DENOTATIONAL SEMANTICS OF BINARY NUMERALS

We will illustrate the whole method with a trivial example which attaches denotations to binary numerals like 101 and 1011. This follows Chapter 4 of Schmidt’s book. The three stages are:

1. abstract syntax
2. semantic algebras
3. valuation functions

Abstract Syntax of Binary Numerals

A simple BNF grammar will suffice:

\[ B \in \text{Binary-numeral} \]
\[ D \in \text{Binary-digit} \]
\[ B ::= B \ D | D \]
\[ D ::= 0 | 1 \]

This grammar yields parse trees like:

Semantic Algebras for Binary Numerals

Clearly we will map our numerals to the natural numbers. The grammar tells us we can have numerals of any length, so we need an infinite domain to represent all possible numerals. We will also need two operations, addition and multiplication.

Domain \( \text{Nat} = \mathbb{N} \)

Operations:

\[ \text{plus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \]
\[ \text{times} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \]

Although we could express these operations in primitive lambda calculus terms, we will simply appeal to our understanding of standard arithmetic. Note that the operations have been curried.

Valuation functions for Binary Numerals

Our plan is to attach semantic meaning to each of the syntactic forms in the grammar using a function. There will be one function for each syntactic category, overloaded with each of the alternatives given in its rule or rules. Starting with the rule for binary digits, we will use a function \( D \) (we could use any name, but \( D \) is easy to remember) that maps each possible parse tree involving the syntactic category \( D \) to its meaning. There are two possible parse trees with \( D \) as their root:
Using our function $\mathbf{D}$ we can write:

\[
\mathbf{D} \quad (\quad \quad ) = \text{zero}
\]

which says that the meaning attached to the parse tree with $\mathbf{D}$ as its root and 0 as its leaf is the value zero taken from \textit{Nat}. We could use a similar form for \textit{one} attached to the tree with 1 as its leaf. Actually we will write a shorthand version:

\[
\mathbf{D}[0] = \text{zero} \\
\mathbf{D}[1] = \text{one}
\]

which says the same thing. The funny square brackets always surround alternatives taken from the syntactic rule for the category that corresponds to the function name. Thus 0 and 1 are alternatives taken from the rule for $\mathbf{D}$.

We can do the same for the recursive rule for $\mathbf{B}$. Here the trees are:

\[
\mathbf{B} \quad (\quad \quad ) = \mathbf{D} \quad (\quad \quad )
\]

Since the leaves of these trees are not terminal symbols, we must use either other functions or recursion to express their meaning:

\[
\mathbf{B}[\mathbf{D}] = \mathbf{D}[\mathbf{D}]
\]

and for the recursive form, we can use the fact that the digits in a binary numeral increase as powers of 2 each time we add one to the right hand side.

\[
\mathbf{B} \quad (\quad \quad ) = (\mathbf{B} \quad (\quad \quad ) \text{ times two } ) \text{ plus } \mathbf{D} \quad (\quad \quad )
\]

In short form this is

\[
\mathbf{B}[\mathbf{BD}] = (\mathbf{B}[\mathbf{B}] \text{ times two }) \text{ plus } \mathbf{D}[\mathbf{D}]
\]
Here we use the fact that each function produces a value from \( Nat \) and so we can use the semantic algebra operations to calculate the values we need. Each valuation function clearly has a functionality that expresses this. In our case these are:

\[
\begin{align*}
B & : \text{Binary-numeral} \rightarrow Nat \\
D & : \text{Binary-digit} \rightarrow Nat
\end{align*}
\]

clearly expressing the mapping of syntax to semantics.

**Example Derivation**

We can show the mapping of syntax to semantics for any binary numeral. For example the numeral 101.

The derivation proceeds by parsing the numeral top-down until the leaves are reached, and attaching the meaning at each step from the valuation functions.

\[
\begin{align*}
B[101] &= (B[10] \text{\textit{times two}}) \text{plus } D[1] \\
&= (((B[1] \text{\textit{times two}}) \text{plus } D[0]) \text{\textit{times two}}) \text{plus } D[1] \\
&= (((D[1] \text{\textit{times two}}) \text{plus } D[0]) \text{\textit{times two}}) \text{plus } D[1] \\
&= (((\text{\textit{one times two}}) \text{plus zero}) \text{\textit{times two}}) \text{plus } \text{one} \\
&= \text{five}
\end{align*}
\]

We have thus derived the denotation of the binary numeral 101 as \text{five}. Notice that the last step involves the semantic algebra operations. If we had instead left the expression unevaluated we would still have a valid denotation, but one that is more like a program which can be run to produce the final result. These unevaluated expressions can be looked at as “compiled” version of the syntax, expressed in a semantic language (lambda calculus) running on the virtual machine of substitution, which is really the only operation in lambda calculus. This indeed meets the criteria we set ourselves of using a simple abstract language with simple, unambiguous rules of evaluation.