Denotational Semantics for Teaching Lazy Functional Programming

Olaf Chitil

Lehrstuhl für Informatik II, RWTH Aachen, Germany
chitil@informatik.rwth-aachen.de
http://www-i2.informatik.RWTH-Aachen.de/~chitil

Text books explain the meaning of a functional program concretely only by showing how an expression is evaluated. Thus the idea that a functional program defines mathematical functions and that a function is a value is not imparted.

To give a concrete idea of a function as value, we represent it as a table of arguments and results (its graph):

(&&)	False	True
False	False	False
True	False	True

In general, such tables are infinite and the tables of multi-argument functions with large domains and higher-order functions are too complex to visualise even partially. Nonetheless any function can easily be *imagined* as being such a table.

With such tables we can establish by look up that for example the value of the expression

```
even 6 && (4 + 2 > 7)
```

is `False`.

To determine the table described by a (recursive) function definition, we have to evaluate the application of the function to some arguments. For evaluation we *combine* reduction with look up in tables for known functions and primitive functions like `(+)`. I claim that using such a mixture of reduction and table look up is a natural way to understand a program. Alternatively, we can construct the table of a recursive function by table look up alone, if we start with arguments that do not require recursive calls and continue such that we only require table entries that we have already determined. For example, we determine the table of the factorial function

```
fac n
  | n == 0 = 1
  | n > 0  = n * fac (n-1)
```

in the order `fac 0`, `fac 1`, `fac 2`, `fac 3`, ...[1]

I believe that the classical comparison of evaluation strategies is the best introduction to laziness / non-strictness. The lazy evaluation strategy is vital for the efficiency of the data-oriented programming style[2] and it explains how infinite data structures can be handled by the computer. However, it is important not to give students the impression that laziness means giving up the denotational point of view. In practise, the lazy reduction sequence of an expression is too complex for a human to follow. On the other hand, functions can easily be composed.

Whereas it is straightforward to extend tables to cover infinite data structures, our table for `(&&)` lacks an entry for determining that the value of `False && (1 == 1/0)` is `False`. Hence we introduce a third boolean value ⊥ which represents undefinedness and complete the table as follows:

(&&)	False	True	⊥
False	False	False	False
True	False	True	⊥
⊥	⊥	⊥	⊥

For analogous reasons every type contains a value ⊥. Moreover, projections like `fst` and `head` demonstrate why ⊥ may appear anywhere in an algebraic data structure and thus gives rise to many partial values:

	⊥	[]	False:[]	True:[]	⊥:[]	False:⊥	...
head	⊥	⊥	False	True	⊥	False	...

We can use these tables together with tables for `null`, `(||)` and `tail` to construct the table of `and`:

```
and xs = null xs || (head xs && and (tail xs))
```

	⊥	[]	False:[]	True:[]	⊥:[]	False:⊥	...
and	⊥	True	False	True	⊥	False	...

As an aside we note that we can also reduce expressions which contain ⊥. In patterns ⊥ matches only variables and the wild-card _.

I taught several Haskell programming courses for second year university students who are familiar with a (usually imperative) programming language. At the beginning of the course I gave no definition of the meaning of Haskell programs but just pointed out the similarity to mathematical definitions and appealed to the students' intuition. Only when I came to laziness I introduced reduction and reduction strategies. Directly afterwards I explained the use of tables and ⊥.

I believe that tables and ⊥ assist in understanding (lazy) functional programs. They could also be used as a starting point for a formal introduction to denotational semantics.

[1] compare with: Simon Thompson: *Haskell: The Craft of Functional Programming*, 2nd edition, Addison-Wesley, 1999, Section 4.2.

[2] John Hughes: *Why Functional Programming Matters*, Computer Journal 32(2), 1989, pp. 98–107.