Specifying a programming language

This week's objective (recall lecture 1)

Language grammar

\[
\begin{align*}
t &::= x \mid v \mid t\ t \\
v &::= \lambda x: T.t \\
T &::= T \to T \mid \text{unit}
\end{align*}
\]

Evaluation rules

\[
\begin{align*}
t_1 \to t_1' & \quad (E\text{-app1}) \\
t_2 \to t_2' & \quad (E\text{-app2})
\end{align*}
\]

\[
\begin{align*}
(\lambda x: T_{11}.t_{12})\ v_2 & \to [x \mapsto v_2]t_{12} \quad (E\text{-abs})
\end{align*}
\]
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 \( \text{APlus (ANum 1) (AMult (ANum 2) (ANum 3))} \) means \( 1 + 2 \times 3 \)

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

\[
a ::= n | a + a | a - a | a \times a
\]
How do we attribute meaning to a language?
How do we attribute meaning to a language?

We show how to run it.

(Operation 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: \(\forall a, \text{aeval (optimize_0plus a)} = \text{aeval a} \).

**Proof.**

\begin{verbatim}
  intros a. induction a.
\end{verbatim}

*(Done in class.)*
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.
Show that `aeval` implements `aevalR`

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

(→) by induction on the derivation tree of the hypothesis.

(←) by induction on the structure of `a`. 
Adding variables

Our goal is to implement an imperative language

Inductive aexp : Type :=
| ANum : nat → aexp
| AIdd : 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.

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

Example, let \texttt{m = \{ 3 \} \{ "x" \rightarrow 2 \}}, what is the result of:

1. \texttt{m "foo"}
2. \texttt{m "x"}
3. \texttt{m ""}
Evaluate an expression with variables

**Definition** state := total_map nat.

**Fixpoint** aeval (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | AId ⇒ ???
  | AP1us 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`

```ml
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)
(* Optimize *)
| 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_mul:
  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 $\forall$/condition to hypothesis
- **generalize dependent** moves variable to $\forall$

Solve:
- **reflexivity** goal $X=X$ and $P \leftrightarrow P$
- **intuition** logical connectives
- **omega** arithmetic expressions
- **auto using Theorem1,Theorem2 with *  
- **condtradsition,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