Programming Languages and Types

Exercise 10

Yi Dai

January 16, 2013

1 Abstract Syntax vs. Concrete Syntax

1.1 Specification vs. Identification

The grammar

$$
\begin{array}{rrcll}
\text{(expression)} & \mathit{Exp} & ::= & \mathit{Int} & \text{(number)} \\
 & & \mid & \mathit{Exp}\ \mathit{Opr}\ \mathit{Exp} & \text{(compound)} \\
 \\
\text{(operator)} & \mathit{Opr} & ::= & + & \text{(plus)} \\
 & & \mid & - & \text{(munus)} \\
 & & \mid & * & \text{(times)} \\
 & & \mid & / & \text{(divides)}
\end{array}
$$

for *specification* vs. the grammar

$$
\begin{array}{rrcll}
\text{(level-0 expression)} & \mathit{Exp} & ::= & \mathit{Exp}1 & \text{(level-1 expression)} \\
 & & \mid & \mathit{Exp}\ \mathit{LOp}\ \mathit{Exp}1 & \text{(compound level 1)} \\
 \\
\text{(level-1 expression)} & \mathit{Exp}1 & ::= & \mathit{Exp}2 & \text{(level-2 expression)} \\
 & & \mid & \mathit{Exp}1\ \mathit{HOp}\ \mathit{Exp}2 & \text{(compound level 2)} \\
 \\
\text{(level-2 expression)} & \mathit{Exp}2 & ::= & \mathit{Int} & \text{(number)} \\
 & & \mid & (\ \mathit{Exp}\ ) & \text{(parenthesized)} \\
 \\
\text{(lower operator)} & \mathit{LOp} & ::= & + & \text{(plus)} \\
 & & \mid & - & \text{(munus)} \\
 \\
\text{(lower operator)} & \mathit{LOp} & ::= & * & \text{(times)} \\
 & & \mid & / & \text{(divides)}
\end{array}
$$

for *identification*.

Note that the levels in the second grammar indicates the *priorities*. Thus level-0 expressions have the *lowest* priority while level-2 expressions have the *highest* priority.

1.2 Theoretical Formulation

The grammar

$$e \in Exp$$
$$n \in Int$$
$$o \in Opr$$

$$\begin{array}{rrcll}
\text{(expression)} & e & ::= & n & \text{(number)} \\
 & & | & e_1 \; o \; e_2 & \text{(compound)} \\
\\
\text{(operator)} & o & ::= & + & \text{(plus)} \\
 & & | & - & \text{(munus)} \\
 & & | & * & \text{(times)} \\
 & & | & / & \text{(divides)}
\end{array}$$

widely used in theoretical work is only *semi-abstract*.

1.3 Abstractness vs. Concreteness

A *fully* abstract syntax renders the *tree structure* of the expressions using constructors.

$$\begin{array}{rrcll}
\text{(expression)} & Exp & ::= & Num(Int) & \text{(number)} \\
 & & | & Cpd(Opr, Exp, Exp) & \text{(compound)}
\end{array}$$

Such a description can be readily translated into representations in a language that supports algebraic data types, like Scala.

```
sealed abstract class Exp

type Opr = String

case class Num(int : Int) extends Exp
case class Cpd(opr : Opr, lhs : Exp, rhs : Exp) extends Exp
```

Once the abstract syntax for a language is given, its concrete syntax can be freely chosen. It can be prefix notation, infix notation, postfix notation, even English, etc.

2 Inductive Definitions and Rule Induction

2.1 Inductive Definitions

1. Inductively-defined sets: natural numbers, arithmetic expressions, etc.

$$\frac{n \in Int}{n \in Exp}\ \textsc{Num} \qquad \frac{e_1 \in Exp \qquad e_2 \in Exp \qquad o \in Opr}{e_1\ o\ e_2 \in Exp}\ \textsc{Cpd}$$

2. Inductively-defined relations: $m\ Div\ n$, evaluation relation of arithmetic expressions, etc.

$$\frac{}{m\ Div\ 0}\ \textsc{Zero} \qquad \frac{m\ Div\ n}{m\ Div\ n+m}\ \textsc{DSum}$$

Note that a relation is just a set of tuples. Hence $m\ Div\ n$ is essentially an inductively-defined set of pairs (m, n).

2.2 Rule Induction

Rule induction rules! For every *sound* inductive definition, we have a rule induction principle for free. The notion is simple, to prove some property P for every element of an inductively defined set, it is sufficient to prove P holds for the conclusion assuming P holds for all the premises, for every rule in the inductive definition. That is, for every rule of the form

$$\frac{premise_1 \qquad \ldots \qquad premise_n}{conclusion},$$

prove $P\,(conclusion)$ assuming $P\,(premise_1)$, ..., $P\,(premise_n)$, for every $n \in \mathbf{N}$. When $n = 0$, a rule becomes an axiom. In this case, you have to prove $P(conclusion)$ “out of thin air”.

1. Prove that the sum of the first n natural numbers is $\frac{n(n+1)}{2}$, that is, prove
$$\sum_{n \in \mathbf{N}} n = \frac{n\,(n+1)}{2}$$

2. Prove that $m\ Div\ n_1$ and $m\ Div\ n_2$ implies $m\ Div\ (n_1 + n_2)$.

Proof. We prove the property

$$P\,(m\ Div\ n_1) = [\forall n_2 \in \mathbf{N}, m\ Div\ n_2 \text{ implies } m\ Div\ (n_1 + n_2)]$$

for every $m\ Div\ n_1$ by rule induction on $m\ Div\ n_1$.

- Base case: for $\dfrac{}{m\ Div\ 0}$, we want to prove

  $$P\,(m\ Div\ 0) = [\forall n_2 \in \mathbf{N}, m\ Div\ n_2 \text{ implies } m\ Div\ (0 + n_2)]$$

  Since $0+n_2 = n_2$, the goal is to prove $m\ Div\ n_2$ implies $m\ Div\ n_2$, which is trivial.

- Inductive case: for $\dfrac{m\ Div\ n_1}{m\ Div\ n_1 + m}$, we want to prove

  $$\begin{aligned} &P\,(m\ Div\ \ (n_1 + m)) = \\ &\qquad [\forall n_2 \in \mathbf{N}, m\ Div\ n_2 \text{ implies } m\ Div\ (n_1 + m + n_2)]\,, \end{aligned}$$

  assuming the inductive hypothesis

  $$\begin{aligned} &P\,(m\ Div\ n_1) = \\ &\qquad [\forall n_2 \in \mathbf{N}, m\ Div\ n_2 \text{ implies } m\ Div\ (n_1 + m)]\,, \end{aligned}$$

  Suppose $\forall n_2 \in \mathbf{N}, m\ Div\ n_2$, by the inductive hypothesis, we have $m\ Div\ \ (n_1 + n_2)$, then apply the rule DSum by instantiating n with $(n_1 + n_2 + m)$, we can conclude $m\ Div\ \ (m\ Div\ n_1 + n_2 + m)$, that is,

  $$\frac{m\ Div\ \ (n_1 + n_2)}{m\ Div\ \ (n_1 + n_2 + m)}\ \textsc{DSum}.$$

  Further, from $m\ Div\ \ (n_1 + n_2 + m)$, by associativity and commutativity of $+$, we can derive $m\ Div\ \ (n_1 + m + n_2)$, which is exactly what we want to prove.

Therefore, we have proved $P\,(m\ Div\ n_1)$ for every $m\ Div\ n_1$, that is, $\forall m, n_1, n_2 \in \mathbf{N}, m\ Div\ n_1$ and $m\ Div\ n_2$ implies $m\ Div\ (n_1 + n_2)$. $\square$

3 Evaluation Semantics vs. Reduction Semantics

1. Give the (structural) evaluation semantics (aka. (structural) big-step (operational) semantics, natural semantics) for arithmetic expressions.

   We first define a syntactic category for values. For convenience, we integrate it into the semi-abstract syntax given above for arithmetic expressions. Hereby we have the following syntax definition:

$$\begin{aligned} e &\in Exp \\ n &\in Int \\ o &\in Opr \\ v &\in Val \end{aligned}$$

$$\begin{array}{rcclr} \text{(expression)} & e & ::= & v & \text{(value)} \\ & & \mid & e_1 \; o \; e_2 & \text{(compound)} \\ \\ \text{(operator)} & o & ::= & + & \text{(plus)} \\ & & \mid & - & \text{(munus)} \\ & & \mid & * & \text{(times)} \\ & & \mid & / & \text{(divides)} \\ \\ \text{(value)} & v & ::= & n & \text{(number)} \end{array}$$

   The evaluation semantics for arithmetic expressions is given by a evaluation relation between expressions and values (the final form of expressions after a series of reductions), notated as $e \Longrightarrow v$, inductively defined as follows:

$$\frac{}{v \Longrightarrow v}\ \text{EvV} \qquad \frac{e_1 \Longrightarrow v_1 \qquad e_2 \Longrightarrow v_2}{e_1 \; o \; e_2 \Longrightarrow op\,(o, v_1, v_2)}\ \text{EvC}$$

   The inductive definition contains two rules, one axiom named EvV, one named EvC with two premises. The axiom EvV essentially says that a value evaluates to itself. In this example, a value can only be a number. The rule EvC says, to obtain the value of a compound expression $e_1 \; o \; e_2$, evaluate e_1 to v_1 and e_2 to v_2, then use a primitive operation corresponding to the operator o to get the value of the whole expression from the two values v_1 and v_2. Note that for simple arithmetic expressions, the order of the two premises does not matter since the evaluation of the two sub-expressions are independent of each other.

These rules should remind you of the interpreter you have crafted for arithmetic expressions.

Here is a demonstration of how to apply these evaluation rules to obtain the value of an example expression: $1+2*3-4$. Note that we assume the meta-function *op* can handle the operators $+$, $*$ and $-$ correctly.

$$\dfrac{\dfrac{\dfrac{}{1 \Longrightarrow 1}\ \textsc{EvV} \qquad \dfrac{\dfrac{}{2 \Longrightarrow 2}\ \textsc{EvV} \qquad \dfrac{}{3 \Longrightarrow 3}\ \textsc{EvV}}{2*3 \Longrightarrow op\,(*,2,3)=6}\ \textsc{EvC}}{1+2*3 \Longrightarrow op\,(+,1,6)=7}\ \textsc{EvC} \qquad \dfrac{}{4 \Longrightarrow 4}\ \textsc{EvV}}{1+2*3-4 \Longrightarrow op\,(-,7,4)=3}\ \textsc{EvC}$$

So we know the value of the expression $1+2*3$ is 7.

2. Compare evaluation semantics and (structural) reduction semantics (aka. (structural) small-step (operational) semantics).

   The reduction semantics for arithmetic expressions is given by a reduction relation between expressions, notated as $e \longrightarrow e'$, inductively defined as follows:

$$\dfrac{}{v_1\ o\ v_2 \longrightarrow op\,(o, v_1, v_2)}\ \textsc{Red}$$

$$\dfrac{e_1 \longrightarrow e_1'}{e_1\ o\ e_2 \longrightarrow e_1'\ o\ e_2}\ \textsc{RdL} \qquad \dfrac{e_2 \longrightarrow e_2'}{e_1\ o\ e_2 \longrightarrow e_1\ o\ e_2'}\ \textsc{RdR}$$

   The axiom RED says that we can invoke the primitve operation corresponding to the operator o only when both its operands have reduced to values. The rule RDL covers one-step reduction of the left operand of a compound expression, while the rule RDR covers that of the right operand. Note that there is no longer a rule like $v \longrightarrow v$, since a value cannot be reduced in one step to anything. If such a rule is included in the definition, after an expression is reduced to a value, one can keep reducing it to itself by applying this rule till the end of the world (Who knows when it is, maybe Maya people?).

   Here is a demonstration of how to apply these reduction rules to obtain the final result the same expression $1+2*3-4$. Again, we assume the correctness of the meta-function *op*.

$$\dfrac{\dfrac{\dfrac{}{2*3 \longrightarrow op\,(*,2,3)=6}\ \textsc{Red}}{1+2*3 \longrightarrow 1+6}\ \textsc{RdR}}{1+2*3-4 \longrightarrow 1+6-4}\ \textsc{RdL}$$

This is just one-step reduction. To reach the final result of the expression, we need continue reducing the result expression $1 + 6 - 4$:

$$\dfrac{\dfrac{}{1 + 6 \longrightarrow op\,(+, 1, 6) = 7}\ \textsc{Red}}{1 + 6 - 4 \longrightarrow 7 - 4}\ \textsc{RdL}$$

Go on reducing $7 - 4$:

$$\dfrac{}{7 - 4 \longrightarrow 3}\ \textsc{Red}$$

Now that the number 3 can no longer be reduced, it is the result of the whole expression. So we have seen that the original expression $1 + 2 * 3 - 4$ is reduced by *three* steps to the number 3, that is,

$$1 + 2 * 3 - 4 \longrightarrow 1 + 6 - 4 \longrightarrow 7 - 4 \longrightarrow 3$$

Note that the number of reduction steps is clearly indicated by the number of occurrences of the rule Red in the three derivation trees.