Even though most of our domains will be simple sets, there is a distinction between, say, the set of natural numbers, $\mathbb{N}$, and the way that such values are handled in programming languages. To express the additional information, we will turn the set into a full-fledged domain by renaming it and adding operations to it that we can then use to support the semantics of real languages. If we do this, we get a semantic algebra. Each operation will either be considered primitive, or we must define it as an equation (or equations) using the functional notations we already know. This gives these algebras the flavor of abstract data-types. Indeed domain theory, which includes semantic algebras, can form the basis of the semantics of data types. As a simple example, take arithmetic on the natural numbers. We can lay out the semantic domain as a set of declarations as follows:

\[ \text{Domain } \text{Nat} = \mathbb{N} \]

\begin{align*}
\text{Operations:} \\
zero : & \text{Nat} \\
\text{one} : & \text{Nat} \\
\vdots & \\
\text{plus} : & \text{Nat} \times \text{Nat} \to \text{Nat} \\
\text{times} : & \text{Nat} \times \text{Nat} \to \text{Nat} \\
\vdots & \\
\end{align*}

The constants zero, one, etc. map onto the members of $\mathbb{N}$, and the functionality of plus and times makes them binary operations. Adding subtraction and division brings a problem. What is three minus five? and what is two div zero? Both of these can be handled in a variety of ways; we could add a special error value to Nat and make it the “result” of semantically meaningless operations; we could lift Nat into $\text{Nat}_\bot$, and use bottom as the error value. Whichever way we choose, we must handle them some way. If a programming language has the problem, then the semantics must reflect this. Our goal of having a static entity as the denotation of the source code can only be met if we solve all of these problems. We could use our knowledge of the axioms of arithmetic to give equational definitions of the arithmetic operations, but that takes us into too much detail. We will often take basic operations on simple domains as primitive to avoid unnecessary complication.

Given this algebra, we can construct expressions based on the operations in the algebra which can be simplified according to definitions of the operations. Thus:

\[ \text{two plus (four times two)} \]

simplifies to ten in two steps according to the meaning of plus and times.

**THE BOOLEAN DOMAIN**

We can define the domain of truth values as:

\[ \text{Domain } \text{Tr} = \mathbb{B} \]

\begin{align*}
\text{Operations:} \\
\text{true} : & \text{Tr} \\
\text{false} : & \text{Tr} \\
\text{not} : & \text{Tr} \to \text{Tr} \\
\text{not(true)} = & \text{false} \\
\text{not(false)} = & \text{true} \\
\text{or} : & \text{Tr} \times \text{Tr} \to \text{Tr} \text{ (equations after the truth table)} \\
\text{(} & \_ \to \_ \_ \text{)} : \text{Tr} \times \text{Tr} \times \text{D} \to \text{D} , \text{ for some given domain } \text{D} \\
\end{align*}
\[(\text{true} \rightarrow m \& n) = m\]
\[(\text{false} \rightarrow m \& n) = n, \text{ for } m, n \in D\]

This last operation is the conditional form previously used to define functions. Here it is properly placed as an operation on the Booleans.

We can now go back to the definition of \(\text{Nat}\) and add some relational operations:

\[\text{equal} : \text{Nat} \times \text{Nat} \rightarrow \text{Tr}\]
\[\text{greater} : \text{Nat} \times \text{Nat} \rightarrow \text{Tr}\]

etc.

Again these are left as primitive, although they could also be defined using arithmetic axioms.

**THE DOMAIN OF LOCATIONS**

When we examine languages with declarations, we will need the idea of a variable’s location, which will give semantics to a machine address. We can define a primitive domain to do this:

**Domain Location**

Operations:

\[\text{first–locn} : \text{Location}\]

\[\text{next–locn} : \text{Location} \rightarrow \text{Location}\]

\[\text{equal–locn} : \text{Location} \times \text{Location} \rightarrow \text{Tr}\]

This will be adequate for stack-based languages; if we need full-blown addresses that map to \(\text{Nat}\), we could do that as well.

**COMPOUND DOMAINS**

Just as sets can be built into compound sets, so can domains be built into compound domains. We have covered these before, in the case of sets, but we will briefly reiterate them here.

The **product** domain builds domains of pairs; the product \(A \times B\), where \(A\) and \(B\) are any two domains consist of pairs \([a, b]\), where \(a \in A\), and \(b \in B\). There are two operations on pairs:

\[\text{fst} : A \times B \rightarrow A\]
\[\text{fst}(a, b) = a\]

\[\text{snd} : A \times B \rightarrow A\]
\[\text{snd}(a, b) = b\]

This can be generalized into and number of domains: \([a_1, a_2, \ldots, a_n] \in A_1 \times A_2 \times \ldots A_n\). Instead of \(\text{fst}\) and \(\text{snd}\) we can use an “nth” operation, written \(\downarrow i\), where \([a_1, a_2, \ldots, a_n] \downarrow i = a_i\), very much like the sequence we defined in the fundamentals section.

The **sum** domain (disjoint union) builds domains where the element of the component domains are tagged, usual with natural numbers, which is the way we defined it earlier. We can define two tagging operations:

\[\text{inA} : A \rightarrow A + B\]
\[\text{inA}(a) = [0, a]\]

\[\text{inB} : B \rightarrow A + B\]
\[\text{inB}(b) = [1, b]\]

To take the tag off, we will use the “cases” form:
cases \(isA(x)\rightarrow \_\) \(isB(y)\rightarrow \_\) end : \((A + B)\times D \times D \rightarrow D\)

cases \(inA(a)\) of \(isA(x)\rightarrow e_1 \_\) \(isB(y)\rightarrow e_2\) end \(= [a \backslash x]e_1\)

cases \(inB(b)\) of \(isA(x)\rightarrow e_1 \_\) \(isB(y)\rightarrow e_2\) end \(= [b \backslash x]e_2\)

The function builder creates a domain \(A \rightarrow B\). Function application is:

\(\_\_\_ : (A \rightarrow B)\times A \rightarrow B\)

\(f(a) = b\), where \(f \in A \rightarrow B\), \(a \in A\), \(b \in B\)

update : \((A \rightarrow B)\times A\times B \rightarrow (A \rightarrow B)\)

update \((f, a, b) = [a \mapsto b] f\)

We can also use the lambda notation::

\((\lambda x.e) a = [a \backslash x]e\)

**EXAMPLE: DYNAMIC ARRAYS**

As a simple example of a compound domain, let us consider the dynamic array. This is an array, which maps the natural number onto the values stored in the array, all of which should be of the same type. If indexing is done on an element which has no value, then an error value should be returned. In this way, we can index the array by any integers and be assured of getting a value back, even if it is the error value.

Then for any domain \(A\) which includes the error value, we can define:

Domain \(Array = Nat \rightarrow A\)

Operations:

\(newarray : Array\)

\(newarray = \lambda n.error\)

\(index : Nat \times Array \rightarrow A\)

\(index(n, a) = a(n)\)

update : \(Nat \times A \times Array \rightarrow Array\)

\(update(n, v, a) = [n \mapsto v] a\)

The update operations is just a function update, since the array is modeled as an array.

Interestingly, we can use the lambda form here as well, as long as we use a well-known result due to Haskell Curry. He proved that there is an isomorphism between \((A \times B) \rightarrow C\), and \(A \rightarrow (B \rightarrow C)\). Basically if we apply the first function to a pair \([a,b]\) taken from \(A \times B\) to produce an element of \(C\), then we can get the same element by applying the second function to \(a\), and the resultant function to \(b\). We can then rewrite the dynamic array operations as single argument lambda functions:

\(index : Nat \rightarrow Array \rightarrow A\)

\(index = \lambda n.\lambda a.a(n)\)

\(update : Nat \rightarrow A \rightarrow Array \rightarrow Array\)

\(update = \lambda n.\lambda v.\lambda a.[n \mapsto v] a\)

We will see such “curried” functions a lot since they stick more closely to the simplicity of the original lambda calculus.