CS720

Logical Foundations of Computer Science

Lecture 11: Formalizing an expression language

Tiago Cogumbreiro

Specifying a programming language

This week's objective (recall lecture 1)

Language grammar

$$t ::= x \mid v \mid t\ t \qquad v ::= \lambda x\colon T.t \qquad T ::= T \to T \mid \mathbf{unit}$$

Evaluation rules

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \texttt{(E-app1)} \qquad \frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}\ \texttt{(E-app2)}$$

$$(\lambda x\colon T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}\ \texttt{(E-abs)}$$

Summary

- Formalizing arithmetic expressions
- Abstract syntax
- Code transformations
- Functions as relations

`Imp.v`

Due Thursday October 18, 11:59pm EST

Arithmetic expressions

Abstract syntax: inductive types versus BNF

- BNF is an informal representation, glosses over some details on how to parse
- BNF is a cleaner way of communicating, better suited for presentation
- The grammar defines all possible terms that we are able to *write* (in this case expressions); terms can still be ill-formed (eg, have typing errors)
- Expression `APlus (ANum 1) (AMult (ANum 2) (ANum 3))` means $1 + 2 \times 3$

```
Inductive aexp : Type :=
  | ANum : nat → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp.
```

$$a ::= n \mid a + a \mid a - a \mid a \times a$$

How do we attribute meaning to a language?

How do we attribute meaning to a language?

We show how to run it.

(Operational Semantics)

Implementing an interpreter

An interpreter is a program that executes an abstract syntax.

```
Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | APlus a1 a2 ⇒ (aeval a1) + (aeval a2)
  | AMinus a1 a2 ⇒ (aeval a1) - (aeval a2)
  | AMult a1 a2 ⇒ (aeval a1) * (aeval a2)
  end.

Goal aeval (APlus (ANum 1) (AMult (ANum 2) (ANum 3))) = 7.
Proof.  reflexivity. Qed.
```

Code transformation steps

We can implement a compiler optimization stage as follows:

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | APlus (ANum 0) e2 ⇒ optimize_0plus e2
  | APlus e1 e2 ⇒ APlus (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 ⇒ AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult e1 e2 ⇒ AMult (optimize_0plus e1) (optimize_0plus e2)
  end.
(* 2 + (0 + (0 + 1)) = 2 + 1 *)
Goal optimize_0plus (APlus (ANum 2) (APlus (ANum 0) (APlus (ANum 0) (ANum 1))))
  = APlus (ANum 2) (ANum 1).
Proof. reflexivity. Qed.
```

Optimizer is correct

```
Theorem optimize_0plus_sound: forall a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a. induction a.
```

(Done in class.)

Evaluation as a relation

```
Reserved Notation "a '\\' n"
  (at level 50, left associativity).
Inductive aevalR : aexp → nat → Prop :=
| E_ANum : forall (n:nat),
    ANum n \\ n
| E_APlus : forall (a1 a2: aexp) (n1 n2 : nat),
    a1 \\ n1 → a2 \\ n2 → APlus a1 a2 \\ (n1 + n2)
| E_AMinus : forall (a1 a2: aexp) (n1 n2 : nat),
    a1 \\ n1 → a2 \\ n2 → AMinus a1 a2 \\ (n1 - n2)
| E_AMult :  forall (a1 a2: aexp) (n1 n2 : nat),
    a1 \\ n1 → a2 \\ n2 → AMult a1 a2 \\ (n1 * n2)

where "a '\\' n" := (aevalR a n) : type_scope.
```

$$\texttt{ANum}(n) \setminus\setminus n$$

$$\frac{a_1 \setminus\setminus n_1 \qquad a_2 \setminus\setminus n_2}{\texttt{APlus}(a_1, a_2) \setminus\setminus n_1 + n_2}$$

$$\frac{a_1 \setminus\setminus n_1 \qquad a_2 \setminus\setminus n_2}{\texttt{AMinus}(a_1, a_2) \setminus\setminus n_1 - n_2}$$

$$\frac{a_1 \setminus\setminus n_1 \qquad a_2 \setminus\setminus n_2}{\texttt{AMult}(a_1, a_2) \setminus\setminus n_1 \times n_2}$$

Show that `aeval` implements `aevalR`

```
Theorem aeval_iff_aevalR : forall a n,
  (a \\ n) <-> aeval a = n.
```

($\longrightarrow$) by induction on the derivation tree of the hypothesis.

($\longleftarrow$) by induction on the structure of `a`.

Adding variables

Our goal is to implement an imperative language

```
Inductive aexp : Type :=
  | ANum : nat → aexp
  | AId : string → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp.
```

How do we represent memory?

Total maps (or dictionaries)

To map strings (identifiers) into some type

Homework: read `Maps.v`, you will need to use it in this homework.

- `{ --> d }` represents an "empty" dictionary with a default value `d`; because this is a total map, all keys are set to `d`.
- `m & { k --> v }` extends a map `m` and assigns value `v` to key `k`

Example, let `m = { --> 3 } { "x" --> 2 }`, what is the result of:

1. `m "foo"`
2. `m "x"`
3. `m ""`

Evaluate an expression with variables

```
Definition state := total_map nat.

Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | AId ⇒ ???
  | APlus a1 a2 ⇒ (aeval a1) + (aeval a2)
  | AMinus a1 a2 ⇒ (aeval a1) - (aeval a2)
  | AMult a1 a2 ⇒ (aeval a1) * (aeval a2)
  end.
```

Evaluate an expression with variables

```
Definition state := total_map nat.

Fixpoint aeval (st : state) (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | AId x ⇒ st x
  | APlus a1 a2 ⇒ (aeval st a1) + (aeval st a2)
  | AMinus a1 a2  ⇒ (aeval st a1) - (aeval st a2)
  | AMult a1 a2 ⇒ (aeval st a1) * (aeval st a2)
  end.
```

Functions as relations (revisited)

And on generalizing code

Revisiting `optimize_0plus`

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  (* No optimization *)
  | ANum n => ANum n
  (* Optimize *)
  | APlus (ANum 0) e2 => optimize_0plus e2
  (* Recurse *)
  | APlus e1 e2 => APlus (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 => AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult e1 e2 => AMult (optimize_0plus e1) (optimize_0plus e2)
  end.
```

> How can we represent `optimize_0plus` as a relation?

optimize_0plus as a relation

```
Inductive Opt_0plus: aexp → aexp → Prop :=
(* No optimization *)
| opt_0plus_skip: forall n, Opt_0plus (ANum n) (ANum n)
(* Optmize *)
| opt_0plus_do: forall a, Opt_0plus (APlus (ANum 0) a) a
(* Recurse *)
| opt_0plus_plus:
  forall a1 a2 a1' a2',
  Opt_0plus a1 a1' →
  Opt_0plus a2 a2' →
  Opt_0plus (APlus a1 a2) (APlus a1 a2')
| opt_0plus_minus: forall a1 a2 a1' a2',
  Opt_0plus a1 a1' → Opt_0plus a2 a2' → Opt_0plus (AMinus a1 a2) (AMinus a1' a2')
| opt_0plus_mult: forall a1 a2 a1' a2',
  Opt_0plus a1 a1' → Opt_0plus a2 a2' → Opt_0plus (AMult a1 a2) (AMult a1' a2').
```

Tactics Cheat Sheet

Variables and conditions in a goal:

- `intros` moves ∀/condition to hypothesis
- `generalize dependent` moves variable to ∀

Solve:

- `reflexivity` goal `X=X` and `P<->P`
- `intuition` logical connectives
- `omega` arithmetic expressions
- `auto using Theorem1,Theorem2 with *`
- `condtradiction`, `contradict H`

Automate:

- `t1;t2` run `t2` after each goal created by `t1`
- `try t` run `t` and ignores failure

Proof by the principle of:

- `destruct` case analysis
- `induction`
- `inversion` injective/disjoint constructors

Theorems as expressions:

- `apply` applies a theorem/hypothesis
- `assert (H:e)` introduces assumption
- `assert (H:=e)` applies a theorem

Rearrange terms:

- `rewrite` equations and equivalences
- `simpl` evaluates an expression
- `unfold` opens a `Definition`

See also `Tactics.html`.

Summary

- Formalizing arithmetic expressions
- Abstract syntax
- Code transformations
- Functions as relations