SEMANTIC DOMAINS

The basic representational technique revolves around the concept of a domain. Actually all of the domains we will look at are simple sets, so for now we will just look a little further at sets of values that we can use in programming languages. Sets can be primitive or compound. Primitive sets are:

- The natural numbers: \( \mathbb{N} \)
- The integers: \( \mathbb{Z} \)
- The real numbers: \( \mathbb{R} \)
- The Booleans: \( \mathbb{B} = \{ \text{true}, \text{false} \} \)
- The characters: \( \mathbb{C} \)
- And perhaps a few others

Compound sets can be formed using the operators product, sum and function:

- \( A \times B \): the set of all pairs of elements from \( A \) and \( B \)
- \( A + B \): the set of all elements from \( A \) and \( B \) tagged with the originating set
- \( A \to B \): a subset of \( A \times B \) where the elements of \( A \) and \( B \) are related in some given way

In addition we will use operators on these sets to help represent operations in the language. Some of these will be special to the particular domain we use; these will be discussed as semantic algebras. Some are more generic:

- to extract the parts of a pair we can use \( \text{fst}(\{x,y\}) = x \) and \( \text{snd}(\{x,y\}) = y \)
- to add a tag to an element of \( A + B \) we will use \( \text{in}A(x) \), where \( x \in A \), and \( \text{in}B(x) \) for \( x \in B \)
- to remove a tag, we will use:
  \[
  \begin{cases}
  \text{cases } (m) \text{ of } \text{is}A(x) \to \ldots x \ldots \text{is}B(y) \to \ldots y \ldots \text{end}
  \end{cases}
  \]
  Here \( m \) is an element of \( A + B \); \( \text{is}A \) and \( \text{is}B \) are tests for the tag on \( m \). If the test succeeds, the tag is removed and the element is bound to the argument of the test operation. The oblong separates the cases. In this binary case there are only two cases. In general there can be many, as in \( A + B + C + \ldots \)

DEFINING A FUNCTION

We will often use a method of defining functions that gives th functionality first and then an equation or equations that defined the mapping intensionally. For instance the addition function on the natural numbers could be defined by:

\[
\text{add: } \mathbb{N} \times \mathbb{N} \to \mathbb{N}
\]
\[
\text{add } (m,n) = m + n
\]

These two lines tell us the name of the function, its functionality (domain and range) and how elements are paired. The + operator is assumed to be already defined.

We can also represent functions as lambda expressions, thus:

\[
\text{add } = \lambda[m,n].m + n
\]

We can simplify this further using a transformation called currying, discovered by Haskell Curry. The sets \( (A \times B) \to C \) and \( A \to (B \to C) \) are isomorphic, i.e. they can be put in on-to-one correspondence. Another way to look at this is that the functions of the first form produce the same answer as the function of the second form when applied to the same elements of \( A \) and \( B \). This means that two argument functions (this can be generalized to multiple arguments) can be rewritten as a sequence of single argument functions, as basic lambda calculus demands. Thus add can be written as \( \lambda m.\lambda n.m + n \) with no loss of generality.
LIFTED DOMAINS

In many programming languages, there is always the possibility of having a syntactically valid program that either makes no sense, produces an error condition, or does not terminate. All of these situations need special handling in a good semantic definition. We need to have the ability of expressing the concepts of “undefined”, “error”, “no result”, “non-termination” and so on. Sometimes we will handle them in a way that suits the language being studied, but, in general, we can handle them all with a special, distinguished value that we will write as ⊥, pronounced “bottom”. When we add this value to any domain, we “lift” the domain: e.g. $A \cup \{\bot\}$, written as $A_\bot$.

A function that has bottom in both domain and range is a lifted function:

- e.g. addTwo: $\mathbb{N}_\bot \rightarrow \mathbb{N}_\bot$
  - $\text{addTwo} = \lambda n. n = \bot \rightarrow \bot \sqcup n + 2$, or
  - $\text{addTwo} = \lambda n. n + 2$, for short

Such a function is strict, because it simply transmits bottom if it receives it – it cannot “correct” an error. A non-strict function will map bottom to some definite value. E.g. $({\lambda x.1}) \bot$ always has the value 1 even when it is given bottom, whereas $({\lambda x.1}) \bot$ is always $\bot$. 
