COMP 617 RAP Seminar, Fall 2006

Presenter: Walid Taha
Scriber: Angela Zhu

September 18, 2006

1 Last Lecture

We discussed a small language for untyped arithmetic expressions. The language describes Boolean values and natural numbers using only λ and a few basic items like T, F, *zero*. Then operators like *and*, *not*, and other natural numbers can all be expressed using them.

2 Difference between sematic rules and its implementation

In Ch4 we have saw an ML implementation of arithmetic expressions. By doing the homework, we saw how the OCaml code implements the language in different ways according to big-step semantics and small-step semantic rules. Generally speaking, the semantics is a smallest n-nary relation satisfying a set of rules. Rules are usually implemented by some forms that directly express the relation or by some function which follows the rules. The difference between rules and their implementation is tricky.

2.1 Rules for generating syntactic forms

Natural numbers can be defined as either

$$\mathbb{N} = \{0, 0+, 0++, \cdots\}$$

or

$$n \in \mathbb{N} ::= 0 | n + 1$$

The first definition simply states out all the included terms, with no computation; while the second one defines every term based on the previous.

They both follow the rules:

$$\frac{}{0 \in \mathbf{N}}, \quad \frac{n \in \mathbf{N}}{n+ \in \mathbf{N}}$$

In this way, we see that the rules give the definition of natural number, i.e., all the way to compose terms in the set. An implementation of natural numbers is an implementation of the above rules. Each member of the original set N can be constructed from a composition of the rules. Below we show some compositions that produce the first three natural numbers:

$$\mathbb{N} = \{\frac{0 \in \mathbf{N}}{,} \quad \frac{0 \in \mathbf{N}}{0+ \in \mathbf{N}}, \quad \frac{\frac{0 \in \mathbf{N}}{0+ \in \mathbf{N}}}{0++ \in \mathbf{N}}, \cdots\}$$

We call the compositions <u>*derivations*</u> of the natural numbers.

2.2 Rules for operational semantics

Operational semantic rules can be implemented by programming languages to perform semantic derivation which can evaluate the terms. Derivations are usually shaped like trees with ground rules as leafs. Looking at the evaluation rules for untyped arithmetic expressions, we have:

$$\frac{\mathrm{t}_1 \to \mathrm{t}_1'}{if\ \mathrm{t}_1\ then\ \mathrm{t}_2\ else\ \mathrm{t}_3 \to if\ \mathrm{t}_1'\ then\ \mathrm{t}_2\ else\ \mathrm{t}_3} \qquad (\mathrm{E-if})$$

$$\frac{}{iszero(0) \to \mathrm{T}} \qquad (\mathrm{E-iszeroero})$$

$$\cdots$$

Suppose **$** is set that include all the trees of these rules, then how can we decide if an evaluation $\mathrm{t}_1 \to \mathrm{t}_2 \in \$$ or not? For example, to see whether

$$if\ iszero(0)\ then\ \mathrm{T}\ else\ \mathrm{F} \to if\ \mathrm{T}\ then\ \mathrm{T}\ else\ \mathrm{T}$$

is within the set, we can match the evaluation with evaluation rules to see if a tree can be built. Here, since we have

$$\$ = \{\cdots, \frac{\overline{iszero(0) \to \mathrm{T}}}{if\ iszero(0)\ then\ \mathrm{T}\ else\ \mathrm{F} \to if\ \mathrm{T}\ then\ \mathrm{T}\ else\ \mathrm{F}} \cdots\}$$

we say that the evaluation can be performed according to the rules given.

2.3 Soundness and Completeness of implementation

Again, looking at the Ocaml implementation of evaluation rules. we want in OCaml $eval(\mathrm{t}_1) = t_2$ if and only if for the rules we have $\frac{\cdots}{\mathrm{t}_1 \to \mathrm{t}_2}$. The direction for both side of

$$\frac{\cdots}{\mathrm{t}_1 \to \mathrm{t}_2} \quad \Leftrightarrow \quad eval(\mathrm{t}_1) = t_2$$

must be assured. If an implementation meets the "$\Rightarrow$" side, we say that it is *complete*, because when ever there is a rule, there will be a evaluation implement the rule. On the other side, if an implementation meets the "$\Leftarrow$" side, we say that it is *sound*, because what ever the implementation evaluate, there is a rule of it. So completeness means the implementation model "say all the truth", while the soundness means the implementation model "never tell false".

Completeness, at the first glance seemed very powerful, but it might render nasty affects when some behavior in the implementation are not specified in the rules. As it can give arbitrary results which is unpredictable, different implementation could evaluate the same term to different values. This is very common in programming languages. Different compiles or interpreters might implement the same set of rules in different ways, that, although all rules are correctly implemented, for the behavior that is not specified by any rule, different interpreters will have different explanation. For example, if the value of some variable will be changed by some side effects, while the way of changing is not specified in any rules. Then, when executing the programm, the programmer can't expect what will be get.

Soundness, on the other hand, means that if we can show by implementation that some evaluation is performable, then there is a rule supports it. This is good at the point that the implementation never behaves incorrectly. But it also means that the implementation might say something can't be evaluated, while there actually is some rule to do the evaluation.

Take an intuitive example, suppose all the rules we have, if fully and correctly implemented, should be

$$eval(x) = a; \; eval(y) = b$$

Implementation only of completeness might give

$$eval(x) = a; \; eval(y) = b$$

together with

$$eval(z) = c$$

. And Implementation only of soundness might give only

$$eval(x) = a$$

One of the things that is hard to implement is concurrency. It is hard to describe it in a sequential way, while the implementation of programming language have no idea to model the concurrency, and trade-offs between soundness and completeness must be made.

3 Hascell: Smaller gap between language and implementation

Haskell is a pure functional language. That means that nothing is allowed to have a side effect. However this is a bit of a problem if we want to do something involving side effects. One of the unique concepts with Haskell to grasp side effect is *monads*. This encapsule-like structure can allow the programmer create their own monads and decide how effects propagate from one statement to the next, meanwhile prevent the side effect from "leaking" outside the monads, thus make Haskell a flavor when side effects is not expected.

4 Summary

The main topic of today's seminar is the difference between (semantic) rules and its implementation. It is important for understanding how a language is implemented and behavior of the implementation.