

## COMP 617 RAP Seminar, Fall 2006

Presenter: Walid Taha

Scribe: Roumen Kaiabachev

Date: 06 Aug 2006

### 1 Last Lecture

There is friction between type systems and the decidability of a predicate: characterizing if a program is bad or good is undecidable. In verification, it is hard to verify programs with a theorem prover. Verification is more ambitious than type systems in its goal: There is a program and some user-specified properties about it (usually in a different language), and the theorem prover has to determine if the properties hold. Type systems typically won't allow user-specified properties, but we will see some type systems that do this (such as Concoction). In general, the boundary between type systems and verification is closing. Verification insists on producing a proof for a particular issue. Checking types on the other hand is easier than proving or inferring types.

Recall that type systems and typed languages are constructed on top of untyped languages to provide guarantees about program execution (type safety, termination, bounds, thread safety). The construction includes:

- Syntax - described by a context-free grammar in BNF;
- Semantics - defines how and when the various constructs of a language should produce a program behavior; implemented by an interpreter or compiler;
- Type system - build on top of syntax and semantics; described by a context-sensitive grammar

### 2 Hierarchy of Terms and Types of Semantics

There are several possible sets of terms generated by syntax and the type system. In order of inclusion, they are:

- $t_1$  (biggest)- set of terms produced by the context-free grammar. *Context-free* implies that the answer to the question "Is a term valid?" does not depend on a context (such as a stack or a push-down automata).
- $t_{good}$  - set of good programs (undecidable)
- $t_2$  (smallest) - type system; an approximation of the set of good programs. Members of the set are *context sensitive*: If we have two valid terms  $f$  and  $x$ , then we have to check their types to determine if the term  $f x$  is a well-typed program.

Now we explain how to define a semantics. Suppose we have a term  $e_1 + e_2$ . An evaluation of such a term might be given by the mapping of terms ( $T$ ) to numbers ( $N$ ):  $\llbracket e_1 + e_2 \rrbracket \rightarrow \llbracket e_1 \rrbracket + \llbracket e_2 \rrbracket$ . The plus sign on the left is defined by the grammar while the plus sign on the right has the meaning of a mathematical addition. There are two types of semantics. In *denotational* semantics terms are defined as a context-free grammar which is mapped by the semantics to a mathematical set. This semantics is good for showing equivalence of programs: two programs are equivalent if their mathematical semantics (which is a value) is. For example,  $\text{Sem}(e_1 + e_2) = \text{Sem}(e_1) + \text{Sem}(e_2)$  since the set of values defined on the left is equivalent to the set of values defined on the right. In *operational* semantics, values are added to the set of terms by a context-free grammar and the semantics maps terms to terms. This semantics makes it easy to talk about things going wrong. Values can be outside the set of typed programs because they can be defined independently of the type system. An untyped program can evaluate to produce untyped results.

The operational semantics has type  $T \rightarrow T$ , in particular  $V \subseteq T, T \rightarrow V + E$  where  $V$  is the set of values and  $E$  a set of errors. The function  $\rightarrow$  is partial because of non-termination errors (the operational semantics does not always produce a value). In the example `let f(x) = f(x) in f 12` there is no run-time error, but the semantics cannot produce anything sensible.

The *big-step* operational semantics produces a value immediately. The *small-step* operational semantics produces typed terms which eventually converge to a typed value.