Defining hereditarily finite sets as words of a formal language, PRELIMINARY.

Ulrich Mutze, www.ulrichmutze.de

July 29, 2013

1 Introduction

Set theory is a formal system in which the relation $\in$ is the fundamental notion. What makes X and Y sets [1] is that $X \in Y$ is true or false, independent of our means to figure this out in the given case.

We are far apart from what one could conceive as an ideal state of affairs: The definitions of X and Y provide data from which an algorithm allows to compute the truth-value of $X \in Y$ and of $Y \in X$.

Interestingly there is a type of sets for which just this state of affairs holds true. These are the *hereditarily finite sets*, which are finite in a recursive manner, i.e. their elements are also finite sets (which would not be the case for the finite set $\{\mathbb{Z}, \mathbb{R}\}$), as are the elements of the elements, and so forth. Further, they are of finite depth: for any hereditarily finite set S there is $n \in \mathbb{N}$ such that all chains of type

$$s_p \in s_{p-1} \in ... \in s_2 \in s_1 \in S$$

satisfy $p \leq n$. Since we will be working with hereditarily finite sets all the time, we need a short name and choose *hf-sets*.

For general set theory the operations such as union, section, and powerset formation are stipulated axiomatically (or derived from axiomatically stipulated operations). Since for a given hf-set we know the elements and the elements of the elements and so forth, it is clear that for hf-sets these operations can be defined by finite constructive processes. Assume, for instance, that for each of two hf-sets a, b we have a list of their elements. Then it is a finite process to create the list of elements of $a \cup b$.

Can one be sure that such finite constructions can be done for all the set operations that in Fraenkel-Zermelo set theory are introduced by axioms?

The most useful way to clarify this point is to implement all these operations in a programming language and test their properties with innumerous random-generated input objects in a computer run. I succeeded in doing so by defining in the programming

[1] the distinction between sets and classes is here ignored as immaterial

language C++ a class *Set*, which defines all set operations that appear in the Fraenkel-Zermelo axioms and much more. To indicate the scope of *Set*, here is a list of some of its more advanced functions: functions that test whether a set is a Kuratowski-Wiener pair, a relation, a function, a number, a numeric function, a group, and a function that represents any set as a pixel graphics written to file (to escape the size limitations of computer displays).

The definition of a mathematical concept in an object oriented programming language (such as C++) requires to solve a problem, which is mathematical in nature and, as I see it, is relevant also outside the field of computer science: One has to represent the *state* of an object in a way that knowing the state for any of the objects under consideration, allows us to infer the state of all objects resulting from operations or functions defined for these objects. This general requirement can be satisfied by defining a suitable *data structure* and to implement these operations and functions as algorithms that accept instances of this data structure as input and either modify the input or output a suitable new instance of this structure.

Defining data structures in programs can work only if some fundamental data structures, fundamental types, as they are often called, are provided as part of the programming language. In C++ we have many fundamental types that come in various sizes to satisfy practical needs. The mathematical concepts that correspond to these types include {true, false}, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$; in addition one has 8-bit-characters. Further for any fundamental type one may form arrays of arbitrary length the components of which store instances of this fundamental type.

For most physical systems the state can be described by a list of real numbers [2]. To have a simple example, consider homogeneous thermodynamic systems consisting of ideal mono-atomic gas in tanks. The state of such a system can be described by three $\mathbb{R}$-valued data: volume V, temperature T, number of moles N. Obviously all three numbers have to be positive for meaningful system states. Using objects in object oriented programming parallels using objects in a laboratory. Before they can be used in an experiment, they have to be acquired. For programmed objects this happens by executing a formative function (named constructor) which is programmed such that a valid state results from the computation.

This suggests a flexible and convenient scheme for defining object states: First, fix a raw structure as some aggregate of simply-typed data, $\mathbb{R}^3$ in our example. Second, define a predicate on the raw structure that singles out the valid states. In programming, a variation of this scheme is more convenient: The constructor does not only flag raw states as valid or not valid, but enforces validity by changing the raw state in a suitable way if necessary. In our example the constructor after being given $\mathbb{R}$-typed data v, $temp$, n as input could create a state with

$$V = |v| \,, \quad T = |temp| \,, \quad N = |n|$$

and would issue a warning if any of the input values was negative.

[2] We assume that a coherent system of units, such as SI, is selected and that the kind of physical quantity is encoded in the position of the number in the list.

The same logic can be found in the definition of formal languages: the words of the language are lists of characters from a specified alphabet and when the list is fed as input into some 'word validation algorithm' and the result 'true' or 'false' says whether the list is a word of the language or not. Notice that for alphabets that include 'blank', or any other separator character, a whole piece of language, such as a book or a program, can be considered a single word which contains also what from a different viewpoint would be considered 'context' and would let parts of the algorithm appear 'context-dependent'.

These considerations aimed at putting into perspective the state definition for hf-sets as it emerged from the development of class *Set*: Here the raw data are character strings over an alphabet of only two characters. In the implementation of *Set* these are chosen as 'a' and 'z', whereas in the following presentation they will be written for compelling reasons as '{' and '}'. Especially it is to be noted that there is no separator character and so to a single hf-set there corresponds a single contiguous string of braces which encodes the state of the hf-set and thus everything that can be meaningfully be said about this particular set. The overwhelming majority of strings of braces cannot be decoded to hf-sets. They are garbage. Non-garbage strings have for each opening brace downstream a closing counterpart. But this is far from sufficient to qualify a string of braces as decodeable to a hf-set. Describing the algorithm which does either signals garbage or gives the list of elements of a hf-list will be the largest part of the present work.

2 Defining basic data types

From now on we change our position quite radically. We no longer know sets and numbers and rely only on proven 'finitary' constructive methods, as close to 'common sense' as possible. Since set formation is normally written with braces { and } and also the omnipresent empty set $\emptyset$ can reasonably be written as {}, it is clear that braces are indispensable for writing down hf-sets. As we will see, more than these two symbols will not be needed and so we will formalize the character encoding of hf-sets so that to every hf-set there corresponds a uniquely determined string of two or more braces.

Using the usual description of syntactic categories in Backus-Naur Form we define

$$\begin{aligned} \langle b\text{-}letter\rangle &::= \ \{ \mid \} \\ \langle b\text{-}word\rangle &::= \langle b\text{-}letter\rangle \mid \langle b\text{-}word\rangle\langle b\text{-}letter\rangle \ . \end{aligned} \tag{1}$$

Here we have expressed the selection of a specific alphabet of braces by the name prefix 'b -'. Notice that here the definition of a b-word deviates from the usual definition of a word over the alphabet of b-letters in that a b-word is never empty. According to our agreement, '{' is the first letter and '}' the last one; we write { < } to express this. This defines the *lexicographic order* for b-words according to which for instance

$$\{\}\{\}\{<\ \{\}\}\}\{<\ \{\}\}\}\}\ <\}\ <\}\{\}\}\}\ . \tag{2}$$

The b-words that encode hf-sets will be called *valid*. The rules that govern validity can hardly be formulated without an elementary concept of natural numbers. This will now

be presented in a form that is as simple as possible for the present purpose. It is here not meant to form the basis for arithmetics, although an algorithmic definition of all arithmetic operations can obviously be given on this basis. From the present point of view it looks more natural to develop arithmetics on the basis of hf-sets being available.

$$\begin{aligned} \langle number \rangle &::= \ 0 \mid \langle positive\ number \rangle \\ \langle positive\ number \rangle &::= \ / \mid \langle positive\ number \rangle / \end{aligned} \tag{3}$$

Thus $0, /, //, ///$ is a list of numbers which is commonly written as $0, 1, 2, 3$. The only operations we need for numbers are increment and decrement for which we use the postfix operation symbols $++$ and $--$ employed in many programming languages. Of course, increment is by appending a slash; only for incrementing 0 a slight modification of this rule is necessary:

$$\langle positive\ number \rangle ++ := \langle positive\ number \rangle / \ , \quad 0 ++ := / \ . \tag{4}$$

Correspondingly decrement is by taking away a slash. Only decrementing / and 0 one needs separate rules.

$$\langle positive\ number \rangle / -- := \langle positive\ number \rangle \ , \quad / -- := 0 \ , \quad 0 -- := nil \ . \tag{5}$$

Notice that here the normal defining equality sign := is used since defining terms is not a matter for Backus Naur notation. Here *nil* is an exception indicator like the value 'NAN' (not a number) in many computer systems. Sometimes it is convenient to consider *nil* as a number

$$\langle extended\ number \rangle \ ::= \ nil \mid \langle number \rangle \tag{6}$$

and to define the operation $++$ and $--$ for all extended numbers by setting

$$nil ++ \ := \ nil \ , \quad nil -- \ := \ nil \ . \tag{7}$$

3 Defining validity of b-words

We begin with defining a function f which parses a b-word w into b-words that encode its elements. The function thus will return one or more b-words, or the result 0, or the result *nil*. The definition will be carried out with the help of an example. Consider

$$w = \{\{\{\{\}\}\{\}\}\{\}\{\{\{\}\}\}\} \tag{8}$$

Then we insert levels as numbers (which, for convenience, we write in the usual style, not in terms of slashes).

$$0\{1\{2\{3\{4\{5\}4\}3\}2\{3\{4\}3\}2\{3\}2\}1\{2\{3\{4\{5\}4\}3\}2\}1\}0$$

The rule for inserting the numbers is obvious: We start with level 0 left to the first brace. An opening brace increases the level and a closing brace decreases it. When in following this rule somewhere one has to decrease 0 and thus obtains *nil* the whole procedure

stops and returns *nil*. The same is true if the last number differs from 0 or if 0 appears at any other place than the beginning and the end.

Since the present example avoided the stop conditions so far, we continue by these steps:

1. purge the first and the last number, as well as the first and last brace. If then no braces are left, the function returns 0 (and the following two steps thus will be omitted).
2. purge all numbers different from 1
3. Put the contiguous groups of braces which are separated by 1s into a list conserving their relative order.

Of course the first step gives

$$1\{2\{3\{4\{5\}4\}3\}2\{3\{4\}3\}2\{3\}2\}1\{2\{3\{4\{5\}4\}3\}2\}1$$

and the second step gives

$$1\{\{\{\{\}\}\}\{\{\}\}\{\}\}1\{\{\{\{\}\}\}\}1$$

The last step finally gives

$$\begin{array}{l}\{\{\{\{\}\}\}\{\{\}\}\{\}\} \\ \{\{\{\{\}\}\}\}\end{array} \tag{9}$$

where each list entry is written on a new line and the order in the lines (from above to below) reflects the order of the entries in the list. If this final list is in strict [3] lexicographic order, it is the result of our function, otherwise the result is *nil*. In our case lexicographic order is obviously satisfied as is easily verified when the entries are precisely printed each below its predecessor . (The two entries coincide in their first seven symbols and the next symbol is { in the first entry and } in the second.)

This list gives one or more b-words. The sum of the lengths is less than the length of w (two braces were lost in the first of the three steps listed above.

The validity of w is defined by multiple application of f according to the following algorithm.

```
Task: Decide whether the b-word w is valid. In case of a positive answer, list the elements of w.
Procedure:
  create  a list of b-words, named worklist, in which w is the only entry;
  set the boolean variable firstrun to true;
label AGAIN:
  if the worklist is empty then we are ready. The results are: w is valid and the
  entries of the list xFirst are valid and are the elements of w;
label WORK:
  take the first item v of the worklist;
```

[3] this excludes that adjacent entries are equal

```
apply function f to v and store the result as x
if x equals nil then we are ready. The results are: w is not valid, there are no elements of w.
if x equals 0 then we have a dichotomy:
  if firstrun is true then we are ready. The results are: w is valid and has no elements.
  else take v out of the workpool and go to label AGAIN;
// if we come here, we know that x consists of one or more b-words
if firstrun is true store x as xFirst and set firstrun to false;
take v out of the worklist and append x to the list;
go to label WORK;
```

Of course, two valid b-words represent the same hf-set if they are identical. So it is not necessary to go to the elements and their elements to decide whether two hf-sets are equal.

Although, as we have seen, it may be quite tedious to test a long b-word for validity, it is rather simple to retrieve the elements from a b-word that is known to be valid. Further, if we are given a list of valid b-words it is easy to order them lexicographically (which implies elimination of duplicates) and to concatenate them in this order and to include them into a pair braces. In this way we obtain in that way a hf-set, the elements of which are just the entries of that list.

In particular, given two hf-sets, we can straightforwardly decide whether one of them is an element of the other, a subset or a superset of the other. Further, the union and the section of the two sets are straightforwardly constructed. All is done by inspecting the finite list of elements and making use of the fact that elements are easily and effectively compared (and tested for equality) as character strings (without a decomposition into elements).

That we used the two braces as the symbols that make up our b-words was a convention. Of course one could use 0 and 1 as for general digital data. However, using braces is helpful in reminding us that the intended interpretation of a character string is in terms of hereditarily finite sets.