Kernel: an application of the formal language theory for system development

Wojciech Fraczak

September 16, 2010

Introduction and motivation

Kernel is a notation for describing computer systems. The notation is both a programing language which can be compiled into a highly optimized machine code, and a specification language used for modeling or verification purposes.

The main features unique to *kernel* notation are:

- the uniform syntax and semantics for printable (flat/non-functional) values as rooted labeled trees;
- its precise mathematical semantics in terms of relations over printable values;
- the operational semantics of *kernel* programs in terms of FIFO systems, i.e., networks of finite automata connected by FIFO channels;
- the main construct of *kernel* notation is the composition of relations and not a "function call";
- interchangeable nature of behavioral and functional aspects of system components in the sense that both are represented by state machines.

The theory of formal languages and automata plays a very important role in the theoretical computer science of today. However in practice of software development, the results of the formal languages theory are too often limited to a the process of "parsing", a small part of compilation, interpretation, or verification of programs. *Kernel* uses the theory of formal languages and automata beyond the process of parsing. In *kernel*, the "values" are seen as rooted labeled trees encoded by strings, and the "functions" are relations represented by (networks of) transducers. All *kernel* constructs correspond to some operations on relations such as composition, intersection, or projection. Such an approach allows us to use techniques and results of formal language theory for every stage of a system development.

We tried to render the book self-contained. Thus, apart from the description of the *kernel* notation, the document can be seen as an introduction to the theory of formal languages and automata.

Chapter 1

Describing infinite objects with finite alphabet

Until what can we count on our fingers?

The theory of formal languages and automata can be seen as a mathematical vehicle for describing infinite objects. The question is how to describe sets (collections of elements such as numbers) in such a way that we could always answer the following three questions.

Does a given element take part of the set? (membership)

Are two sets equal? (equality)

Is one set a subset of the other? (inclusion)

We start by defining a way of writing down an "element". For example, in the usual decimal numerical system we use the ten digits, from zero (0) to nine (9), to describe natural numbers. The very first natural numbers have intuitive meanings for most of us, like zero (0), one (1), two (2), or three (3) — we know how to count fingers. Bigger numbers have just names, such as "ninety-two thousand, three hundred seventy-one" (92371). Even bigger numbers are for us just sequences of digits, e.g., "54527615000298376514238721 3".

It seems acceptable to consider natural numbers just as finite sequences of digits. But natural numbers is more than sequences of digits. For example, even if I do not know how to name "54527615000298376514238721 3", I know that it is "bigger" than "12300983640954", and I even know how to "add" or "multiply" the two numbers by producing a representation of the result. The representation of numbers and the transformations defined of the representations of numbers are the only things I really need.

Let us try to "redo" the job of defining natural numbers, but assuming that we start with three fingers. We will use the *Kernel* notation.

We start with defining our digits, or rather "*three*gits":

```
<type> threegit <is> ['z,'o,'t];
```

The above statment intruduced a new name "`threegit`' which denotes the set (type) of three distinct elements. At the moment these elements are just tokens which differ by their *labels*; labels are just names starting with a quote. An element of type `threegit` can be either `'z`, `'o`, or `'t`.

We may define a *three*git number as follows:

```
<type> thnumber <is>
     [ 'z thnumber, 'o thnumber, 't thnumber, '_ ];
```

The above definition is recursive because the set of all the *three*git numbers is infinit. The definition of type `thnumber` should be read as: an element of type `thnumber` is either token `'z`, `'o`, or `'t` followed by another `thnumber` element, or just token `'_`. For example: `'_`, `'o'_`, `'o'z'_`, etc., are elements of type `thnumber`.

In the above snippet of *Kernel* code we defined two types. The type `digit` is a variant of ten different symbols. The variations are enumerated within square brackets. The type `string_of_digits` is also a variant; either `'empty` or a record `'string` composed of two fields listed withing curly brackets: `'digit` of type `digit` and `'cont` itself of type `string_of_digits`. This type definition is said to be *recursive* since the reference to the type being defined is used within its definition.

A type defines a set of *values*. Type `digit` has ten values and type `string_of_digits` has infinite number of values.

Usually we give ourselves a finite set of atomic symbols, here ten digits, and a method (procedure) to build description of objects (here natural numbers) from those symbols.

Let's start with a small example.

Consider the set of all natural numbers, i.e., one, two, three, ... and so on. Firstly, we need a notation for describing the numbers. Probably the simplest and the most natural encoding of natural numbers is, so called, "unary numerical system". In this notation, number one is represented by one vertical line `|`, two by two vertical lines `||`, three by `|||`, and so on. Thus, any sequence of vertical lines corresponds to a natural number.

1.1 FIFO system for Stack

see Examples/stack.k,pipe.k

1.2 How to encode trees in order to filter them with FIFOs?