

\[
f(x) = y \Rightarrow \neg (\exists z \bullet z \neq y \land a \mapsto z \in f)
\]

[4]In our bunch theory relational image is not defined since the value of \( R[S] \) can be written as \( \{ R(\_\neq \_S) \} \).
These allow us to define a function by its applicative effect and without additional reference to its domain. Let’s see how this works in the case of the factorial function, given the following properties which characterise it as an element and describe its application. Note that we use Hehner’s large equals as a low precedence equality.

\[
\xi(fact) = 1
\]

\[
fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot fact(n-1) & \text{else} \end{cases}
\]

These properties suggest an operational interpretation for executing the function \( fact \), but in our approach we don’t have direct access to such an interpretation. We follow the B-Method in taking functions to be mathematical rather than computational objects, and we handle non-termination at the level of operations by means of operation pre-conditions. Here, \( fact \) is particular type of set, a function from \( \mathbb{N} \to \mathbb{N} \), and we need a formal way to relate the above description of \( fact \) to this set. Furthermore, we wonder what would be a reasonable interpretation of the factorial of a negative number, for example \( fact(-3) \). The interpretation our theory provides is that \( fact(-3) = \text{null} \) on the grounds that no value is described by \( fact(-3) \).

We take the usual approach of using a higher order function \( F \) to formulate a fixed point equation \( F(fact) = fact \) and finding \( fact \) as the solution of this equation. We will be able to obtain the descriptive interpretation of \( fact \) as the greatest fixed point solution, \( fact = \nu F \).

\[
F \triangleq \lambda f \bullet \lambda n \bullet \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else} \end{cases}
\]

The existence of greatest and least fixed point solutions to \( F(fact) = fact \) is guaranteed by the Knaster-Tarski theorem. The associated partial order, which reflects how deterministic a function is, is the inverse of set inclusion ordering, and is thus a complete lattice. Since \( F \) is continuous, as well as monotonic, we will be able to derive our solutions as the limits of monotonic sequences of approximations to \( fact \). To obtain the greatest fixed point \( \nu F \) we use as first approximation the everywhere over-deterministic function \( \{ \} \). Each successive approximation is obtained by applying \( F \) to the previous one. Thus:

\[
\text{fact}_0 = \{ \}
\]

\[
\text{fact}_1 = F(\text{fact}_0) = \\
\lambda n \bullet \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \{ \} & \text{else} \end{cases}(n-1) \end{cases} = \\
\lambda n \bullet \begin{cases} 1 & \text{if } n = 0 \\ \text{null} & \text{else} \end{cases} = \\
\{ n | n = 0 \rightarrow 1, \neg (n = 0) \rightarrow \text{null} \} = \\
\{ 0 \rightarrow 1 \}
\]

\[
\text{fact}_2 = F(\text{fact}_1) = \\
\lambda n \bullet \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \{0 \rightarrow 1\} & \text{else} \end{cases}(n-1) \end{cases} = \\
\{ n | n = 0 \rightarrow 1, \neg (n = 0) \rightarrow n = 1 \rightarrow 1 \} = \\
\{ 0 \rightarrow 1, 1 \rightarrow 1 \}
\]

Each successive approximation nibbles away at the over-deterministic part of the previous function, providing one more point where a result is known. In
the limit, the result of applying $\nu F$ to any non-negative argument is known. However, application to a negative value remains overdeterministic, yielding $null$, in accordance with our descriptive requirements.

To obtain $\mu F$, the weakest fixed point solution of $F(fact) = fact$, we make our first approximation $All$, defined to be the function that returns $\sim Z$ at every point where it is applied. In this case each successive approximation replaces one point at which the previous approximation yielded $\sim Z$ with a point that yields a deterministic value. In the limit, the result of applying $\mu F$ to any non-negative argument is known. However, application to a negative value remains maximally non-deterministic, yielding $\sim Z$.

The reader may wonder whether we could use, as an initial approximation, a function which is everywhere non-terminating, and thus provide a computational interpretation of a function which shows for which arguments it would fail to terminate. That would require a more complicated theory of bunches than we provide here, but see Appendix A for a way in which we can overcome this apparent limitation when considering the fixed point treatment of loops.

In [JLS10] Jones et al classify attempts to deal with undefined terms into three main categories: those that use a logic adapted to undefinedness, those which provide an arbitrary, and usually unknown, denotation for each undefined term, and those that deal with undefined terms at the level of predicates. We have reviewed examples of all three approaches above, and shown that bunch theory provides another approach which is completely different as each term has a non-arbitrary denotation. However, terms may be non-deterministic, which allows them to be either over-defined, to the point of non-existence, or under-defined in the case of ambiguity. Whereas Jones et al advocate a logic adapted to undefinedness, bunch theory provides a theory of values which avoids undefinedness. In particular the use of $null$ and the improper bunch $\bot$ allows us to reason about the non-existent in a mathematically consistent way. The main adjustment required from the user of the theory is the loss of some laws, for example $A < B \lor A \geq B$ which can be false when we are dealing with non-deterministic values, e.g. we have $\neg (1, 3 < 2)$ but also $\neg (1, 3 \geq 2)$. But of course, all such laws are obeyed when we are dealing only with elements.

### 3.2 Bunch theory and programming language semantics

Within the semantics of a programming language we can use bunches to model non-determinism, and to express the results of a backtracking search carried out by a reversible computation.

Our agenda in this section will first of all be to see how non-deterministic or over deterministic values can be accommodated within the weakest pre-condition (WP) semantics used in the B-Method. We will then go on to derive, from predicate transformer semantics, an alternative “prospective value” (PV) semantics, equivalent in expressive power over the semantic components of B, but which also enables us to extend the expression language of B in an interesting way to express the results of a backtracking search, to express preferential choice, and
to incorporate probabilistic choice along with non-deterministic choice.

Before proceeding, we review some bunch theory results needed for the development of these theories.

A bunch is said to be atomic if it cannot be formed by the union of two non-null bunches which are both different from itself. Thus \( z = A, B \Rightarrow z : A \lor z : B \).

Elements are atomic; the null bunch is atomic; the improper bunch is atomic. For the theory we present here we take the design decision that no other bunches are atomic. To achieve this we make \( \bot \) absorptive in most contexts. E.g. \( a \mapsto \bot = \bot \) and \( \bot \mapsto b = \bot \). Also \( \bot \setminus A = \bot \). Indeed, although \( A : \bot \) for any bunch \( A \), if \( A \neq \text{null} \) our formalism gives us no way to express the elements that are part of \( \bot \) but not part of \( A \).

For any atomic bunch \( z \) we have the bunch comprehension property:

\[
z : \{ x \mid E \} \Leftrightarrow (\exists x \mid z : E)
\]

Recalling that the bunch comprehension \( \{ x \mid E \} \) is the bunch formed from all values of \( E \) as \( x \) ranges over its type, the bunch comprehension property tells us that if \( z \) is a sub-bunch of the bunch comprehension, there must be some value of \( x \) for which \( z : E \). This is intuitively obvious for elements; where \( z \) is the improper bunch it means that there is some \( x \) which yields the improper bunch all at once.

Atomicity also relates to bunch inclusion. We have \( A : B \) if, for an arbitrary atomic bunch \( z \), \( z : A \Rightarrow z : B \). Bunches \( A \) and \( B \) are equal if \( A : B \land B : A \), i.e. if \( z : A \Leftrightarrow z : B \).

We note that \( \bot : A \) is equivalent to \( \bot = A \), since \( A : \bot \) is true for any bunch \( A \).

Atomicity is mainly important because it allows us to write proofs without taking the improper bunch as a special case.

### 3.3 Programming language syntax

Here we give indicative abstract syntax for our guarded command language. In the following \( S \) and \( T \) are commands, \( g \) is a guard (i.e. a simple predicate which may be evaluated during the execution of a program), \( P \) is a predicate, \( x, y, u, v \) are a variables or comma separated variable lists, \( E \) is an expression or expression list.

The purpose of an abstract syntax is to characterise the possible parse trees of a language. This allows us to give a description that omits details of precedence and associativity (these details are provided in appendix B). Our descriptions are of syntactic forms. For example when we say \( S \cap T \) is an operation, this is to be taken as shorthand for “if \( S \) and \( T \) are the parse trees of operations, then \( S \cap T \) is also the parse tree of an operation, having \( \cap \) as its root node, \( S \) as its left sub-tree, and \( T \) as its right sub-tree”.
With this in mind we can list the syntactic forms of our commands.

*Skip* and *x := E* (basic commands).

*P | S* (pre-conditioned command), *S ; T* (sequence), *S \sqcap T* (choice), *S \triangleright T* (preferential choice) *S \oplus_p T* (probabilistic choice) *g \Rightarrow S* (guarded command).

```
if g then S end
if g then S else T end
while g do S end.
```

The syntax for procedure definition is

```
y \leftarrow Op(x) \triangleq P \mid S
```

and for procedure call:
```
v \leftarrow Op(u)
```

### 3.4 Programming language semantics

The B-Method [Abr96] is based on weakest pre-condition semantics. We write *

\[ S \] Q

* for the pre-condition necessary before execution of *S* to ensure that the post-condition *Q* will exist after *S* is executed.

We will give the rules for applying this semantics over the various abstract syntax forms of our language. These will generally follow those given in [Abr96] with the exception of unpounded non-deterministic assignment, where the presence of bunches will make the classical rule redundant.

We will also discuss an alternative and more powerful semantics of “prospective values”, in which we write *S \diamond E* for the value expression *E* would take were *S* to be executed. In this approach we also allow *S \diamond E* as an expression, which will give us a very eloquent expression language that will be illustrated in a case study.

#### 3.4.1 Weakest pre-condition and conjugate weakest pre-condition semantics

We write *

\[ S \] Q

* for the condition that must hold before operation *S* in order to ensure that predicate *Q* will hold afterwards. For the assignment *x := F* this condition, as given in the B-Book [Abr96], is as follows:

\[
[x := F]Q = Q[F/x]
\]

Now if *F* can be a bunch of values, each of which represents some value which *F* might take, we require, in order to guarantee *Q* after *S*, that *Q* would hold for each and all of the values comprising *F*, so our law becomes:

\[
[x := F]Q = \forall x' \cdot x': F \Rightarrow Q[x'/x]
\]
The B-Method has a non-deterministic assignment statement, but introducing bunches renders this redundant as we can now use a standard assignment to express unbounded non-determinism. Incorporating the new rule for non-deterministic assignment given above, and omitting a now redundant rule for unbounded non-determinism, our WP rules are as follows:

| Name             | Rule                                                                 |
|------------------|----------------------------------------------------------------------|
| skip             | [skip]Q = Q                                                          |
| assignment       | [x := F]Q = ∀x • x : F ⇒ Q                                          |
| pre-condition    | [P | S]Q = P ∧ [S]Q                                                  |
| guard            | [P ⇒ S]Q = P ⇒ [S]Q                                                |
| bounded choice   | [S ∩ T]Q = [S]Q ∧ [T]Q                                             |
| seq comp         | [S ; T]Q = [S][T]Q                                                 |

We have a dual set of rules for the conjugate weakest pre-condition (CWP), ⟨S⟩Q, the condition that Q might be true after S. We obtain the CWP of S from the WP of S according to:

⟨S⟩Q = ¬[S]¬Q

where the double negation corresponds to that seen in De Morgan’s laws, or more generally to that seen in changing from universal to existential quantification. Here is the derivation of the CWP law for bounded choice:

⟨S ∩ T⟩Q = “defn of CWP”
¬[S ∩ T]¬Q = “WP law for bounded choice”
¬([S]¬ Q ∧ [T]¬ Q) = “logic - de Morgan”
¬[S]¬ Q ∨ ¬[T]¬ Q = “defn of CWP”
⟨S⟩Q ∨ ⟨T⟩Q

### 3.4.2 Prospective value semantics

If S is an operation, we write S ∘ E for the value expression E would take were program S to be executed. We call S ∘ E the prospective value of E following S. For example x := 2 ∘ x + 10 = 12. Where S is non-deterministic, this will be reflected, for suitable E, in S ∘ E being a non-elemental bunch. For example x := 1 ∩ x := 2 ∘ x + 10 = 11, 12

### 3.4.3 A basic law linking prospective values and CWP

We now give an intuitive law that links prospective values with CWP. For any atomic bunch z where z is not free in E (which we write as z \ E) we have:

z : (S ∘ E) = ⟨S⟩z : E where z \ E “The basic law”

### 3.4.4 Derivation of the rules for S ∘ E

We can use the basic law to derive the following rules.

---

\[^{*}\text{See [SZD20] appendix B for a proof of this law from an explicit definition of S ∘ E.}\]
Name | Rule
--- | ---
skip | $\text{skip} \circ E = E$
assignment | $x := F \circ E \equiv (\lambda x.E)(F)$
pre-condition | $P \mid S \circ E = P \mid (S \circ E)$
guard | $P \implies S \circ E = P \implies (S \circ E)$
bounded choice | $S \sqcap T \circ E = (S \circ E), (T \circ E)$
seq comp | $S ; T \circ E = S \circ T \circ E$

The precedence of $\circ$ is below that of programming connectives, whose precedence, in descending order, is $\implies, =, \sqcap, ;, |$. We follow Hehner in introducing a large equals $\equiv$, which is a low precedence symbol used in equational reasoning. When written between expression or predicates it indicates that they are equivalent.

Our derivations are based on the proposition that for any bunches $A$ and $B$, and for any atomic bunch $z$ we have:

$$A = B \iff (z : A \iff z : B).$$

We provide example derivations for assignment and sequential composition.

For assignment and unbounded choice we have:

$$z : (x := F \circ E) = \text{"basic law, see section 3.4.3"}$$

$$\langle x := F \rangle z : E = \text{"CWP rule} (x := F)Q \iff (\exists x \bullet x : F \land Q)\text{"}$$

$$\exists x \bullet z : x : F \rightarrow E = \text{"property of bunch guard"}$$

$$z : \oint x \bullet x : F \rightarrow E \text{"and thus"}$$

$$x := F \circ E = \oint x \bullet x : F \rightarrow E$$

And for sequential composition:

$$z : (S ; T \circ E) = \text{"basic law"}$$

$$\langle S ; T \rangle z : E = \text{"CWP rule} (S ; T)Q \iff (S)\langle T \rangle Q\text{"}$$

$$\langle S \rangle (T) z : E = \text{"basic law"}$$

$$\langle S \rangle z : (T \circ E) = \text{"basic law and right assoc of \circ"}$$

$$z : (S \circ T \circ E) \text{"and thus"}$$

$$S ; T \circ E = S \circ T \circ E$$

3.5 Bunches, homogeneity and the avoidance of complexity

The contribution of bunch theory to the simplicity of our prospective-value semantics can be seen in the simplicity of the rules for $\text{skip}$ and for sequential composition compared to the rules that would be required were we to use classical set theory. In the latter case, to accommodate non-determinism, we require a set to express the possible values taken by expression $E$ following $S$. Both forms of the rules are given in the following table.

A full set of derivations is provided in [SZD20].
Name | Bunch Rule | Set Rule
--- | --- | ---
Skip | $\text{skip} \circ E = E$ | $\text{skip} \circ E = \{E\}$
Seq comp | $S ; T \circ E = S \circ T \circ E$ | $S ; T \circ E = \bigcup(S \circ T \circ E)$

The set rule for $\text{skip}$ illustrates a loss of homogeneity, which has a corresponding effect on the set rule for sequential composition. We can illustrate the added complexity of the set form by analysing $\text{skip} ; \text{skip} \circ E$ in both cases. Using the bunch rule we have:

$$
\text{skip} ; \text{skip} \circ E = \text{skip} \circ \text{skip} \circ E = \text{skip} \circ E = E
$$

and for the set rule

$$
\text{skip} ; \text{skip} \circ E = \bigcup \text{skip} \circ \text{skip} \circ E = \bigcup \text{skip} \circ \{E\} = \bigcup \{\{E\}\} = \{E\}
$$

### 3.6 Selection and iteration

The attentive reader will have noted that the semantic components do not include selection and iteration constructs. These are handled by means of choice and guard, as in the B-Method [Abr96].

The **if** construct is defined as:

$$
\text{if } g \text{ then } S \text{ else } T \text{ end} \equiv g \Rightarrow S \sqcap \neg g \Rightarrow T
$$

Let’s see how this works when evaluating the prospective value of expression $E$ after an **if** construct:

$$
\text{if } g \text{ then } S \text{ else } T \text{ end} \circ E = \text{“by defn of if”} \\
g \Rightarrow S \sqcap \neg g \Rightarrow T \circ E = \text{“by rule for bounded choice”} \\
(g \Rightarrow S \circ E), (\neg g \Rightarrow T \circ E) = \text{“by rule for guard”} \\
g \Rightarrow (S \circ E), \neg g \Rightarrow (T \circ E)
$$

When $g$ is true, the properties of the bunch guard tell us that this reduces to $(S \circ E), \text{null}$, i.e. to $S \circ E$. In this case the **else** clause contributes nothing to the result, and our ability to express this nothing by means of the empty bunch $\text{null}$ is essential to our formulation. Similarly, when $g$ is false the construct reduces to $\text{null}, (T \circ E)$, i.e. to $T \circ E$.

We have an iteration construct

$$
\text{while } g \text{ do } S \text{ end}, \text{ which we will refer to as } W.
$$

We define $W$ recursively by unwrapping the first iteration of the loop:

$$
W = \text{if } g \text{ then } S \circ W \text{ else } \text{skip} \text{ end}
$$

Giving us a recursive equation in programs. That such an equation has a solution is shown in the B-Book. We give a bunch theoretic treatment in appendix [A].
3.7 Provisional choice and backtracking

The semantics of choice, which we have used to incorporate demonic non-determinism into our language, can be used to describe provisional choice, as used in a backtracking application. This idea has a long history. As far back as 1967, in his paper “Non-deterministic Algorithms” [Flo67], Floyd talked of “programs governed in part.. by final causes for the sake of which their effects are carried out”. In “The Specification Statement” [Mor88], Carroll Morgan mentions the possibility as follows: “Ordinarily we limit the syntax of our programming language so that miracles cannot be written in it. If we relax this restriction, allowing naked guarded commands, then operational reasoning suggests a backtracking implementation”. Backtracking constructs are also discussed by Nelson [Nel89], and Norvel and Hehner [NH92] [Heh93].

We have explored the possibility of exploiting a prospective value semantics to represent backtracking, and in particular the implementation of backtracking via reversible computing [SZL06, SZD10, Zey07]. Operationally, a naked guarded command \( g \Rightarrow S \) will provoke reversal of computation if \( g \) is false, and will execute as \( S \) if \( g \) is true. Also, choice from a set \( x \in E \) will provoke reversal of computation if the expression \( E \) denotes an empty set. Reverse computation undoes any changes made during forward execution, and continues until a previous provisional choice is reached, at which point forward execution will recommence if a previously untried choice is available. Should there be no untried choices available and the program has been executed from a terminal, it returns to the user but with a different prompt, e.g. “ko” rather than “ok”, to indicate execution was infeasible.

The most novel feature of our proposed language is the incorporation, within our sub-language of expressions, of terms of the form \( \{ S \circ E \} \). These greatly increase the expressive power of our language, as illustrated in the following case study.

3.8 Case study

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 as 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 case study. We assume a global integer program variable \( \text{square} \) which holds the current position of the knight, and an operation \( \text{Move} \) which will make a provisional choice of a move.

---

*We have developed an implementation platform for this approach, in the form of a reversible virtual machine [SZL10].*
to a new, previously unvisited square. The set of all possible squares the knight can move to is given by the term \{Move ◦ 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.

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

In the computation of a term of the form \{S ◦ E\}, 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 ◦ E\} is then known, and forward execution can recommence without the evaluation of term \{S ◦ E\} having caused any change in the program state (or, operationally speaking, any enduring change, since any changes are restored during reverse execution).

Thus we have a simple semantics which allows us to express non-determinism and demonic choice along with provisional choice and backtracking within the same development framework. However, in a term of the form \{S ◦ E\} any choice within S must be considered as provisional choice rather than implementer’s choice, and must not be subject to refinement.

3.9 Preference as a refinement of provisional choice

In G Nelson’s generalisation of Dijkstra’s WP calculus [Nel89], we find a biased choice operator \(S \sqcup T\). In a backtracking scenario, \(S\) would be chosen in preference to \(T\), but if \(S\) is unable to produce a solution, \(T\) will be tried instead. Unfortunately, Nelson’s construct is limited by the fact that it is only \(S\) which has to be able to find a solution. We would like to extend this idea to take account not only of \(S\), but also of the rest of the program which follows. Nelson’s biased choice is defined using the feasibility of \(S\). We will first examine the idea of feasibility, and then extend it to take account of the effect of any further computation, which we refer to as the continuation. To achieve this we will use a construct which cannot be formulated in classical WP semantics, but which is made possible by our use of PV semantics and bunches.
Following Abrial we write the feasibility of $S$ as $\text{fis}(S)$ and recall that a program is feasible if it cannot establish false, i.e. $\text{fis}(S) \equiv \neg [S]\text{false}$.

We also recall a monotonicity condition of the WP calculus, that if predicate $Q'$ is weaker than $Q$, i.e. if $Q \Rightarrow Q'$, a program able to establish $Q$ will also be able to establish $Q'$. Thus, since $\text{false} \Rightarrow Q$ for any predicate $Q$, a program able to establish the post condition $\text{false}$ could establish absolutely anything. The truths established in this way, often known as “miracles” [MRG88], are vacuous, and depend on the computation having no after states. Thus they may be compared to the true result obtained from universal quantification over an empty set. These vacuous truths play an essential role in our semantics.

For example, in the decomposition, first proposed by Hehner[7] of a conditional $\text{if } g \text{ then } S \text{ else } T \text{ end}$ into the guarded choice $g \Rightarrow S \cap \neg g \Rightarrow T$, we use a vacuous truth to express the contribution of the choice which is not taken.

In a backtracking interpretation, a program is infeasible if it is always forced to backtrack, and accordingly has no after-states. Using this idea, Nelson’s biased choice can be defined as:

$$S \bowtie T \equiv S \cap \neg \text{fis}(S) \Rightarrow T$$

And we can clearly see that it is the feasibility of $S$ alone that guides the choice. We want to extend the idea of feasibility so it applies to the rest of the program, but first let us see how the predicate $\neg \text{fis}(S)$ might be expressed in PV semantics. Starting from our CWP definition $\text{fis}(S) \equiv \langle S \rangle\text{true}$, we have:

$$\neg \text{fis}(S) \equiv \text{“by defn of fis}(S)“}$$
$$\neg \neg [S]\text{false} \equiv \text{“expressing in CWP“}$$
$$\neg \langle S \rangle\text{true} \equiv \text{“since } z : \bot = \text{true“}$$
$$\neg \langle S \rangle z : \bot \equiv \text{“basic law“}$$
$$\neg z : (S \odot \bot) \equiv \text{“since } z \text{ is arbitrary“}$$
$$(S \odot \bot) : \text{null}$$

We would like to design a similar form of choice, which we will write as $\rangle\rangle$, but rather than looking just at the feasibility of $S$, we would like to consider the feasibility of $S$ along with any continuation of the program after the choice. Thus for the program $S \rangle\rangle T : U$, (where choice $\rangle\rangle$ binds more tightly than operation sequence “ ; ”) we would be interested in the feasibility of $S ; U$, and it would be the infeasibility of $S ; U$ that would force the choice of $T$. We propose the following definition of preferential choice

$$S \rangle\rangle T \circ E \equiv (S \circ E) , (S \circ E) = \text{null} \rightarrow (T \circ E)$$

This executes $S$ in preference to $T$, but if execution of $S$ or its continuation results in backtracking to this point, $T$ is then executed. A simple example will illustrate how bunch theory allows the expression $E$ to represent the feasibility of a continuation. We look at a program that would prefer the assignment $x := 1$ to $x := 2$, but is forced by a future guard to make the alternative choice.

7As recounted in [Heh06], this idea was first made public in a technical report because the reviewers of the paper in which it was proposed required it to be removed as a condition of acceptance.
In this example the message concerning the future evolution of the computation after the preferential choice is contained in the expression $x = 2 \rightarrow x$. This has the value null (i.e. expresses non-existence) unless $x = 2$, and in this way conveys that future execution must backtrack unless $x = 2$.

We have seen that our attempt to capture preference in WP led us only to biased choice; we were unable to capture preference in WP. However, preference occurs naturally in implementations, since these are intrinsically deterministic. We were able to capture preference in our PV semantics because in the term $S \circ E$ the expression $E$ is able to represent the total future behaviour of the code following $S$ (its continuation). When $E = \text{null}$ this continuation is infeasible.

Preferential choice is easily shown to be a refinement of provisional choice. The condition that program $S$ is refined by program $T$, when both programs have a state variable or variable list $x$, is expressed in terms of PV semantics as

$$S \sqsubseteq T \triangleq (T \circ E) : (S \circ E)$$

We can argue as follows:

$$S \triangleright T \circ E \triangleq \text{defn of preferential choice}$$

$$(S \circ E), (S \circ E) = \text{null} \rightarrow (T \circ E) : \text{“which is a sub-bunch of”}$$

“since $g \rightarrow \beta : \beta$ and $\beta' : \beta \Rightarrow \alpha, \beta' : \alpha, \beta$”

$$(S \circ E), (T \circ E) \triangleq \text{PV defn of non-deterministic choice}$$

$S \sqsubseteq T \circ E$

### 3.10 Using bunch properties in the formulation of probabilistic choice.

In classical probability theory, if $X$ is a random variable that takes value $x_1$ with probability $p$ and value $x_2$ with probability $1 - p$, then the “expected value” of $X$ is given by the weighted addition $E(X) = x_1 p + x_2$

where $x_1 p + x_2 = p * x_1 + (1 - p) * x_2$

In [SZ13] we formulate a probabilistic choice between programs, where for feasible continuations, $S \oplus T$ will choose execution of $S$ with probability $p$ and $T$
with probability $1 - p$. We define the expectation of some expression $X$ (of suitable type) after executing such a choice as a weighted addition. For $0 < p < 1$ this expectation is given by:

$$S_p \oplus T \circ_E X \equiv (S \circ_E X) \ p^+ \ (T \circ_E X)$$

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.

However, that is not quite enough to describe the backtracking behaviour of our reversible machine. If a probabilistic choice leads to infeasibility, then execution will reverse to the point of probabilistic choice and take the other choice. We incorporate this into our weighted choice by expressing a weighted choice as a bunch of three terms. We are using the same technique we previously showed for preferential choice.

$$X_1 p^+ X_2 \equiv X_1 = \text{null} \rightarrow X_2, X_2 = \text{null} \rightarrow X_1, \ p \ast X_1 + (1 - p) \ast X_2$$

Where $X_1$ and $X_2$ are non-null, this corresponds to the probabilistic choice not leading to infeasibility. The first two terms then equate to null and do not contribute to the result, which is given by the third term. If either $X_1$ or $X_2$ is null, then, by the absorptive properties of null, the third term will be null and the result will be given by the first two terms, at most one of which will be non-null.

Finally we need to consider what our expectation should be if a probabilistic choice could lead to an aborting computation. Our approach is that any possibility of abort is counted as abort. To achieve this needs no further modification of our weighted addition. If either $X_1$ or $X_2$ describes an abortive continuation, its value will be $\bot$. The term $p \ast X_1 + (1 - p) \ast X_2$ will then be $\bot$ (by the absorptive power of $\bot$), and the whole weighted addition will equate to $\bot$.

Through use of bunches we have been able to capture interactions between probabilistic choice, non-deterministic choice, feasibility and aborting behaviour in the world of values.

4 Extending axiomatic set theory to incorporate bunch theory

There are two aspects to extending set theory to incorporate bunch theory.

The first is to add new axioms which are specific to bunches, and make a slight adjustment to one of our existing axioms.

\footnote{In \cite{SZ13} 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 change it here to $S \circ_E X$.}
The second is to weaken classical logic FOPL to FOPLN, which is necessary because we can no longer assume that every expression denotes an element.

4.1 Our axioms

Hehner gives an axiomatisation in which bunch theory is presented as an independent, stand alone theory, to which the axioms of set theory may subsequently be added. Here we take a different approach in which we add axioms to set theory to capture the idea of a bunch as the contents of a set. We choose the formalisation of set theory given by Abrial in the B-Book\cite{Abr96}.

We must express one of Abrial’s axioms differently for it to be meaningful in the language of bunch theory. In the B-Book\cite{Abr96} Abrial writes the axiom of choice as:

\[ \exists x \cdot x \in s \implies \text{choice}(s) \in s \]

Since \( \text{null} \in s \) vacuously in bunch theory, once bunches are introduced Abrial’s form of the axiom allows \( \text{choice}(s) = \text{null} \). We use an equivalent form that forces \( \text{choice}(s) \) to be an element. To prove the equivalence of the two forms of this axiom, abstract them as \( a \implies b \) and \( a \implies c \), note that \( b \iff c \) by the one point rule,\textsuperscript{12} then note that by propositional logic:

\[ (b \iff c) \implies (a \implies b \iff a \implies c) \]

We now state our axioms:

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

We follow the general mathematical convention, also used by Abrial, by which variables may be implicitly universally quantified at the top level.\textsuperscript{13} This applies, for example, to the variables \( E, F, s \) and \( t \) in the ordered pair axiom. An added detail in bunch theory is that, since quantification takes place over elements, such implicitly quantified variables will always represent elements.

We add two axioms to characterise the meaning of infinite, a detail which is omitted from the B Book.

\textbf{Infinity 1} \quad \textit{infinite}(s) \Rightarrow \exists x \cdot x \in s

\textbf{Infinity 2} \quad \textit{infinite}(s) \iff \textit{infinite}(\{x \mid x \in s \land x \neq \text{choice}(s)\})

We now give the additional axioms which enable us to incorporate bunch theory. We keep the number of axioms to a minimum and we rely on post-hoc definitions to introduce many of the notations associated with bunches, including bunch union and intersection, bunch inclusion, the null bunch, and the lifted nature of bunches and bunch inclusion.

\textsuperscript{10}We conjecture that the same approach can be used with any axiomatisation of set theory.

\textsuperscript{11}Notation: we use \( x \setminus E \) to indicate that identifier \( x \) must not occur free in expression \( E \).

\textsuperscript{12}(\exists x \cdot x = E \land P) \iff P[E/x] \setminus E

\textsuperscript{13}By their nature, axioms do not include any free variables.
set membership within our bunch theory. This approach facilitates validation of our model and lowers the risk of missing axioms - since what might have been candidates for axioms become properties provable from post-hoc definitions.

| Name          | Rule       | Condition                                  |
|---------------|------------|--------------------------------------------|
| Packaging 1   | $\{\sim A\} = A$ | $A$ an element and a set                  |
| Packaging 2   | $\sim\{E\} = E$ | $E$ is an element                           |
| Element 1     | $\sim A$ is an element iff $A$ is a singleton set. |
| Element 2     | $\{E\}$ is an element                         |
| Guard 1       | $g \Rightarrow g \rightarrow E = E$          |
| Guard 2       | $\neg g \Rightarrow g \rightarrow E = \sim\{\}$ |

The packaging axioms introduce the unpacking operator $\sim$, and the use of set brackets to “package” a bunch. They tell us that packaging and unpackaging are inverses. That is not quite enough to characterise packaging - indeed these axioms would be true if neither packaging nor unpackaging had any effect; this is ruled out by the element axioms. The first element axiom tells us that the landscape of set theory extended with bunches contains mathematical objects not present in classical set theory, namely those values obtained by unpacking non-singleton sets. The second element axiom tells us packaging creates an element - this axiom will need to be qualified when we extend our theory further with the introduction of the improper bunch.

In the second guarded bunch axiom the term $\sim\{\}$ can be read as null, but we cannot write it that way as null is not yet defined.

4.2 Weakening classical logic to deal with undefinedness

We need to weaken the FOPL inference rules for quantifiers to obtain those we need for FOPLN. For a more complete discussion see [SDM22].

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:

$$
\frac{HYP \vdash P[E/X]}{HYP \vdash \exists x \cdot P} \quad \exists\text{-intro}
$$

However, once we are dealing with bunches an expression may represent null, and $P[E/x]$ will then be vacuously true, and provide no evidence that some $x$ exists such that $P$ is true. Our $\exists$-intro rule uses an additional premise that the expression we are citing is an element.

$$
\frac{HYP \vdash \delta(E), \ HYP \vdash P[E/X]}{HYP \vdash \exists x \cdot P} \quad \exists\text{-intro}
$$

The $\exists$-elim rule of FOPL is gives a generalised form of argument by cases. Suppose $x$ is not free in $Q$ (which we will write as $x \notin Q$), and that according to the hypotheses under which we are reasoning we can prove $P[\alpha/x] \Rightarrow Q$ where $\alpha$ is fresh, and suppose also that under the same hypotheses we can prove $\exists x \cdot P$, then we can conclude $Q$. 

23
The ∃-elim rule for FOPL is:

\[
\frac{HYP \vdash P[\alpha/x] \Rightarrow Q, \ HYP \vdash \exists x \bullet P}{HYP \vdash Q} \quad \alpha \text{ fresh}, \ x \notin Q \quad \exists\text{-elim}
\]

The ∃-elim rule for FOPLN provides an additional hypothesis δ(α) which we can use in establishing its conclusion.

The ∃-elim rule for FOPLN is:

\[
\frac{HYP, δ(α) \vdash P[α/x] \Rightarrow Q, \ HYP, δ(α) \vdash \exists x \bullet P}{HYP \vdash Q} \quad \alpha \text{ fresh}, \ x \notin Q \quad \exists\text{-elim}
\]

For a full description of FOPLN see [SDM22]. Similar adjustments are required for the universal quantification rules, but it will be sufficient to demonstrate the soundness of its existential quantifier rules to show the soundness of FOPLN.

5 Model

The aim of our model is to give the meaning of any expression, predicate or inference rule from our language of bunch theory underpinned by the logic FOPLN, in terms of the well established “meta-language” of set theory.

The set theory we use for our meta-language will support “environments”. An environment is akin to a tuple, but with its components accessed by name rather than position. We use environments to associate the names of free variables in our language with the values they denote in the meta-language of our model.

If we are considering a free variable x that takes a value 1, 2, we have the problem that the bunch value 1, 2 is not something that exists in the meta-language of our model. However, we can model it with the set \{1, 2\}. Proceeding in this way we associate every value in our language with a unique and distinct denotation in our meta-language.

An environment which associates the name x with 1, 2 and the name y with 3 will be written as:

\[\langle x \sim \{1, 2\}, \ y \sim \{3\} \rangle\]

Values of the components of an environment are obtained by projection. e.g.

\[\langle x \sim \{1, 2\}, \ y \sim \{3\} \rangle. x = \{1, 2\}\]

If E is an expression in our language, and ρ an environment that associates values with all free variables of E, then \[E]^\nu(\rho)\] will be the value denoted by E in our model when its free variables are bound to values given in the environment ρ. For example if ρ = \[\langle x \sim \{1, 2\}, \ y \sim \{3\} \rangle\] then

\[[x + y]^\nu(\rho) = \{4, 5\}\]

Because we distinguish predicates from values, our semantics for predicates differs from our semantics of values. \[P]^\nu(\mathcal{E})\] represents the “truth denotation” of predicate P, over the set of environments \mathcal{E}, its value being the subset of \mathcal{E} consisting of those environments in which P is true. Here are two examples in
which \( \rho \) is defined as above and \( \mathcal{E} \) is a singleton set:

\[
[x = y]^{\tau}(\{ \rho \}) = \{ \}
\]
\[
[x < y]^{\tau}(\{ \rho \}) = \{ \rho \}
\]

Within quantified predicates we may have to deal with variable capture, where
a quantified variable has the same name as some variable from an outer scope,
or we may experience an extension of an environment, where newly quantified
variables are added. To deal with these situations it is useful to equip our
meta-language with “binding override”, operation \( \oplus \). Here an example of its
use:

\[
(\{ x \sim 1, 2 \}, \ y \sim \{3\}) \oplus (\{ y \sim 4, 5 \}, \ z \sim \{2, 5, 7\}) =
(\{ x \sim 1, 2 \}, \ y \sim \{4, 5\}, \ z \sim \{2, 5, 7\})
\]

We provide rules to give the meaning of any expression, predicate or inference
rule in our target language in terms of its syntactic components: e.g. we would
give the meaning of an expression that has the form \( E \mapsto F \) in terms of the
meanings of \( E \) and \( F \) and the semantics of the maplet symbol. Thus our se-
manitics is compositional and based on the denotational rules we give for each
abstract syntax form in our language.

5.1 The rules of our model

In the following, \( c \) is an element and a constant; \( x \) a variable; \( E, F \) are general
expressions; \( s, t \) are elements and sets, \( \rho \) is an environment, and \( \mathcal{E} \) is a set of
environments.

In the term \([E]^{\tau}(\rho)\), \( E \) is an expression, and \( \rho \) an environment which associates
the free variables of \( E \) with particular values. \([E]^{\tau}(\rho)\) represents the “value
denotation” of expression \( E \) in environment \( \rho \).

Environments exist in the mathematical world of our meta language of set the-
ory, but their formation records values taken by variables in our bunch theory.
The binding \( x \sim 1, 2 \) records that the variable \( x \) is associated with the bunch
1, 2.

If \( E \) is composed entirely of constant terms, for example \( 3 \mapsto 4 \), it will not
depend on an environment, and we can write its denotation as just \([E]\).

We present our rules a few at a time, and before each set of rules we explain
the intuitions behind their formulation.

We first give the denotations for constants and variables.

The denotation of a constant is independent of its environment, for example \( 3 \) will denote \( \{3\} \) in the meta language of our model. Our semantics wraps each
constant term in set brackets.

\[
[c]^{\tau}(\rho) = [c] = \{c\}
\]

For variables, the value represented is given directly by the environment. No
For the denotation of an element we obtain the set denoting all maplets in bunch $E$.

$[x]^v(\rho) = \rho.x$

We next give denotations for packing and unpacking.

To obtain the denotation of $\{E\}$ we wrap the denotation of $E$ in brackets.

$[[E]]^v(\rho) = \{ [[E]]^v(\rho) \}$

For the denotation of unpacking we consider for the moment the unpacking of a single set $s$. The denotation of $s$ will be a singleton set, which we can unwrap by “choosing” its only element. Here we make use of the fact that, although the concept of unpacking a set does not, in general, exist in the set theory of our semantic universe, we can mimic it for the case of a singleton set.

$[\sim s]^v(\rho) = \text{choice}([[s]]^v(\rho))$

We now consider some further expressions which can be expressed without considering the denotation of predicates.

The maplet rule gives the denotation of $E \mapsto F$ for general expressions $E$ and $F$, which may be bunches with multiple elements. The values denoted by $E$ and $F$ in the semantic universe will be sets, and by taking the cross product of these we obtain the set denoting all maplets in bunch $E \mapsto F$.

$[[E \mapsto F]]^v(\rho) = [[E]]^v(\rho) \times [[F]]^v(\rho)$

For the denotation of an element which is a power set, we can allow ourselves to reflect that any elementary values we can represent in our target language of bunch theory will also exist in the set theory language of our semantic universe. Indeed, the term $\text{choice}([[s]]^v(\rho))$ will represent, in the semantic universe, exactly the value that we intend $s$ to represent in our target language under environment $\rho$. We call this the “facsimile” of $s$. We obtain our denotation by taking the power-set of this term and enclosing it in set brackets.

$[[P(s)]]^v(\rho) = \{ P(\text{choice}([[s]]^v(\rho))) \}$

Consider a simple application of the above rule in which $s$ is the constant term $\{1, 2\}$. Then using our denotation for constants, we have $[[s]]^v(\rho) = [[\{1, 2\}]] = \{ \{1, 2\} \}$. Hence $\text{choice}([[s]]^v(\rho)) = \{1, 2\}$ and:

$\{ P(\text{choice}([[s]]^v(\rho))) \} = \{ P(\{1, 2\}) \} = \{ \{\}, \{1\}, \{2\}, \{1, 2\} \}$.}

For the cross product denotation, we form the terms $\text{choice}([[s]]^v(\rho))$ and $\text{choice}([[t]]^v(\rho))$ in the semantic universe, and, since these will have the same values as we intend for $s$ and $t$ under environment $\rho$ in the target language, we form the denotation by enclosing the cross product of the former terms in set brackets.

$[[s \times t]]^v(\rho) = \{ \text{choice}([[s]]^v(\rho)) \times \text{choice}([[t]]^v(\rho)) \}$

For the denotation of $\text{choice}(s)$, we identify the special cases in which $s$ has denotation $\{\}$ or $\{\{\}\}$ which correspond to $s$ being $\text{null}$ or the empty set respectively. In these cases $\text{choice}(s)$ will be $\text{null}$ and its denotation will be the empty set. Otherwise we form the term $\text{choice}([[s]]^v(\rho))$ in the semantic uni-
verse, since this will have the value we intend for \( s \) under environment \( \rho \) in our target language. We take the choice of this term and our denotation is the singleton set containing this element.

\[
[\text{choice}(s)]^\nu(\rho) = \\
\text{if } [s]^\nu(\rho) = \{ \} \lor [s]^\nu(\rho) = \{ \{ \} \} \\
\text{then } \{ \} \\
\text{else } \{ \text{choice}(\{[s]^\nu(\rho)\}) \}
\]

The diligent reader may wish to work through this formula for a simple example. We will do so for \([s]^\nu(\rho) = \{\{1,2\}\}\), (as would be the case, for example, if \( s \) was the constant term \{1,2\}) and where \text{choice}\{1,2\} = 2. In this case the denotation is given by the \text{else} clause, and we have:

\[
\{ \text{choice}(\text{choice}(\{[s]^\nu(\rho)\})) \} = \{ \text{choice}(\{\{1,2\}\}) \} = \{ \text{choice}(\{\{1,2\}\}) \} = \{2\}.
\]

We next turn to “truth denotations” of fundamental predicates. Recall that given a set of environments \( E \) and a predicate \( P \), \([P]^\nu(E)\) will be that subset of \( E \) containing environments in which \( P \) is true.

\[
[\text{true}]^\nu(E) = E \quad [\text{false}]^\nu(E) = \{ \}
\]

To express the semantics of equality we form the set comprehension of those environments from \( E \) for which the denotation of \( E \) is equal to that of \( F \).

\[
[E = F]^\nu(E) = \{ \rho \mid \rho \in E \land [E]^\nu(\rho) = [F]^\nu(\rho) \}
\]

For set membership \( E \in s \), \( E \) should consist only of elements which belong to \( s \). The denotation of \( E \) in some environment \( \rho \) will be the set containing these elements, and this must be a subset of the semantic facsimile of \( s \).

\[
[E \in s]^\nu(E) = \{ \rho \mid \rho \in E \land [E]^\nu(\rho) \subseteq \text{choice}([s]^\nu(\rho)) \}
\]

We need a predicate to express that a value is an element. This will be the case when its denotation is a singleton set.

\[
[\text{card}(E)]^\nu(E) = \{ \rho \mid \rho \in E \land \text{card}([E]^\nu(\rho)) = 1 \}
\]

Now composite predicates.

\[
[\neg P]^\nu(E) = \{ \rho \mid \rho \in E \land [P]^\nu(\rho) = \{ \} \}
\]

\[
[P \land Q]^\nu(E) = [P]^\nu(E) \cap [Q]^\nu(E)
\]

\[
[P \lor Q]^\nu(E) = [P]^\nu(E) \cup [Q]^\nu(E)
\]

For the guarded bunch \( g \rightarrow E \) we need a way to say whether \( g \) is true in a particular environment \( \rho \). For this we apply the truth semantics of \( g \) to the singleton set \( \{\rho\} \). Where \( g \) is true we have \([g]^\nu(\{\rho\}) = \{\rho\}\).

\[
[g \rightarrow E]^\nu(\rho) = \text{if } [g]^\nu(\{\rho\}) = \{\rho\} \text{ then } [E]^\nu(\rho) \text{ else } \{ \} \end{end}
\]
Free variables in our source language may be declared in the forms \{x \mid x \in s \land P\}, \forall x \bullet x \in s \Rightarrow P, \exists x \bullet x \in s \land P. To express the denotation of an occurrence of a free variable \(x\) ranging over \(s\) we form the facsimile of \(s\) in the semantic universe and declare a free variable \(x'\) which ranges over the facsimile, and we use binding override \(\rho \oplus (x \sim \{x'\})\) to describe the variable capture in the new environment corresponding to the scope of \(x\). The normal semantics for a variable then applies, with: \([x]^{\nu}(\rho \oplus (x \sim \{x'\})) = \{x'\}\).

We next consider set comprehension, where we will again need to employ binding override. We form, in the semantic universe, the facsimile of the set we are intending to represent in the source language, and we enclose this within another layer of set brackets. To produce the facsimile of \(\{x \mid x \in s \land P\}\) we use a set comprehension in which bound variable \(x'\) ranges over the facsimile of \(s\), and we retain those \(x'\) for which our intention is that predicate \(P\) is true in environment \(\rho\) overridden with a binding of \(x\) to \(\{x'\}\). The trick for expressing this is to apply the truth semantics of \(P\) to a singleton set of environments containing just \(\rho\) with the binding for \(x\) overridden by \((x \sim \{x'\})\). If this yields an empty set, \(P\) is false in \(\rho\), otherwise it is true.

\[
[[x \mid x \in s \land P]]^{\nu}(\rho) = \{ \{x'\mid x' \in \text{choice}(\{s\}^{\nu}(\rho)) \land [P]^{\nu}(\{\rho \oplus (x \sim \{x'\})\}) \neq \{\} \} \}
\]

The truth denotations for universally and existentially quantified predicates use similar techniques to those employed for set comprehension.

\[
[\forall x \bullet x \in s \Rightarrow P]^{\tau}(\mathcal{E}) = \{ \rho \mid \rho \in \mathcal{E} \land ([\forall x'\bullet x' \in \text{choice}(\{s\}^{\nu}(\rho)) \Rightarrow [P]^{\tau}(\{\rho \oplus (x \sim \{x'\})\}) \neq \{\}) \}
\]

\[
[\exists x \bullet x \in s \land P]^{\tau}(\mathcal{E}) = \{ \rho \mid \rho \in \mathcal{E} \land ([\exists x'\bullet x' \in \text{choice}(\{s\}^{\nu}(\rho)) \land [P]^{\tau}(\{\rho \oplus (x \sim \{x'\})\}) \neq \{\}) \}
\]

Where the predicate \(P\) is \textit{true}, the denotation for the existential predicate becomes:

\[
[[\exists x \bullet x \in s]]^{\tau}(\mathcal{E}) = \{ \rho \mid \rho \in \mathcal{E} \land ([\exists x'\bullet x' \in \text{choice}(\{s\}^{\nu}(\rho))) \}
\]

We now consider the denotations of expressions and predicates subject to substitution of certain terms. For these, we capture the effect of the substitution through binding override.

\[
[[E/F/x]]^{\nu}(\rho) = [[E]]^{\nu}(\rho \oplus (x \sim [[F]]^{\nu}(\rho)))
\]

\[
[[P/F/x]]^{\tau}(\mathcal{E}) = \{ \rho \mid \rho \in \mathcal{E} \land [P]^{\tau}(\{\rho \oplus (x \sim [[F]]^{\nu}(\rho))) \neq \{\}) \}
\]

Our final two rules give the denotation of the set \textit{BIG} and the semantic characterisation of an infinite set. We use the naturals to construct the denotation of \textit{BIG}. The idea here is just to provide an infinite set for the construction of mathematical objects.

\[
[[\textit{BIG}]]^{\nu}(\rho) = \{ \mathbb{N} \}
\]

The semantic characterisation of an infinite set requires its facsimile in the semantic model to be infinite.
\[ [\text{infinite}(s)]^\tau(\mathcal{E}) = \{ \rho \mid \rho \in \mathcal{E} \land \text{infinite}(\text{choice}([s]^\nu(\rho))) \} \]

6 Proving the existence of a model

So far we have given axioms for bunch theory, and given semantic rules for translating any expression or predicate in the basic language of bunch theory into the language of our semantic universe.

At this point, the key question is whether the semantic translations of the axioms of bunch theory yield predicates which can be proved to be true in the semantic domain. If we can establish this we will have proved that a model exists for our version of bunch theory.

All expressions in our source language which are intended to denote elements will have denotations which are singleton sets. The formal proof of this is by structural induction over the syntactic forms of the source language, appealing to the semantic rules, and with the base cases being for constants and variables.

Where an axiom has the form \( P \iff Q \) we validate it within the model by showing \([P]^\tau(\mathcal{E}) = [Q]^\tau(\mathcal{E})\) for arbitrary \( \mathcal{E} \).

Where an axiom has the form \( P \Rightarrow Q \) we validate it by showing \([P]^\tau(\mathcal{E}) \subseteq [Q]^\tau(\mathcal{E})\) for arbitrary \( \mathcal{E} \).

Where an axiom has the form \( E = F \) we validate it by showing \([E]^\nu(\rho) = [F]^\nu(\rho)\) for arbitrary \( \rho \).

Where an axiom has the general form \( P \) we validate it by showing \([P]^\tau(\mathcal{E}) = \mathcal{E}\) for arbitrary \( \mathcal{E} \).

To validate an inference rule we show that any environment that results in the premises of the rule being true will also result in the conclusion of the rule being true.

For our validation we use the set theory formulated in the B-Book [Abr96], augmented with environments. During the validations we call on various properties of this set theory, including its axioms.

We give illustrative examples of axiom validations. An more extensive validation is given in [SZD20].

6.1 Example validations of set theory axioms

It is no surprise to find that our set theory axioms can be validated; we already have faith in set theory! However, the validation provides assurance that our model is correct. We give example validations for the ordered pairs and set comprehension axioms.
6.1.1 Validation of the ordered pair axiom

To validate $E \mapsto F \in (s \times t) \iff E \in s \land F \in t$

We show:

$$[E \mapsto F \in (s \times t)]^\tau(\mathcal{E}) = [E \in s \land F \in t]^\tau(\mathcal{E})$$

Taking the LHS

$$[E \mapsto F \in (s \times t)]^\tau(\mathcal{E}) = \text{“by semantics of set membership”}$$

$$\{ \rho \mid \rho \in \mathcal{E} \land [E \mapsto F]^\nu(\rho) \subseteq \text{choice}([s \times t]^\nu(\rho)) \}$$

= “by semantics of ordered pair and cross product”

$$\{ \rho \mid \rho \in \mathcal{E} \land [E]^\nu(\rho) \times [F]^\nu(\rho) \subseteq \text{choice}([s]^\nu(\rho) \times \text{choice}([t]^\nu(\rho))) \}$$

= “applying choice and by property that for non-empty $a$ and $b$,

$\ a \times b \subseteq c \times d$ $\iff a \subseteq c \land b \subseteq d$”

$$\{ \rho \mid \rho \in \mathcal{E} \land [E]^\nu(\rho) \subseteq \text{choice}([s]^\nu(\rho)) \land [F]^\nu(\rho) \subseteq \text{choice}([t]^\nu(\rho)) \}$$

= “by property of set intersection”

$$\{ \rho \mid \rho \in \mathcal{E} \land [E]^\nu(\rho) \subseteq \text{choice}([s]^\nu(\rho)) \} \cap \{ \rho \mid \rho \in \mathcal{E} \land [F]^\nu(\rho) \subseteq \text{choice}([t]^\nu(\rho)) \}$$

= “by semantics of set membership”

$$[E \in s]^\tau(\mathcal{E}) \cap [F \in t]^\tau(\mathcal{E}) = \text{“by semantics of } \land \text{”}$$

$$[E \in s \land F \in t]^\tau(\mathcal{E}) \square$$

Here we have validated a more general result than the original ordered pair axiom, since $E$ and $F$ in the above may be general expressions rather than expressions that must represent elements.

6.1.2 Validation of set comprehension axiom

$E \in \{ x \mid x \in s \land P \} \iff E \in s \land P[E/x]$ where $x \notin s$

We show:

$$[E \in \{ x \mid x \in s \land P \}]^\tau(\mathcal{E}) = [E \in s \land P[E/x]]^\tau(\mathcal{E})$$

Taking LHS

$$[E \in \{ x \mid x \in s \land P \}]^\tau(\mathcal{E}) = \text{“by semantics of set membership”}$$
\[
\{ \rho \mid \rho \in \mathcal{E} \land [E]^\nu(\rho) \subseteq \text{choice}(\llbracket \{ x \mid x \in s \land P \}\rrbracket^\nu(\rho)) \}
\]

= “by semantics of set comprehension”

\[
\{ \rho \mid \rho \in \mathcal{E} \land [E]^\nu(\rho) \subseteq
\text{choice}(\{ x' \mid x' \in \text{choice}(\llbracket s\rrbracket^\nu(\rho)) \land [P]^\tau(\{ \rho \oplus \{ x \leadsto \{ x' \} \}) \neq \{ \} \}) \}
\]

= “taking choice of singleton set”

\[
\{ \rho \mid \rho \in \mathcal{E} \land [E]^\nu(\rho) \subseteq
\text{choice}(\llbracket s\rrbracket^\nu(\rho)) \land [P]^\tau(\{ \rho \oplus \{ x \leadsto \{ x' \} \}) \neq \{ \}) \}
\]

= “by singleton set property \{ a \} \subseteq B \iff a \in B”

Now recalling set theory axiom 3:

\[
E \in \{ x \mid x \in s \land P \} \iff E \in s \land P[E/x]
\]

applying this to

\[
\text{choice}([E]^\nu(\rho)) \in
\{ x' \mid x' \in \text{choice}(\llbracket s\rrbracket^\nu(\rho)) \land [P]^\tau(\{ \rho \oplus \{ x \leadsto \{ x' \} \}) \neq \{ \}) \}
\]

with:

\[
\text{choice}([E]^\nu(\rho)) \text{ for } E
\]

\[
x' \text{ for } x
\]

\[
\text{choice}(\llbracket s\rrbracket^\nu(\rho)) \text{ for } s
\]

and \([P]^\tau(\{ \rho \oplus \{ x \leadsto \{ x' \} \}) \neq \{ \}) \text{ for } P
\]

the predicate \( E \in s \land P[E/x] \) in the axiom becomes

\[
\text{choice}([E]^\nu(\rho)) \in \text{choice}(\llbracket s\rrbracket^\nu(\rho)) \land
([P]^\tau(\{ \rho \oplus \{ x \leadsto \{ x' \} \}) \neq \{ \}) \text{choice}(\llbracket E]^\nu(\rho)) / x'
\]

which after performing the indicated substitution becomes

\[
\text{choice}([E]^\nu(\rho)) \in \text{choice}(\llbracket s\rrbracket^\nu(\rho)) \land
\llbracket P\rrbracket^\tau(\{ \rho \oplus \{ x \leadsto \{ \text{choice}([E]^\nu(\rho)) \} \}) \neq \{ \})
\]

And thus, LHS =

\[
\{ \rho \mid \rho \in \mathcal{E} \land \text{choice}([E]^\nu(\rho)) \in \text{choice}(\llbracket s\rrbracket^\nu(\rho)) \land \llbracket P\rrbracket^\tau(\{ \rho \oplus \{ x \leadsto \{ \text{choice}([E]^\nu(\rho)) \} \}) \neq \{ \}) \}
\]

We now take RHS and show it equal to the same expression:
\[ [E \in s \land P[E/x]]^\tau(\mathcal{E}) = \text{“by semantics of } \land \text{”} \]
\[ [E \in s]^\tau(\mathcal{E}) \cap [P[E/x]^\tau(\mathcal{E}) \]
\[ = \text{“by semantic rules for set membership and substitution”} \]
\[ \{ \rho \mid \rho \in \mathcal{E} \land [E]^{\nu}(\rho) \subseteq \text{choice}(\{[x]^{\nu}(\rho)) \} \} \cap \]
\[ \{ \rho \mid \rho \in \mathcal{E} \land [P]^{\tau}(\{\rho \oplus \langle x \sim [E]^{\nu}(\rho)) \} \neq \{ \} \} \]
\[ = \text{“by property of set intersection”} \]
LHS □

### 6.2 Validation of the axioms specific to bunch theory

The validation of the axioms specific to bunch theory establishes that they are true (with respect to the model).

#### 6.2.1 Bunch theory axiom, packaging 1

\[ \{\sim A\} = A \text{ where } A \text{ is an element and a set.} \]

To validate we show:
\[ [[[\sim A]]^{\nu}(\rho) = [[A]]^{\nu}(\rho) \]

Taking LHS
\[ [[[\sim A]]^{\nu}(\rho) = \text{“by semantics of packaging”} \]
\[ \{[[\sim A]]^{\nu}(\rho)\} = \text{“by semantics of unpackaging”} \]
\[ \{\text{choice}([[A]]^{\nu}(\rho))\} = \text{“since the choice is from a singleton set”} \]
\[ [A]^{\nu}(\rho) \]

#### 6.2.2 Bunch theory axiom, packaging 2

\[ \sim\{E\} = E \]

To validate we show:
\[ [[[\sim\{E\}]]^{\nu}(\rho) = [[E]]^{\nu}(\rho) \]

Taking LHS
\[ [[[\sim\{E\}}^{\nu}(\rho) = \text{“by semantics of unpackaging”} \]
\[ \text{choice}([[E]]^{\nu}(\rho)) = \text{“by semantics of packaging”} \]
\[ \text{choice}([[E]^\tau(\rho)]) = \text{“taking choice”} \]
\[ [E]^\nu(\rho) \]

□
6.2.3 Bunch theory element axiom 1

\[ \delta(\sim A) \leftrightarrow A = \{ \text{choice}(A) \} \quad (A \text{ an element and a set.}) \]

To validate we show:

\[ [\delta(\sim A)]^\tau(\mathcal{E}) = \mathcal{A} = \{ \text{choice}(A) \}^\tau(\mathcal{E}) \]

\[ = \quad \text{by semantics of } \delta \text{ on LHS and } = \text{ on RHS} \]

\[ \{ \rho \mid \rho \in \mathcal{E} \land \text{card}(\sim A)^\nu(\rho) = 1 \} = \]

\[ \{ \rho \mid \rho \in \mathcal{E} \land [A]^\nu(\rho) = \{ \text{choice}(A) \}^\nu(\rho) \} \]

\[ = \quad \text{by property of set comprehension} \]

\[ \text{card}(\sim A)^\nu(\rho) = 1 \leftrightarrow [A]^\nu(\rho) = \{ \text{choice}(A) \}^\nu(\rho) \]

\[ = \quad \text{by semantics of unpacking on LHS, with } A \text{ an element, and semantics of packing on RHS} \]

\[ \text{card}(\text{choice}([A]^\nu(\rho))) = 1 \leftrightarrow [A]^\nu(\rho) = \{ \text{choice}(A) \}^\nu(\rho) \]

\[ = \quad \text{applying semantics of choice to RHS} \]

\[ \text{card}(\text{choice}([A]^\nu(\rho))) = 1 \leftrightarrow \{ \text{choice}(\text{choice}([A]^\nu(\rho))) \} \]

Noting the case \( \neg \exists x' \cdot x' \in \text{choice}([A]^\nu(\rho) \) does not apply since \( A \) is an element and therefore in non-null and has a non-empty denotation, our proof goal becomes:

\[ \text{card}(\text{choice}([A]^\nu(\rho))) = 1 \leftrightarrow [A]^\nu(\rho) = \{ \text{choice}(\text{choice}(A)^\nu(\rho)) \} \]

Since \( A \) is an element its denotation will be a singleton set, and we can rewrite \( [A]^\nu(\rho) \) as \{ choice([A]^\nu(\rho)) \}, giving:

\[ \text{card}(\text{choice}([A]^\nu(\rho))) = 1 \leftrightarrow \text{choice}(A)^\nu(\rho) = \{ \text{choice}(\text{choice}(A)^\nu(\rho)) \} \]

Writing \( S \) for \( \text{choice}([A]^\nu(\rho) \) we have:

\[ \text{card}(S) = 1 \leftrightarrow \{ S \} = \{ \text{choice}(S) \} \]

\[ = \quad \text{card}(S) = 1 \leftrightarrow S = \{ \text{choice}(S) \} \]

\[ = \quad \text{true} \]

6.2.4 Bunch theory element axiom 2

\[ \delta(\{ E \}) \]

To validate we show:

\[ [\delta(\{ E \})]^\tau(\mathcal{E}) = \mathcal{E} \]

\[ [\delta(\{ E \})]^\tau(\mathcal{E}) = \text{by semantics of } \text{element} \]
\{ \rho \mid \rho \in \mathcal{E} \land \text{card}(\llbracket [E] \rrbracket \nu(\rho)) = 1 \} = \text{"by semantics of packaging"}

\{ \rho \mid \rho \in \mathcal{E} \land \text{card}(\llbracket [E] \nu(\rho) \rrbracket ) = 1 \} = \mathcal{E}

### 6.2.5 First guarded bunch axiom

\( g \Rightarrow g \rightarrow E = E \)

To validate we show:

\( [g]^{\tau}(\mathcal{E}) \subseteq [g \rightarrow E = E]^{\tau}(\mathcal{E}) \)

= by set expansion of LHS of \( \subseteq \) and applying equality semantics to RHS.

\( \{ \rho \mid \rho \in \mathcal{E} \land [g]^{\tau}(\rho) = \{ \rho \} \} \subseteq \\
\{ \rho \mid \rho \in \mathcal{E} \land [g \rightarrow E]^{\nu}(\rho) = \llbracket [E]^{\nu}(\rho) \rrbracket \} \)

= by a property of set comprehension

\( \rho \in \mathcal{E} \land [g]^{\tau}\{\rho\} = \{\rho\} \Rightarrow \rho \in \mathcal{E} \land [g \rightarrow E]^{\nu}(\rho) = \llbracket [E]^{\nu}(\rho) \rrbracket \)

We must prove the RHS of the implication under assumptions \( \rho \in \mathcal{E} \) and \( [g]^{\tau}\{\rho\} = \{\rho\} \)

By semantics of the guarded bunch, the RHS becomes:

\( \rho \in \mathcal{E} \land \text{if } [g]^{\tau}\{\rho\} = \{\rho\} \text{ then } [E]^{\nu}(\rho) = \llbracket [E]^{\nu}(\rho) \rrbracket \) \text{ else } \{\} \text{ end } = [E]^{\nu}(\rho) \)

now \( \rho \in \mathcal{E} \) is already assumed and using assumption \( [g]^{\tau}\{\rho\} = \{\rho\} \) to resolve the conditional statement it remains to prove:

\( [E]^{\nu}(\rho) = \llbracket [E]^{\nu}(\rho) \rrbracket \)

Which is true by the equality law we inherit from predicate logic with equality.

### 6.2.6 Second guarded bunch axiom

\( \neg g \Rightarrow g \rightarrow E = \sim\{\} \)

To validate we show:

\( [\neg g]^{\tau}(\mathcal{E}) \subseteq [\neg g \rightarrow E = \sim\{\}]^{\tau}(\mathcal{E}) \)

= “by applying semantics of negation to LHS of \( \subseteq \) and equality semantics to RHS”

\( \{ \rho \mid \rho \in \mathcal{E} \land [g]^{\tau}\{\rho\} = \{\} \} \subseteq \\
\{ \rho \mid \rho \in \mathcal{E} \land [\neg g \rightarrow E]^{\nu}(\rho) = \llbracket [\sim\{\}]^{\nu}(\rho) \rrbracket \} \)

= “by property of set comprehension and semantics of unpacking”

\( \rho \in \mathcal{E} \land [g]^{\tau}\{\rho\} = \{\} \Rightarrow \rho \in \mathcal{E} \land [g \rightarrow E]^{\nu}(\rho) = \{\} \)
We must prove the RHS of the implication under assumptions
\( \rho \in E \) and \( \llbracket g \rrbracket^\tau \{ \rho \} = \{ \} \)

By semantics of the guarded bunch, the RHS becomes:
\[ \rho \in E \land \text{if } \llbracket g \rrbracket^\tau \{ \rho \} = \{ \rho \} \text{ then } \llbracket E \rrbracket^\nu (\rho) \text{ else } \{ \} \text{ end } = \{ \} \]

now \( \rho \in E \) is already assumed and using assumption \( \llbracket g \rrbracket^\tau \{ \rho \} = \{ \} \) to resolve
the conditional statement it remains to prove:
\[ \{ \} = \{ \} \text{ which is true by equality.} \]

6.3 Establishing the soundness of our logic

The soundness of a sequent \( P \vdash Q \) is established by demonstrating the cor-
responding semantic entailment \( P \models Q \) which holds when \( \llbracket P \rrbracket^\tau (E) \subseteq \llbracket Q \rrbracket^\tau (E) \)

The soundness of a proof rule:
\[
\begin{array}{c}
HYP \vdash P \\
\hline
HYP \vdash Q
\end{array}
\]

is established by demonstrating \( (HYP \models P) \Rightarrow (HYP \models Q) \) which is equiva-
lent to \( \llbracket P \rrbracket^\tau (E) \subseteq \llbracket Q \rrbracket^\tau (E) \)

Other proof rules can be massaged into this form bearing in mind that the
sequent \( P, Q \vdash R \) is equivalent to \( P \vdash Q \Rightarrow R \) and that where a proof rule
has as its premises the two sequents \( P \vdash Q \) and \( P \vdash R \) these can be replaced
by the single premise \( P \vdash Q \land R \).

35

6.3.1 The soundness of the exists-intro rule of FOPLN

Our rule is:
\[
\begin{array}{c}
HYP \vdash \delta(E), \ HYP \vdash P[E/X] \\
\hline
HYP \vdash \exists x \bullet P
\end{array}
\]

\( \exists \)-intro

By the arguments given above it will be sufficient to prove:
\[ \delta(E)^T(\mathcal{E}) \cap [P[E/x]^T(\mathcal{E}) \subseteq \exists x \bullet P]^T(\mathcal{E}) \]

which by the definitions of $[\delta(E)]^T(\mathcal{E})$, $[P[E/x]^T(\mathcal{E})$ and $[\exists x \bullet P]^T(\mathcal{E})$

is equivalent to

\[
\{ \rho \mid \rho \in \mathcal{E} \land \text{card}(\mathcal{E}) = 1 \} \cup \{ \rho \mid \rho \in \mathcal{E} \land (\exists x' \bullet [P]^T(\{\rho \oplus (x \sim x')\}) = 1 \}
\]

extracting predicates

\[
\rho \in \mathcal{E} \land \text{card}(\mathcal{E}) = 1 \land [P]^T(\{\rho \oplus (x \sim [E]^\rho)\}) = 1
\]

which we can abstract as the form $a \land b \land c \Rightarrow a \land d$

and noting from logic that $(a \land b \land c \Rightarrow a \land d \Leftrightarrow (a \Rightarrow (b \land c \Rightarrow d))$

we can write our proof goal as:

\[
\rho \in \mathcal{E} \Rightarrow (\text{card}(\mathcal{E}) = 1 \land [P]^T(\{\rho \oplus (x \sim x')\}) = 1
\]

\[
(\exists x' \bullet [P]^T(\{\rho \oplus (x \sim x')\}) = 1)
\]

which, since $b \Rightarrow (a \Rightarrow b)$ is implied by

\[
\text{card}(\mathcal{E}) = 1 \land [P]^T(\{\rho \oplus (x \sim [E]^\rho)\}) = 1
\]

\[
(\exists x' \bullet [P]^T(\{\rho \oplus (x \sim x')\}) = 1)
\]

We now appeal to the one point rule: $(\exists x \bullet x = E \land P) \Leftrightarrow P[E/x]$ with $x' = \text{choice}(\mathcal{E})$. Since from LHS of our implication we have $\text{card}(\mathcal{E}) = 1$ we have $\{x'\} = [E]^\rho$ and thus:

\[
\text{card}(\mathcal{E}) = 1 \land [P]^T(\{\rho \oplus (x \sim [E]^\rho)\}) = 1
\]

which is true by logic.

### 6.3.2 The soundness of the exists-elimination rule of FOPLN

Our rule is:

\[
\frac{\text{HYP}, \delta(\alpha) \vdash P[\alpha/x] \Rightarrow Q, \text{HYP}, \delta(\alpha) \vdash \exists x \bullet P}{\text{HYP} \vdash Q} \quad \alpha \text{ fresh, } x \notin Q \quad \exists\text{-elim}
\]

We will manipulate our rule into the best form for validation. First note that since we have $\alpha$ fresh we know $\alpha \notin P$ which renders the $\delta(\alpha)$ in the second premise redundant \(^{13}\). The $\delta(\alpha)$ in the first premise can be moved across the turnstile to appear on the LHS of an implication. We can furthermore combine the two premises as a conjunction, and rewrite our inference rule as:

\[
\frac{\text{HYP} \vdash (\delta(\alpha) \Rightarrow (P[\alpha/x] \Rightarrow Q)) \land (\exists x \bullet P)}{\text{HYP} \vdash Q} \quad \alpha \text{ fresh, } x \notin Q
\]

\(^{13}\)Our motivation for including this redundant term is to provide symmetry when using the rule in inverted form for tree proofs.
\neg p \lor q \text{ can be written as:}

\[
\text{HYP} \vdash (\neg (\delta(\alpha) \land (P[\alpha/x]) \lor Q)) \land (\exists x \cdot P) \quad \alpha \text{ fresh, } x \not\in Q
\]

and to validate we show:

\[
[\neg (\delta(\alpha) \land P[\alpha/x]) \lor Q) \land (\exists x \cdot P)] 
\subseteq [Q].
\]

which, recalling the semantic rules

\[
\begin{align*}
[\neg P]^{\ast}(\mathcal{E}) &= \mathcal{E} \setminus [P]^{\ast}(\mathcal{E}) \\
[P \land Q]^{\ast}(\mathcal{E}) &= [P]^{\ast}(\mathcal{E}) \cap [Q]^{\ast}(\mathcal{E}) \\
[P \lor Q]^{\ast}(\mathcal{E}) &= [P]^{\ast}(\mathcal{E}) \cup [Q]^{\ast}(\mathcal{E})
\end{align*}
\]

we can express as:

\[
(\mathcal{E} \setminus [\delta(\alpha) \land P[\alpha/x]]) \cup [Q]^{\ast}(\mathcal{E})) \cap [(\exists x \cdot P)] 
\subseteq [Q].
\]

We first derive two lemmas.

Applying the semantic rule for \(\delta(E)\) to \(\delta(\alpha)\) gives:

\[
[\delta(\alpha)]^{\ast}(\mathcal{E}) = \{\rho \mid \rho \in \mathcal{E} \land \text{card}(\{\alpha\})^{\rho} = 1\}
\]

Noting that from set theory \(\text{card}(s) = 1 \iff (\exists x \cdot s = \{x\})\) we can write this equiviently as:

\[
[\delta(\alpha)]^{\ast}(\mathcal{E}) = \{\rho \mid \rho \in \mathcal{E} \land (\exists x' \cdot [\alpha]\rho = \{x'\})\}
\]

But in our proof rule \(\alpha\) is an element, and we thus have:

\[
[\delta(\alpha)]^{\ast}(\mathcal{E}) = \mathcal{E}
\]

and therefore \(\mathcal{E} = \{\rho \mid \rho \in \mathcal{E} \land (\exists x' \cdot [\alpha]\rho = \{x'\})\}\)

from which we can conclude:

Lemma 1. \(\exists x' \cdot [\alpha]\rho = \{x'\}\)

Which we will use in the proof of:

Lemma 2. \([\delta(\alpha) \land P[\alpha/x]](E) = [\exists x \cdot P]^{\ast}(\mathcal{E})\)

Proof of lemma 2.

We recall the following semantic rules:

\[
\begin{align*}
[P[\mathcal{E}/x]]^{\ast}(\mathcal{E}) &= \{\rho \mid \rho \in \mathcal{E} \land [P]\{\rho \oplus \{x \leadsto [\mathcal{E}]\}\}\} \\
[\exists x \cdot P]^{\ast}(\mathcal{E}) &= \{\rho \mid \rho \in \mathcal{E} \land (\exists x' \cdot [P]\{\rho \oplus \{x \leadsto \{x'\}\}\} \neq \{\}\}\}
\end{align*}
\]

Thus, along with the derived semantics for \(\delta(\alpha)\) given above, we have:
\[ \delta(\alpha) \land P[\alpha/x][\mathcal{E}] = \{ \rho \mid \rho \in \mathcal{E} \land (\exists \, x' \bullet \llbracket \alpha \rrbracket \rho = \{ x' \}) \land [\mathcal{P}]^\tau((\rho \oplus \llbracket \alpha \rrbracket \nu(\rho))) \neq \{ \} \} \]

= “extending the scope of \( x' \) to the right and writing \( \{ x' \} \) for \( \llbracket \alpha \rrbracket \nu(\rho) \)"

\[ \{ \rho \mid \rho \in \mathcal{E} \land (\exists \, x' \bullet \llbracket \alpha \rrbracket \rho = \{ x' \}) \land [\mathcal{P}]^\tau((\rho \oplus \llbracket \alpha \rrbracket \nu(\rho))) \neq \{ \} \} \]

= “see denotational rule for \( \exists \, x \bullet P \) quoted above”

\[ [\exists \, x \bullet P]^\tau(\mathcal{E}) \]

Which completes the proof of lemma 2.

Returning to the validation of the \( \exists \)-elim rule, we recall that we have to show:

\( ((\mathcal{E} \setminus [\delta(\alpha) \land P[\alpha/x]][\mathcal{E}]) \cup [\exists \, x \bullet P]^\tau(\mathcal{E})) \cap [\exists \, x \bullet P]^\tau(\mathcal{E}) \subseteq [\exists \, x \bullet P]^\tau(\mathcal{E}) \)

we now abstract with \( \mathcal{E} \) as \( A \), \( [\delta(\alpha) \land P[\alpha/x]][\mathcal{E}] \) as \( B \), \( [\exists \, x \bullet P]^\tau(\mathcal{E}) \) as \( B' \) and \( [\exists \, x \bullet P]^\tau(\mathcal{E}) \) as \( C \).

We must show \(( (A \setminus B) \cup C) \cap B' \subseteq C \)

i.e. by distribution of set intersection over set union:

\( ((A \setminus B) \cap B') \cup (C \cap B') \subseteq C \)

But from lemma 2 we have \( B = B' \) and from set theory we have:

\( B = B' \Rightarrow (A \setminus B') \cap B = \{ \} \)

Thus our task becomes to show

\( \{ \} \cup (C \cap B') \subseteq C \)

Which follows directly.

### 7 Conservative extensions to bunch notation

So far the only notation that distinguishes our axiomatic bunch theory from the axiomatic set theory on which it is based, is the unpacking operator \( \sim \).

However, we can now extend our notations through a series of definitions. We assume standard definitions from set theory, e.g. for set union, intersection, and difference. We will also use set comprehension expressions of the form \( \{ x \mid P \bullet E \} \), representing the set of values expression \( E \) can take as \( x \) ranges over the values allowed by predicate \( P \).

We define the \( \text{null} \) bunch, bunch union and intersection, and bunch inclusion:

\[ \text{null} \triangleq \sim \{ \} \]

\[ E \cup F \triangleq \sim(\{ E \} \cup \{ F \}) \]

\[ E \cap F \triangleq \sim(\{ E \} \cap \{ F \}) \]

\[ E : F \triangleq \{ E \} \subseteq \{ F \} \]
Now conditional expressions:

\[
\text{if } g \text{ then } E \text{ else } F \text{ end } \triangleq g \rightarrow E, \neg g \rightarrow F
\]

And a form of bunch comprehension:

\[
\{ x \mid x \in S \land P \bullet E \} \triangleq \{ x \mid x \in S \land P \rightarrow E \}
\]

Since \( y \in \{ x \mid x \in S \land P \bullet E \} \iff \exists x \bullet x \in S \land P \land y : E \) we can write \( \{ x \mid x \in S \land P \bullet E \} \) as \( \{ x \mid \text{true} \bullet x \in S \land P \rightarrow E \} \) which we abbreviate to \( \{ x \bullet x \in S \land P \rightarrow E \} \). Also, since \( x \in S \land P \rightarrow E \) is an expression, we can take \( \{ x \bullet E \} \) as a general form for set comprehension, with the proviso that \( E \) must imply the type of \( x \). Corresponding to this we can use \( \{ x \mid E \} \) as a general form for bunch comprehension.

We now come to some redefinitions, required to lift existing definitions, which apply only to elements, to apply element-wise to bunches.

We incur a proof obligation to show the new definitions will be equivalent to the old when applied to arguments for which the old definitions were defined, i.e. to elements. To avoid need for renaming this obligation can be discharged prior to the redefinition. Thus, noting that when \( E \) and \( S \) are elements:

\[
\{ E \} \subseteq \bigcap \{ S \} \iff E \in S
\]

we are now free to introduce the following redefinition of \( E \in S \), in which \( E \) and \( S \) are not restricted to being elements, because we know it is compatible with the existing definition in the case where \( E \) and \( S \) are elements.

\[
E \in S \triangleq \{ E \} \subseteq \bigcap \{ S \}
\]

The effect of this definition is that for \( E \in S \) to hold every \( e : E \) must be a member of every \( s : S \). This is in line with our use of bunches to represent non-deterministic terms, and guarantees that membership will hold after the non-determinism is resolved.

Here is a redefinition which lifts unpacking operators. The new definition of \( \sim \) is expressed in terms of the original \( \sim \), which appears in the body of the original definition.

\[
\sim E \triangleq \sim(\cup\{ E \})
\]

A more general way of lifting an operation, and one where the accompanying proof obligation can be safely assumed to be discharged, is to represent it in terms of a bunch comprehension. Here is the definition of the lifted power set operator:

\[
\mathbb{P}(S) \triangleq \{ x \bullet s : S \rightarrow \mathbb{P}(s) \}
\]
8 Some results in bunch theory

8.1 Bound variables in bunch theory range over elements

This result follows directly from FOPLN, the underlying logic for our bunch theory [SDM22].

8.2 Properties of bunch union and bunch intersection

The following are immediately provable by appealing to the definitions of null, bunch intersection, and bunch union.

\[
\begin{align*}
E \cdot F &= F \cdot E & E, F &= F, E & \text{commutativity} \\
(E \cdot F) \cdot G &= E \cdot (F \cdot G) & E, (F, G) &= (E, F), G & \text{associativity} \\
E \cdot (F, G) &= (E \cdot F), (E \cdot G) & E, (F, G) &= (E, F) \cdot (E, G) & \text{distributivity} \\
E : E, F &= E : F & E \cdot F : E & \text{inclusion} \\
E, null &= E & E \cdot null &= null & \text{zero and unit} \\
E, E &= E & E \cdot E &= E & \text{idempotence}
\end{align*}
\]

8.3 Equality properties

The set equality axiom \( \forall x \cdot x \in A \iff x \in B \Rightarrow A = B \) now requires the proviso that \( A \) and \( B \) are elements, but we have the following laws for equality of general bunch expressions \( E \) and \( F \)

\[
\begin{align*}
\forall x \cdot x : E \iff x : F & \Rightarrow E = F \\
\{E\} = \{F\} & \iff E = F
\end{align*}
\]

8.4 Properties of guarded bunches

The guarded bunch has distributive properties which can be obviously proved by case analysis:

\[
\begin{align*}
g \rightarrow E, F &= (g \rightarrow E), (g \rightarrow F) \\
g \rightarrow E \cdot F &= (g \rightarrow E) \cdot (g \rightarrow F)
\end{align*}
\]

8.5 Generalisation of certain laws

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

\[
\begin{align*}
x : E, F & \iff x : E \land x : F \ (x \text{ an element}) \\
x \in EF & \iff x \in E \land x \in F
\end{align*}
\]

Which generalise to the following bunch comprehension laws:

\[
\begin{align*}
x : \{y \mid y : Y \rightarrow E \} & \iff \exists y \cdot y : Y \land x : E \ (x \text{ an element}) \\
x \in \{y \mid y : Y \rightarrow E\} & \iff \forall y \cdot y : Y \Rightarrow x \in E
\end{align*}
\]
Some of the set theory axioms can be generalised. The ordered pair axiom $E \mapsto F \in (s \times t) \iff E \in s \land F \in t$ holds for non-elemental values, and this has already been validated in our model.

The power set axiom $s \in P(t) \iff (\forall x \bullet x \in s \Rightarrow x \in t)$ where $x$ is not free in $s$ or $t$, still holds when $t$ is a non-elemental value.

9 Axioms and semantics of “Improper bunch theory”

For modelling program semantics within a total correctness framework we associate with each type $T$ an ‘improper’ bunch $\perp_T$. We consider it to be of type $T$. We require the following additional axioms.

$E : \perp$ “maximality”

$\perp : A, B \Rightarrow A = \perp \lor B = \perp$ “atomicity”

$\{\perp_T\} = \perp_{P(T)}$ “improper packaging”

$\sim \perp_{P(T)} = \perp_T$ “improper unpackaging”

And we need to modify the axiom of proper bunch theory that says $\{E\}$ is an element, which we can now assert only when $E \neq \perp$

$E \neq \perp \Rightarrow \delta(\{E\})$ “guarded element”

The maximality axiom tells us that for any type, any bunch other than the improper bunch is a proper sub-bunch of the improper bunch.

The atomicity axiom tells us that $\perp$ cannot be constructed from the union of two bunches both different from itself.

The packaging axioms tell us that impropriety cannot be removed by packaging or unpackaging, although the results of packing and unpacking an improper bunch will be of the expected type.

Note that we no longer have an unguarded axiom to say $\{E\}$ is an element, but the atomicity of $\{E\}$ follows from the atomicity axiom together with the guarded element axiom.

9.1 Extending our model to include the improper bunch

Within our semantic universe we distinguish sets constructed directly from $BIG$, which we term “basic” or “given” sets, from those defined by using the power-set and set product constructors.

We recall once more that any such given set is maximal and corresponds to a
type, as do all sets constructed from maximal sets.

The semantic support for a basic set, \( T \), say, in our language of improper bunch theory, is given by a set \( T' \) in the meta-language of our semantic universe which contains the elements of the set \( T \), along with an additional distinguished element \( \kappa_T \).

We define the semantic representation of \( \perp_T \) by \( \llbracket \perp_T \rrbracket = T' \)

Thus if the given set \( T \) is defined so that \( T = \{ a, b \} \) then to support this within our semantic universe we have a set \( T' = \{ a, b, \kappa_T \} \).

The denotations of all possible bunches of type \( T \) are as follows:

\[
\begin{align*}
\llbracket \text{null}_T \rrbracket &= \{ \}, \\
\llbracket a \rrbracket &= \{ a \}, \\
\llbracket b \rrbracket &= \{ b \}, \\
\llbracket a, b \rrbracket &= \{ a, b \}, \\
\llbracket \perp_T \rrbracket &= \{ a, b, \kappa_T \}
\end{align*}
\]

We note that \( \perp_T \) is the only term of type \( T \) whose denotation contains the distinguished element \( \kappa_T \). This allows us to model the atomicity of the improper bunch. We also note that the denotation of the improper bunch for a given type is a maximal set. This allows us to model the maximality of the improper bunch.

We use similar techniques for any constructed type.

Within each maximal constructed set of the semantic universe, we have a \( \kappa \) element defined structurally as: \( \kappa_T \times U = \kappa_T \mapsto \kappa_U \) and \( \kappa_P(T) = \llbracket \perp_T \rrbracket \).

Denotations for the improper bunch of each type are:

\[
\begin{align*}
\llbracket \perp_T \rrbracket &= T' (T \text{ a given set}) \\
\llbracket \perp_T \times U \rrbracket &= \llbracket \perp_T \rrbracket \times \llbracket U \rrbracket \\
\llbracket \perp_P(T) \rrbracket &= \mathcal{P}(\llbracket \perp_T \rrbracket)
\end{align*}
\]

Thus, the denotation for the improper bunch of any type will be a maximal set.

We recall that the purpose of the improper bunch is to represent the effect of running a program outside its pre-condition. In line with the philosophy of total correctness, we take a pessimistic view of such effects, which may go well beyond non-termination, and include, for example, stack and array overflows which corrupt pointers or code, leading to totally unpredictable results - the car may crash! To reflect this we want the improper bunch to be absorptive with respect to all expression connectives. We extend a number of definitions to take account of special cases arising from this property. We can do this using bunch guard. In each of the following definitions, the definition body is a bunch of terms, the first of which is a guarded bunch. If the bunch guard is true, that term, and hence the whole body of the definition, represents an improper bunch. Otherwise the guarded term represents null and disappears.

\[
\begin{align*}
\{ E \} & \equiv E = \perp_T \rightarrow \perp_P(T) , \{ E \} \\
\sim E & \equiv E = \perp_P(T) \rightarrow \perp_T , \sim E \\
A \cup B & \equiv A = \perp \lor B = \perp \rightarrow \perp , A \cup B
\end{align*}
\]

\[15\] It is possible to write denotations for these forms directly, rather than using redefinition. For example \( \llbracket \{ E \} \rrbracket \rho = \text{if } \llbracket E \rrbracket \rho = \llbracket \perp_T \rrbracket \text{ then } \llbracket \perp_P(T) \rrbracket \text{ else } \{ \llbracket E \rrbracket \rho \} \) \text{ end }.
\[ A \cap B \models A = \bot \lor B = \bot \rightarrow \bot, \ A \cap B \]
\[ D \mapsto E \models D = \bot \lor E = \bot \rightarrow \bot, \ D \mapsto E \]
\[ S \times T \models S = \bot \lor T = \bot \rightarrow \bot, \ S \times T \]

For bunch union and bunch intersection we can deduce absorptive properties from the above, e.g.,

\[ A, \bot T = \text{“by definition of bunch union”} \]
\[ \sim \{A\} \cup \{\bot T\} = \text{“by improper packaging axiom”} \]
\[ \sim \{A\} \cup \bot P(T) = \text{“by guarded redefinition of set union”} \]
\[ \sim \bot P(T) = \text{“by improper unpackaging axiom”} \]

One construct that is not subject to these absorptive properties is the bunch guard, since the second guarded bunch axiom tells us that \( \text{false} \mapsto E = \text{null} \).

This is to capture the effect, in the world of values, of code that cannot run, e.g., the contribution from the branch of a conditional statement that is not executed.

### 9.2 Validation of the axioms of improper bunch theory

Let \( D(T) \) be the set of denotations for bunches of type \( T \). We formalise the properties, already discussed above, that support maximality and atomicity of the improper bunch.

**Property.** \( x \in D(T) \Rightarrow x \subseteq [\bot T] \)

To validate of the maximality of the improper bunch, \( E : \bot \), we must show \( [E : \bot] \tau (E) = E \). We need the property, already discussed above \( x \in D(T) \Rightarrow x \subseteq [\bot T] \)

\[ [E : \bot] \tau (E) = E \quad \text{“by semantics of bunch inclusion”} \]
\[ \{ \rho \mid \rho \in E \land [E] \tau (\rho) \subseteq [\bot] \} = E \quad \text{“for arbitrary } \rho \text{”} \]
\[ [E] \tau (\rho) \subseteq [\bot] \quad \text{“by property given above”} \]
\[ \text{true} \]

Validation of the axioms for improper packaging, improper unpacking, and guarded element follow immediately from the revised semantics of packaging.

For a fuller treatment which includes validation of the atomicity of the improper bunch see [SZD20]

### 10 Related work

The main material on bunch theory and its applications is found in Hehner’s book “A practical theory of programming” [Heh93], which uses a predicative style for program descriptions. Bunches are used to incorporate non-determinism in functions by Morris and Bunkeburg [MB01]. They propose a model that axiomatises boolean and function types in the presence of non-
determinism and considers the adaptation of logic to bunches of truth values. This contrasts with our approach which provides a full integration with set theory and isolates the use of bunches from logic by distinguishing truth semantics from value semantics, so we can continue to use classical predicate logic along with bunches.

An early application of bunches was to formal grammars, with a language being a bunch. In “The functional treatment of parsing” by René Leermakers [Lee93] (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 non-deterministic to deterministic algorithms, and that the use of bunches allows the simple expression of formulae which “blow up” if expressed in set theory. We illustrated the same point in section 3.5

11 Conclusions

Set Theory is foundational in mathematics. A set is a packaged collection, and when packaging and collecting are separated, we obtain Bunch Theory. This is a radically different theory with significant advantages.

In the area of “descriptions”, traditional approaches give rise to the possibility of undefined or non-denoting terms, e.g. when a partial function is applied outside its domain. This is a problem that has generated an enormous amount of discussion, from the early approaches of Russell and Frege, to the treatment of undefined terms in specification languages. The essential problem is that no approach (apart from Bunch Theory) is able to include “nothing” as a conceptual object. In our theory, undefined terms still denote, and are handled comfortably.

Bunches, along with the convention of “lifting” most operations and function application, naturally represent non-deterministic terms. We illustrate this by describing a semantics of “prospective values”, in which \( S \diamond E \) represents the value or values expression \( E \) will take after program \( S \) has executed. The simplicity of the semantics we derive above depends on the preservation of homogeneity when non-determinism is introduced: the type of \( S \diamond E \) is the same as the type of \( E \). This is essential to obtain the simplicity of the rule

\[
S; T \diamond E = S \diamond (T \diamond E).
\]

Our other illustrations show how bunch theory can provide continuations, preferential choice that takes account of the feasibility of continuations in a backtracking context, and the integration of probabilistic choice. We introduce an “improper bunch”, which is maximally non-deterministic, to represent failed computations. Later in the paper, after “wholistic” function application is introduced, we use bunch theory to represent, in a notationally pleasant way, a

\[16\] If we used sets instead of bunches to record the possible values of \( E \) after \( S \). Then \( S \diamond E \) would be a set, and \( S \diamond T \diamond E \) would be a set of sets, and our rule would need to incorporate generalised set union to remove the build up of unwanted structure.
formal grammar, composed, mathematically, of simultaneous fixed point equations which are shown to have solutions.

Our formal description of bunch theory is constructed by adding packing and unpacking operations to a typed set theory. We use a theory, taken from the B method, which distinguishes predicates, which are subject to proof, from expressions, which are subject to evaluated. This separation means that adding bunches to our theory leaves the underlying classical logic unchanged. This contrasts with the approaches of Hehner and Morris, who both have to contend with additional truth values in their logics.

We provide and validate a model for our theory. This allows us to see that the axiomatisation of bunches is orthogonal to the existing axiomatisation of set theory. Our method could be used to add bunches to other axiomatisations of set theory.

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 current 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.

References

[Abri96] J-R Abrial. The B Book. Cambridge University Press, 1996.

[AM02] J-R Abrial and W J Mussat. On Using Conditional Definitions in Formal Theories. In Bert D, Bowen J, Henson M, and Robinson K, editors, ZB2002, number 2272 in Lecture Notes in Computer Science, 2002.

[Bac81] R J R Back. Proving total correctness of nondeterministic programs in infinitary logic. Acta Informatica, 15, 1981.

[BCJ84] H Barringer, J H Chang, and C B Jones. A logic covering undefinedness in program proofs. Acta Informatica 21, 1984.

[Flo67] R W Floyd. Non-deterministic Algorithms. Journal of the ACM, 14(4), 1967.

[Gri96] D Gries. Foundations for calculational logic. In M Broy and B Schieder, editors, Mathematical Methods in Program Development (NATO ASI Series), volume 158. Springer, 1996.

[Heh93] E C R Hehner. A Practical Theory of Programming. Springer Verlag, 1993. Latest version available on-line.
[Heh06] E R Hehner. Retrospective and Prospective for Unifying Theories of Programming. In S. E. Dunne and W Stoddart, editors, *UTP2006 The First International Symposium on Unifying Theories of Programming*, number 4010 in Lecture Notes in Computer Science, 2006.

[HJ98] C A R Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[JLS10] C B Jones, M J Lovert, and L J Steggles. Semantic Models for a Logic of Partial Functions. Technical Report CS-TR-1220, Newcastle University, 2010.

[Lee93] René Leermakers. *The functional treatment of parsing*. Springer Verlag, 1993. 2nd edition 2012.

[MB01] J Morris and A Bunkenburg. A theory of bunches. *Acta Informatica*, 37(8), 2001.

[Mor88] C Morgan. The specification statement. *ACM Transactions on Programming Systems*, 10(3), 1988.

[Mor90] C Morgan. *Programming from Specifications*. Prentice Hall International, 1990.

[MRG88] C Morgan, K Robinson, and P Gardiner. *On the Refinement Calculus*. Oxford University Computing Laboratory, 1988. Programming Research Group Monograph PRG-70.

[Nel89] G Nelson. A Generalization of Dijkstra’s Calculus. *ACM Transactions on Programming Languages and Systems*, Vol 11, No. 4, 1989.

[NH92] T S Norvell and E C R Hehner. Logical specifications for functional programs. In R S Bird, C C Morgan, and J C P Woodcock, editors, *Mathematics of Program Construction*. Springer Verlag, 1992.

[Nic95] John Nicholls, editor. *Z Base Standard, Version 1.2*. Oxford University Computing Laboratory PRG, 11 Keble Rd, Oxford OX1 3QD, 1995.

[Rus05] Bertrand Russell. On denoting. *MIND*, 14:479–493, 1905.

[SDG99] W. J. Stoddart, S. E. Dunne, and A. J. Galloway. Undefined Expressions and Logic in Z and B. *Formal Methods in System Design*, 15(3):201–215, November 1999.

[SDM22] W J Stoddart, S Dunne, and Chunyan Mu. Much ado about nothing, logic for bunch theory and undefinedness. arXiv, 2022.

[SZ13] W J Stoddart and F Zeyda. A Unification of Probabilistic Choice within a Design-based Model of Reversible Computation. *Formal Aspect of Computing*, 2013.

[SZD10] W J Stoddart, F Zeyda, and S Dunne. Preference and nondeterministic choice. In *ICTAC’10. Proceedings of the 7th International colloquium conference on Theoretical aspects of computing*, volume 6255 of *LNCS*, 2010.
Appendices

A Prospective values in a fixed point treatment of loops

We have an iteration construct

\textbf{while } g \textbf{ do } S \textbf{ end } , which we will refer to as W.

We define W recursively by unwrapping the first iteration of the loop:

\[ W = \text{if } g \text{ then } S \text{ ; } W \text{ else skip end} \]

and for our prospective value approach we are interested in obtaining a recursive equation which we can eventually use to establish the existence of \( W \diamond E \), the bunch of prospective values for expression \( E \) following \( W \).

We first establish the monotonicity property, namely \( E : F \Rightarrow (S \diamond E) : (S \diamond F) \), which will hold for any program \( S \) defined using the semantic primitives of section 3.4.4. We start from the monotonicity property of WP semantics that for any program \( S \) and predicates \( P, Q \), that \( (P \Rightarrow Q) \Rightarrow ([S]P \Rightarrow [S]Q) \). This can be re-expressed in CWP terms as \( (P \Rightarrow Q) \Rightarrow ([S]P \Rightarrow [S]Q) \). i.e. if \( S \) might establish \( P \) and \( P \Rightarrow Q \), then \( S \) might establish \( Q \). Now let \( E \) and \( F \) be bunch expressions such that \( E : F \). Let \( z \) be an arbitrary element of the correct type, then \( z : E \Rightarrow z : F \) and thus using the monotonicity property for CWP we have \( (z : E \Rightarrow z : F) \Rightarrow ([S]z : E \Rightarrow [S]z : F) \). We can now argue as follows:
Let us first establish, therefore, that a recursive equation in bunch expressions will nevertheless give us the basis to call on Tarski’s fixed point theorem for complete lattices, namely:

**Theorem** If the elements of a set \( P \) form a complete lattice under some partial order and \( F \in P \to P \) is monotonic wrt to that order, then the equation \( X = F(X) \) has a minimum solution (a minimum fixed point).

We must show we can call on this theorem when given a recursive equation in bunch expressions, \( U = E \), where \( U \) occurs as a sub-expression in \( E \), without variable capture, and \( E \) is monotonic increasing wrt \( U \) and bunch inclusion ordering. Under the assumption \( \sim \{ U \} = U \) We can rewrite our equation as a fixed point equation in set expressions

\[
\{ U \} = \{ E[\sim \{ U \} / U] \}
\]

or writing \( X \) for \( \{ U \} \) and \( D \) for \( \{ E \} \) we obtain the following equation in which \( D \) will be monotonic increasing wrt \( X \) and inverse set inclusion ordering:

\[
X = D[\sim X / U]
\]

Which using lambda abstraction we can write as:

\[
X = (\lambda Y \cdot D[\sim Y / U])X
\]

Giving us a classical fixed point equation with a set inclusion ordering that forms a complete lattice, having the form \( X = F(X) \) where \( F \in P(T) \to P(T) \) for some \( T \) with the associated partial order \( (P(T), \subseteq) \) which is a complete lattice.

The above conversion fails, however, when the assumption \( \sim \{ U \} = U \) does not hold, and this will occur when \( U = \bot_T \), since \( \{ \bot_T \} = \bot_{P(T)} \). We will still be able to provide a full analysis of \( W \circ E \), but we will need to first introduce a weaker prospective value semantics in which the improper bunch does not occur, because the effect of a non-terminating computation is to yield the prospective value \( \text{All} \), i.e. the most non-deterministic proper bunch, rather than \( \bot \). We will call this “liberal prospective value” or LPV, semantics and it will have the same discriminating power as weakest liberal pre-condition (WLP) semantics: neither is able to discriminate between a non-terminating computation and a maximally non-deterministic computation that does terminate.

Let us write the LPV value of expression \( E \) after computation \( S \) as \( S \circ_0 E \). The rules for \( S \circ_0 E \) over our program connectives are identical to those for \( S \circ E \) with the single exception of the pre-conditioned computation \( P \mid S \), where we
have
\[ P \mid S \circ_0 E = \neg P \rightarrow All, \ S \circ_0 E \]
i.e. a computation carried out outside its precondition has the prospective value \( All \).

We now return to our analysis of the loop \( W \), which we defined as
\[ W = \text{if } g \text{ then } S ; W \text{ else } \text{skip} \text{ end} \]
so in terms of LPV semantics
\[ W \circ_0 E = \text{if } g \text{ then } S ; W \text{ else } \text{skip} \circ_0 E \]
Applying our definition for the \( \text{if} \) construct this becomes:
\[
W \circ_0 E = g \Rightarrow S ; W \cap \neg g \Rightarrow \text{skip} \circ_0 E
\]
= "applying rule for choice"
\[
(g \Rightarrow S ; W \circ_0 E) , (\neg g \Rightarrow \text{skip} \circ_0 E)
\]
= "applying rule for sequential composition"
\[
(g \Rightarrow S \circ_0 W \circ_0 E) , (\neg g \Rightarrow \text{skip} \circ_0 E)
\]
= "applying rules for guard and skip"
\[
(g \rightarrow S \circ_0 W \circ_0 E) , (\neg g \rightarrow E)
\]
Which, noting that \( \circ_0 \) is right associative, gives us a recursive equation in the bunch expression \( W \circ_0 E \):
\[
W \circ_0 E = (g \rightarrow S \circ_0 W \circ_0 E) , (\neg g \rightarrow E)
\]
and we can conclude that this equation has a minimum solution for \( W \circ_0 E \)

Aside: For the non-terminating loop \( \text{while true do skip end} \) our equation for \( W \circ_0 E \) is:
\[
W \circ_0 E = (true \rightarrow \text{skip} \circ_0 W \circ_0 E) , false \rightarrow E
\]
"which reduces to"
\[
W \circ_0 E = W \circ_0 E
\]
an equation which has every value as a fixed point, and in particular has the bunch \( All \) as its least fixed point. End Aside

It remains to obtain the value of \( W \circ E \), i.e. the prospective value for \( E \) after \( W \) rather than its liberal prospective value. For this we note that although LPV cannot always distinguish between non-terminating computations and terminating computations that are maximally non-deterministic, it can do so for feasible loops, since a terminating loop will leave the loop guard false, and thus cannot be maximally non-deterministic. This enables us to express \( W \circ E \) in terms of \( W \circ_0 E \) as:
\[
W \circ E = (W \circ_0 E) = All \rightarrow \bot , (W \circ_0 E)
\]
B Precedence and associativity

The following symbols are right associative: \( \bullet \Rightarrow \diamond \nabla \equiv \). All other binary connectives are left associative.

In descending order, with connectives of equal precedence included within brackets, e.g. \((+ \ mod)\), the order of precedence is:

- function application (symbol elided)
- \((+ \ mod)\)
- \((\ast / \ mod)\)
- \((\cap \cup \oplus)\)
- \((>, <, \leq, \geq)\)
- \((=, \neq, :\subseteq, \not\in)\)
- \((\land, \lor, \Rightarrow, \equiv)\)
- \(\bullet\) as used in quantifications
- \(\neg\)
- \(\Rightarrow\)
- \(\equiv\)
- \(\equiv\)

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

- \(\sim, \neg\)

The precedence level of \(\bullet\), together with the definition of \(\forall\) and \(\exists\) as unary prefixes, defines the scope of quantified variables in predicates.