BUNCH THEORY: A SIMPLE SET THEORY FOR COMPUTER SCIENCE

Eric C.R. HEHNER
Computer Systems Research Group, University of Toronto, Toronto M5S 1A1, Canada

Received April 1980; revised version received September 1980

Formal languages, programming languages, set theory, grammars

1. Introduction

Perhaps because Set Theory is so useful in Mathematics, Computer Science has adopted it for some purposes. Sets appear explicitly in some programming languages (e.g. SETL, Pascal), and very commonly in reasoning about programs. Sometimes sets appear disguised in other notations. For example, in Pascal, an alternative of a **case** statement is labelled by a set of selector values (here, set notation would perhaps be too cumbersome). A programming language is a set of programs, but we do not specify a language in set notation; instead we use a grammar in the form of rewriting rules.

Set Theory in its entirety is more powerful than necessary for most of Computer Science. For example, though a set theorist may define the integer 2 as $\{\emptyset, \{\emptyset\}\}$, we do not view it as such. We treat it as a primitive; we do not think of its members, or form a union or intersection of 2 with anything. In the other direction, we have little use for sets of uncountable cardinalities.

We may, of course, use only those parts of Set Theory that are useful to us, and ignore the rest. But the notation of Set Theory, designed for power we don't want, is sufficiently cumbersome that we have tended instead to invent many special notations where a weaker set theory would have served well. These dissatisfactions lead us to propose a set theory that has the right power for Computer Science, and that is notationally more convenient for our purposes than mathematical Set Theory. The new theory, called Bunch Theory, does not exclude Set Theory; when appropriate, we can form sets of bunches, or bunches of sets.

2. Bunch Theory

Corresponding to a set, such as

$\{1, 3, 6\}$

which contains the three integer elements 1, 3 and 6, we introduce the bunch, written without curly braces, i.e.

$1, 3, 6$

which contains the same three integer elements. We shall give bunch axioms in a moment, but we already can see a consequence of the notation: an element, and a bunch containing only that element, are indistinguishable. More generally, a bunch built from other bunches does not retain the structure, as does a set whose elements are sets. This corresponds well to Pascal, in which sets of sets are illegal (though possibly more from implementation than from programming reasons). It does not correspond well to the definition of the SETL language (sets of sets are legal), but it does correspond to its use. In graph theory, one often partitions a graph into sets of nodes, but it is doubtful whether any algorithm requires the ability to specify sets of sets of nodes. Examples may be taken from throughout Computer Science.

We now give a definition of bunch (bunch axioms):

(1) (*existence*) There exist bunches that are considered 'elementary'. Each such bunch is called an 'element'.

(2) (*specification*) A predicate p on the elements specifies the bunch consisting of those elements for which the predicate is true. This bunch is denoted

$x \$ p(x)$.

(The dollar sign may be thought of as an 's' overstruck with a 't', and is pronounced 'such that'.)

(3) (*union*) If a and b are bunches, then

a, b

is the bunch containing the elements of both a and b.

There are no other bunches.

For comparison with Set Theory, we list set axioms (according to one axiomatization):

(*set axioms*)

(1) (*existence*) There exists a set.

(2) (*specification*) Given a set a, a predicate p on the elements of a specifies the set consisting of those elements for which the predicate is true. This set is denoted

$\{x \in a \mid p(x)\}$.

(3) (*union*) If a and b are sets, then

$a \cup b$

is the set containing the elements of both a and b.

(4) (*pairing*) If a and b are sets, then

$\{a, b\}$

is the set whose elements are the sets a and b.

(5) (*power*) If a is a set, then

2^a

is the set whose elements are the subsets of a.

There are no other sets.

In Set Theory, we can construct the null set $\emptyset$ from the one set whose existence is postulated, call it z, as follows:

$\emptyset = \{x \in z \mid \text{false}\}$.

Though we did not know the elements (nor even the cardinality) of z, we do know those of $\emptyset$, and from it we construct all other sets whose elements are known. The game in Set Theory is to begin with almost nothing (not even the null set), and build sets of unlimited cardinalities.

As stated in the Introduction, our game is different. We may begin, for example, with the boolean values true and false, or with the rational numbers, or with character strings, or with a mixture of values of different types. From this beginning we can construct the null bunch

null = x \$ false

and the universal bunch

universe = x \$ true

and bunches that are 'in between'. In Set Theory, postulating the universal set is inconsistent, leading to paradoxes. But the universal bunch causes no problems, and for this reason our specification axiom is simpler. In a sense, we begin by choosing our universe. In Computer Science there is little use for sets of uncountable cardinality, yet they are defined implicitly by the set axioms. By choosing a countable universe of elements, no bunches of uncountable cardinality can be constructed.

3. Names and operations

As in Set Theory, we introduce the relations $\in$ (element of), $\subseteq$ (sub-bunch of), and = (equal to). For element x and bunch a, $x \in a$ is defined by looking separately at the three possible ways in which a was constructed:

(1) If a is an element then $x \in a$ means $x = a$.

(2) If a is y \$ p(y) then $x \in a$ means p(x).

(3) If a is b, c then $x \in a$ means $x \in b \lor x \in c$.

For bunches a and b, $a \subseteq b$ is defined as $(\forall x)\ x \in a \Rightarrow x \in b$ and $a = b$ is defined as $a \subseteq b \land b \subseteq a$. When the left operand is an element, the relations $\in$ and $\subseteq$ coincide. For equality, the order and multiplicity of elements is irrelevant; the theory of ordered multibunches (analogous to ordered multisets, i.e. sequences) will not be pursued in this paper.

To give a name to a bunch, we shall write the name, followed by a colon, followed by the bunch. For example,

a: 1, 3, 6

means that wherever a occurs, it stands for 1, 3, 6. With this definition, writing

b: 6, a, 4

is equivalent to writing

b: 6, 1, 3, 6, 4

which in turn is equivalent to writing

b: 1, 3, 4, 6.

Note: By this notation, we mean what would be written more typically in the style of mathematics texts as 'let a = 1, 3, 6'. We shall assume that free substitution of value for name can be made, avoiding the problem of clashing with bound variable names by choosing all names to be distinct.

If $\circ$ is a binary operation on the elements of our universe, define $X \circ Y$ for bunches X and Y as

$$X \circ Y = z \ \$ \ (\exists x)(\exists y)\ x \in X \wedge y \in Y \wedge z = x \circ y .$$

For example,

$$(1, 2) + (10, 20) = 11, 12, 21, 22,$$

$$(1, 2) + 10 = 11, 12,$$

$$1 + 1 = 2 .$$

Parentheses are needed in the first two examples assuming '+' has higher precedence than ','. In the last example, the ambiguity as to whether we are adding bunches or integers is no more bothersome than the ambiguity as to whether we are adding integers or rationals.

The following defines a bunch containing the natural numbers:

natno: 0, natno + 1 .

By this definition, the name natno stands for '0, natno + 1' wherever it occurs, including in the definition itself. The bunch contains (all and only) the elements seen by repeatedly substituting '0, natno + 1' for natno in the definition. Provably, natno is the smallest bunch satisfying the equation

$$X = 0, X + 1,$$

i.e. the least fixed point of

$$f(X) = 0, X + 1 .$$

Primarily for the similarity to the next paragraph, we give one more example of an operation on bunches that is a natural extension of (or reduces properly to) the 'same' operation on numbers. For any bunch of numbers X, let

$$X^0 = 1, \qquad X^{n+1} = X^n * X .$$

Then, $X^1 = X$, and for larger n, $X^n = X * X * \cdots * X$ (n-fold multiplication) as expected.

Consider now the universe of character strings. Let λ denote the null string, and if s and t are strings, let st denote the concatenation of s and t. For any bunch of strings S, let

$$S^0 = \lambda, \qquad S^{n+1} = S^n S .$$

Also, let

$$S^? = \lambda, S,$$

$$S^* = \lambda, S^* S,$$

$$S^+ = S, S^+ S,$$

Then,

$$S^? = S^0, S^1,$$

$$S^* = S^0, S^1, S^2, \ldots$$

$$S^+ = S^1, S^2, S^3, \ldots$$

4. Applications

4.1. Within a programming language

We now suggest a few of the possible uses of bunch within an ALGOL-like programming language.

In addition to the notations of the specification and union axioms, it seems useful to introduce the notation i **to** j, for suitable integer expressions i and j, to mean the bunch $k \ \$ \ i \leqslant k \leqslant j$, that is, $i, i + 1, \ldots, j - 1, j$. Now, perhaps with limitations, bunches are useful in iteration control, as case statement labels, as nameable and assignable values, as type specifications, and possibly in other ways.

A constant definition, such as

pi: 3.14

becomes just a special case of the naming notation introduced in the previous section. One may wish to prohibit recursion, or to include it by taking the 'lazy evaluation' approach to implementation. A definition such as

indices: 1 **to** 4

serves the purposes that in Pascal require the two distinct but similar notations

const indexset = **set**(1, 2, 3, 4)

and

type indextype = 1 .. 4.

With our definition, we may declare variable i so that it may be assigned any element of the bunch indices thus:

i :∈ indices.

Following this declaration, the assignment

i := 2

is legal. Also with this definition of indices, we may declare

s :⊆ indices,

so that variable s may be assigned any subbunch of indices. Then

s := null, s := 1, 3, s := indices

are all legal.

By failing to distinguish a type from a value, i.e., by using bunch notation for both purposes, we lose none of the benefits of so-called 'strong-typing', and we gain the following important benefit: where we want to pass a type as a parameter, we can do so without inventing 'generic' procedures or inventing the second-class type 'type'. One may wish to make the restriction that only certain bunches, e.g. manifestly contiguous bunches, are allowable in certain contexts, e.g. to the right of :∈ and :⊆. This restriction is in a redundant part of the language, so the ability to detect errors is restricted but the ability to express the computation is not.

A programming language that is not in the ALGOL mould can be designed by taking the bunch as its central notion. The state of the computation is a bunch; the computation proceeds by operations on this bunch. The initial state is either the null or the universal bunch; the final state is 'the answer'.

4.2. As a language description

A language can be described as a bunch program of strings, using definitions such as the following:

```
program:     declaration* statement*
declaration: identifier (':', ':∈', ':⊆') expression ';'
statement:   variable ':=' expression ';',
             'if' expression 'then' statement*
                 ('else' statement*)? 'fi'
expression:  constant,
             variable,
             expression ('+', '–', '*', '/') expression,
             identifier '$' expression,
             expression ',' expression,
             expression 'to' expression.
```

This is obviously a fragment of a grammar in a notation that is not novel. It is increasingly common to put quotes on terminals (rather than angle brackets on nonterminals), so that terminals can be distinguished from metasymbols and so that blank spaces can be described (if desired). It is also common to include a notation for 'zero or more of' (usually curly brackets rather than '*'), and optional (usually square brackets rather than '?'). There are, however, differences between our approach and the usual grammatical approach. To show the differences, we must first outline the usual grammatical approach.

A context-free grammar (CFG) is defined as:

(a) a set V_T of terminals, plus
(b) a set V_N of nonterminals, including
(c) a distinguished nonterminal $S \in V_N$, plus
(d) a set of rewriting rules.

For set V, let V^* be the 'Kleene star' of V, i.e. the set of all finite concatenations (strings) of elements of V (for bunch V, V^* is exactly analogous). Then, each rewriting rule is of the form $A \to \alpha$, where $A \in V_N$ and $\alpha \in (V_N \cup V_T)^*$. The notation

$$A \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$$

is introduced as a shorthand for the n rules

$$A \to \alpha_1$$

$$A \to \alpha_2$$

$$\vdots$$

$$A \to \alpha_n$$

The rewriting rules induce a relation between strings as follows. If $A \to \alpha$ is a rule, then $\beta A \gamma \to \beta \alpha \gamma$ for any $\beta, \gamma \in (V_N \cup V_T)^*$. Let $\to^*$ denote the reflexive transitive closure of $\to$. Then α is a program if $S \to^* \alpha$ and $\alpha \in V_T^*$.

Using the rewriting rule $A \to \alpha$, we can rewrite the string $\beta A \gamma$ as $\beta \alpha \gamma$. This is similar to the mathematical practice of substituting for variable x in formula f(x) a formula e to obtain f(e), which is called an instance of f(x). Indeed nonterminals are often called 'syntactic variables' (and terminals 'syntactic constants'), and $\beta\alpha\gamma$ an 'instance' of $\beta A\gamma$. But in the CFG formalism, contrary to mathematical practice, the substitution is *not* systematic. For example, in the string expression '+' expression, we do not necessarily substitute the same string for the two occurrences of expression. Mathematical practice requires that the *same* new formula e be substituted for *all* original occurrences of variable x in f.

In bunch notation, the definitions for program, expression, etc., mean that these names stand for the entire expression to the right of the colon. Whereas in the CFG formalism, a substitution (rewriting) is a (non-systematic) instantiation, which may lose information, here it is merely a replacement of a name by its definition, losing no information. Rather than invent a special concept (rewriting rules), we rely on the standard mathematical practice that allows us, with definitions

circum: 2 ∗ pi ∗ r,   pi: 3.14,

to write

circum = 2 ∗ 3.14 ∗ r.

By *systematic* substitution and simplification, all elements of program are constructed (perhaps the names programs, expressions, etc., would be more appropriate).

In the CFG formalism, a nonterminal is an elementary symbol. We must repeatedly rewrite a string to eliminate nonterminals, and therefore the (reflexive) transitive closure $\rightarrow^*$ is defined. In bunch notation, a name, such as program or expression, is *not* an elementary symbol. We cannot form, nor is there a need to form, a bunch corresponding to the set V_N. The relation that corresponds to $\rightarrow^*$ is simply the subbunch relation $\subseteq$. For example, with the definition

$A: \alpha_1, \alpha_2, ..., \alpha_n,$

we have, for each i,

$\alpha_i \subseteq A,$

and, for any bunches β and γ,

$\beta\alpha_i\gamma \subseteq \beta A\gamma$

precisely because we do not treat the symbol A differently from what it stands for.

To extend the CFG formalism to allow the right side of a rewriting rule to be a regular expression, we must introduce α^*, where α is a regular expression, and metaparentheses, and extend the meaning of rewriting rules. This new ∗ is annoyingly similar to, but not identical to, the Kleene star (α is a regular expression whereas V is a set). The entire exercise is more complicated than necessary. Bunch theory uses fewer definitions to accomplish the same purpose, and very few that are specific to this purpose. No contribution to parsing theory is claimed here. We have merely restated the parsing problem as the design of a bunch membership algorithm.

5. Conclusion

As research in Computer Science proceeds, we make new definitions, invent new notations, and coin new terms at a dizzying speed. The research can hardly be said to proceed in a straight line, from first principles to a desired end. Rarely does one piece of research extend directly from another; usually it projects from the middle of the previous research at an angle. Only parts of the previous research remain useful. At the same time, at another place, others invent other rats' nests of definitions, notations, and terms that span the same space. And all of these remain in use even though they overlap and are only partly appropriate for our current state.

From time to time it is worthwhile to draw the straight line from first principles to present location. Of course, any new line may be viewed as a contribution to the mess, and some people may be disappointed that this paper (in some sense) does not cover any new ground. That is not its intention. We hope that the simplicity (beauty), and generality (usefulness) of our definitions and notations will make them a welcome replacement for a large number of others.

Acknowledgment

I thank Brad Silverberg for his comments on a rough draft, the members of IFIP Working Group 2.3 for giving these ideas a hearing, and Bill McKeeman for constructive criticism.