barkhauseninstitut.org

barkhausen institut

Foundations of Certified Programming Language and Compiler Design

Dr.-Ing. Sebastian Ertel

Composable Operating Systems Group, Barkhausen Institute

Outline

Lecture	Logic	Formalisms	PL
1	Propositional and first-order logic		
2			Functional programming
3		Syntax and Semantics	
4			The untyped lambda calculus
5		Types	
6			The typed lambda calculus
7			Polymorphism
8		Curry-Howard	
9			Higher-order types
10			Dependent types

Inference Rules Recap

So far, we used inference rules to specify:

Inference Rules Recap

So far, we used inference rules to specify:

- the syntax of terms

$$\frac{}{0 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{succ}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

Inference Rules Recap

So far, we used inference rules to specify:

- the syntax of terms

$$\frac{}{0 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{succ}\; t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

- the “preservation” of a predicate

$$\frac{}{P(\texttt{Zero})} \qquad \frac{t_1 \in \texttt{Term} \quad P(t_1)}{P(\texttt{Succ}\; t_1)} \qquad \frac{t_1 \in \texttt{Term} \quad P(t_1) \quad t_2 \in \texttt{Term} \quad P(t_2)}{P(\texttt{Add}\; t_1\; t_2)}$$

Inference Rules Recap

- That is, we used inference rules to specify relations:

Inference Rules Recap

- That is, we used inference rules to specify relations:
- the relation $\in$ of terms(/words) t and the set of all terms $\mathcal{T}$

$$\boxed{t \in \mathcal{T}}$$

$$\frac{}{0 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{succ}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

Inference Rules Recap

- That is, we used inference rules to specify relations:
- the relation $\in$ of terms(/words) t and the set of all terms $\mathcal{T}$

$$\boxed{t \in \mathcal{T}}$$

$$\frac{}{0 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{succ}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

- the property P of terms t

$$\boxed{P(t)}$$

$$\frac{}{P(\texttt{Zero})} \qquad \frac{t_1 \in \texttt{term} \quad P(t_1)}{P(\texttt{Succ}\ t_1)} \qquad \frac{t_1 \in \texttt{term} \quad P(t_1) \quad t_2 \in \texttt{term} \quad P(t_2)}{P(\texttt{Add}\ t_1\ t_2)}$$

Inference Rules Recap

- That is, we used inference rules to specify relations:
- the relation $\in$ of terms(/words) t and the set of all terms $\mathcal{T}$

$$\boxed{t \in \mathcal{T}}$$

$$\frac{}{0 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{succ}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

- the property P of terms t

$$\boxed{P(t)}$$

$$\frac{}{P(\texttt{Zero})} \qquad \frac{t_1 \in \texttt{term} \quad P(t_1)}{P(\texttt{Succ}\ t_1)} \qquad \frac{t_1 \in \texttt{term} \quad P(t_1) \quad t_2 \in \texttt{term} \quad P(t_2)}{P(\texttt{Add}\ t_1\ t_2)}$$

- A function is a type of relation that associates one input with exactly one output.

Inference Rules Recap

- That is, we used inference rules to specify relations:
- the relation $\in$ of terms(/words) t and the set of all terms $\mathcal{T}$

$$\boxed{t \in \mathcal{T}}$$

$$\frac{}{0 \in \mathcal{T}} \qquad \frac{t_1 \ \in \ \mathcal{T}}{\texttt{succ}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

- the property P of terms t

$$\boxed{P(t)}$$

$$\frac{}{P(\texttt{Zero})} \qquad \frac{t_1 \in \texttt{term} \quad P(t_1)}{P(\texttt{Succ}\ t_1)} \qquad \frac{t_1 \in \texttt{term} \quad P(t_1) \quad t_2 \in \texttt{term} \quad P(t_2)}{P(\texttt{Add}\ t_1\ t_2)}$$

- A function is a type of relation that associates one input with exactly one output.
- A binary relation is a set of (ordered) pairs where one input maybe related to more than one output.

Inference Rules Recap

- That is, we used inference rules to specify relations:
- the relation $\in$ of terms(/words) t and the set of all terms $\mathcal{T}$

$$\boxed{t \in \mathcal{T}}$$

$$\frac{}{0 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{succ}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

- the property P of terms t

$$\boxed{P(t)}$$

$$\frac{}{P(\texttt{Zero})} \qquad \frac{t_1 \in \texttt{term} \quad P(t_1)}{P(\texttt{Succ}\ t_1)} \qquad \frac{t_1 \in \texttt{term} \quad P(t_1) \quad t_2 \in \texttt{term} \quad P(t_2)}{P(\texttt{Add}\ t_1\ t_2)}$$

- A function is a type of relation that associates one input with exactly one output.
- A binary relation is a set of (ordered) pairs where one input maybe related to more than one output.
- Here is one important relation for the definition of programming languages:

Evaluation

Mathematically: For programming languages such a relation is the evaluation of a term:

Definition (One-Step Evaluation Relation)

The *one-step* evaluation relation $\longrightarrow$ is the smallest binary relation that relates a term t of a language to another term t'.

Evaluation

Mathematically: For programming languages such a relation is the evaluation of a term:

Definition (One-Step Evaluation Relation)

The *one-step* evaluation relation $\longrightarrow$ is the smallest binary relation that relates a term t of a language to another term t'.

Logically: This relation defines a term rewriting system.

Semantically: We talk about evaluating a term, i.e., we reduce it to another term of a smaller size.

Example: Booleans

Syntax

t	$::=$		terms:
	$\mid$	`true`	constant true
	$\mid$	`false`	constant false
	$\mid$	`if` t `then` t `else` t	conditional

v	$::=$		values:
	$\mid$	`true`	true value
	$\mid$	`false`	false value

Example: Booleans

Syntax

$$
\begin{array}{lllll}
t & ::= & & & \text{terms:} \\
 & | & \texttt{true} & & \text{constant true} \\
 & | & \texttt{false} & & \text{constant false} \\
 & | & \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t & & \text{conditional}
\end{array}
$$

$$
\begin{array}{llll}
v & ::= & & \text{values:} \\
 & | & \texttt{true} & \text{true value} \\
 & | & \texttt{false} & \text{false value}
\end{array}
$$

Evaluation

$$\frac{}{\texttt{if true then}\ t_2\ \texttt{else}\ t_3 \longrightarrow t_2}\ \text{E-IfTrue} \qquad \frac{}{\texttt{if false then}\ t_2\ \texttt{else}\ t_3 \longrightarrow t_3}\ \text{E-IfFalse}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \longrightarrow \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3}\ \text{E-If}$$

Computations with Booleans

Derivation Trees

$$s \stackrel{\text{def}}{=} \texttt{if true then false else false}$$

Computations with Booleans

Derivation Trees

$$s \overset{\text{def}}{=} \texttt{if true then false else false}$$

$$t \overset{\text{def}}{=} \texttt{if } s \texttt{ then true else true}$$

Computations with Booleans

Derivation Trees

$$s \stackrel{\text{def}}{=} \texttt{if true then false else false}$$

$$t \stackrel{\text{def}}{=} \texttt{if}\ s\ \texttt{then true else true}$$

$$u \stackrel{\text{def}}{=} \texttt{if false then true else true}$$

$$s \stackrel{\text{def}}{=} \texttt{if true then false else false}$$
$$t \stackrel{\text{def}}{=} \texttt{if } s \texttt{ then true else true}$$
$$u \stackrel{\text{def}}{=} \texttt{if false then true else true}$$

$$\frac{}{\texttt{if } t \texttt{ then false else false} \longrightarrow \texttt{if } u \texttt{ then false else false}}\ \text{E-IF}$$

Computations with Booleans

Derivation Trees

$$s \stackrel{\text{def}}{=} \texttt{if true then false else false}$$
$$t \stackrel{\text{def}}{=} \texttt{if } s \texttt{ then true else true}$$
$$u \stackrel{\text{def}}{=} \texttt{if false then true else true}$$

$$\dfrac{\dfrac{}{\texttt{if } s \texttt{ then true else true} \longrightarrow \texttt{if false then true else true}}\ \text{E-IF}}{\texttt{if } t \texttt{ then false else false} \longrightarrow \texttt{if } u \texttt{ then false else false}}\ \text{E-IF}$$

Computations with Booleans

Derivation Trees

$$s \stackrel{\text{def}}{=} \texttt{if true then false else false}$$
$$t \stackrel{\text{def}}{=} \texttt{if } s \texttt{ then true else true}$$
$$u \stackrel{\text{def}}{=} \texttt{if false then true else true}$$

$$\dfrac{\dfrac{\dfrac{}{\texttt{if true then false else false} \longrightarrow \texttt{false}}\ \text{E-IfTrue}}{\texttt{if } s \texttt{ then true else true} \longrightarrow \texttt{if false then true else true}}\ \text{E-If}}{\texttt{if } t \texttt{ then false else false} \longrightarrow \texttt{if } u \texttt{ then false else false}}\ \text{E-If}$$

Determinism

Theorem (Determinacy of One-Step Evaluation)

If $t \longrightarrow t'$ *and* $t \longrightarrow t''$ *then* $t' = t''$.

Determinism

Theorem (Determinacy of One-Step Evaluation)

If $t \longrightarrow t'$ and $t \longrightarrow t''$ then $t' = t''$.

Proof.

The proof is by induction **on the derivation** $t \longrightarrow t'$, i.e., on the structure of t.

Determinism

Theorem (Determinacy of One-Step Evaluation)

If $t \longrightarrow t'$ *and* $t \longrightarrow t''$ *then* $t' = t''$.

Proof.

The proof is by induction **on the derivation** $t \longrightarrow t'$, i.e., on the structure of t.

Case $\quad t \longrightarrow t' \quad t \longrightarrow t''$

Determinism

Theorem (Determinacy of One-Step Evaluation)

If $t \longrightarrow t'$ *and* $t \longrightarrow t''$ *then* $t' = t''$.

Proof.

The proof is by induction **on the derivation** $t \longrightarrow t'$, i.e., on the structure of t.

Case	$t \longrightarrow t'$	$t \longrightarrow t''$	
E-IfTrue	$$\dfrac{\vdots}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow t_2}\ \text{E-IfTrue}$$ where: $t_1 = \texttt{true}$	E-IfFalse: E-If:	$t_1 = \texttt{false} \neq t_1 = \texttt{true}$ $t_1 \longrightarrow t_1'$ but $t_1 = \texttt{true}$ such that $\texttt{true} \longrightarrow$???

Determinism

Theorem (Determinacy of One-Step Evaluation)

If $t \longrightarrow t'$ *and* $t \longrightarrow t''$ *then* $t' = t''$.

Proof.

The proof is by induction **on the derivation** $t \longrightarrow t'$, i.e., on the structure of t.

Case	$t \longrightarrow t'$	$t \longrightarrow t''$	
E-IfTrue	$$\dfrac{\vdots}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow t_2}\ \text{E-IfTrue}$$ where: $t_1 = \texttt{true}$	E-IfFalse: E-If:	$t_1 = \texttt{false} \neq t_1 = \texttt{true}$ $t_1 \longrightarrow t_1'$ but $t_1 = \texttt{true}$ such that $\texttt{true} \longrightarrow$???
E-IfFalse	analogous with $t_1 = \texttt{false}$		

Theorem (Determinacy of One-Step Evaluation)

If $t \longrightarrow t'$ and $t \longrightarrow t''$ then $t' = t''$.

Proof.

The proof is by induction **on the derivation** $t \longrightarrow t'$, i.e., on the structure of t.

Case	$t \longrightarrow t'$	$t \longrightarrow t''$	
E-IfTrue	$$\dfrac{\vdots}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow t_2}\ \text{E-IfTrue}$$ where: $t_1 = \texttt{true}$	E-IfFalse:	$t_1 = \texttt{false} \neq t_1 = \texttt{true}$
		E-If:	$t_1 \longrightarrow t_1'$ but $t_1 = \texttt{true}$ such that $\texttt{true} \longrightarrow$???
E-IfFalse	analogous with $t_1 = \texttt{false}$		
E-If		E-If :	By induction hypothesis $t_1' = t_1''$ for $t_1 \longrightarrow t_1'$ and $t_1 \longrightarrow t_1''$

The Result of an Evaluation

Definition (Normal Form)

A term t is in *normal form* if no evaluation rule applies to it, i.e., there exists no t' such that $t \longrightarrow t'$.

The Result of an Evaluation

Definition (Normal Form)

A term t is in *normal form* if no evaluation rule applies to it, i.e., there exists no t' such that $t \longrightarrow t'$.

- Interesting properties for our Booleans language would be:

Theorem

Every value is in normal form.

The Result of an Evaluation

Definition (Normal Form)

A term t is in *normal form* if no evaluation rule applies to it, i.e., there exists no t' such that $t \longrightarrow t'$.

- Interesting properties for our Booleans language would be:

Theorem

Every value is in normal form.

Theorem

If t is in normal form, then t is a value.

Multi-step Relation

Definition (Multi-step Evaluation Relation)

The multi-step evaluation relation $\longrightarrow^*$ is the reflexive, transitive closure of the one-step evaluation $(\longrightarrow)$. That is, it is the smallest relation such that

Multi-step Relation

Definition (Multi-step Evaluation Relation)

The multi-step evaluation relation $\longrightarrow^*$ is the reflexive, transitive closure of the one-step evaluation ($\longrightarrow$). That is, it is the smallest relation such that

$$\frac{t \longrightarrow t'}{t \longrightarrow^* t'}$$

Multi-step Relation

Definition (Multi-step Evaluation Relation)

The multi-step evaluation relation $\longrightarrow^*$ is the reflexive, transitive closure of the one-step evaluation $(\longrightarrow)$. That is, it is the smallest relation such that

$$\frac{t \longrightarrow t'}{t \longrightarrow^* t'} \qquad \frac{}{t \longrightarrow^* t}$$

Multi-step Relation

Definition (Multi-step Evaluation Relation)

The multi-step evaluation relation $\longrightarrow^*$ is the reflexive, transitive closure of the one-step evaluation $(\longrightarrow)$. That is, it is the smallest relation such that

$$\frac{t \longrightarrow t'}{t \longrightarrow^* t'} \qquad \frac{}{t \longrightarrow^* t} \qquad \frac{t \longrightarrow^* t' \quad t' \longrightarrow^* t''}{t \longrightarrow^* t''}$$

Multi-step Relation

Definition (Multi-step Evaluation Relation)

The multi-step evaluation relation $\longrightarrow^*$ is the reflexive, transitive closure of the one-step evaluation $(\longrightarrow)$. That is, it is the smallest relation such that

$$\frac{t \longrightarrow t'}{t \longrightarrow^* t'} \qquad \frac{}{t \longrightarrow^* t} \qquad \frac{t \longrightarrow^* t' \quad t' \longrightarrow^* t''}{t \longrightarrow^* t''}$$

Theorem (Uniqueness of Normal Forms)

If $t \longrightarrow^* u$ and $t \longrightarrow^* u'$, where u and u' are normal forms, then $u = u'$

Multi-step Relation

Definition (Multi-step Evaluation Relation)

The multi-step evaluation relation $\longrightarrow^*$ is the reflexive, transitive closure of the one-step evaluation $(\longrightarrow)$. That is, it is the smallest relation such that

$$\frac{t \longrightarrow t'}{t \longrightarrow^* t'} \qquad \frac{}{t \longrightarrow^* t} \qquad \frac{t \longrightarrow^* t' \quad t' \longrightarrow^* t''}{t \longrightarrow^* t''}$$

Theorem (Uniqueness of Normal Forms)

If $t \longrightarrow^* u$ and $t \longrightarrow^* u'$, where u and u' are normal forms, then $u = u'$

Proof.

Corollary of the determiniacy of one-step evaluation. □

Termination

Theorem (Termination of Evaluation)

For every term t there is some normal form t' such that $t \longrightarrow^* t'$.

Termination

Theorem (Termination of Evaluation)

For every term t there is some normal form t' such that $t \longrightarrow^* t'$.

Proof.

We define a *termination measure* f for the states in our system, i.e., terms in our language. We observe that for every reduction step the size (is a natural number that) decreases and the usual order on the natural numbers is well founded. That is, if there exists a reduction $t \longrightarrow t'$ then $f(t') < f(t)$. $\square$

Theorem (Termination of Evaluation)

For every term t there is some normal form t' such that $t \longrightarrow^* t'$.

Proof.

We define a *termination measure* f for the states in our system, i.e., terms in our language. We observe that for every reduction step the size (is a natural number that) decreases and the usual order on the natural numbers is well founded. That is, if there exists a reduction $t \longrightarrow t'$ then $f(t') < f(t)$. $\square$

```
Equations size (t:term) : nat :=
size TTrue := 0;
size TFalse := 0;
size (If t1 t2 t3) := 1 + (size t1) + (size t2) + (size t3).
```

Stuckness

Definition (Stuckness)

A closed term is *stuck* if it is in normal form but not a value.

Definition (Stuckness)

A closed term is *stuck* if it is in normal form but not a value.

Consider the extension of our Booleans language to natural numbers:

Additional Syntax

$$
\begin{array}{lcll}
t & ::= & \ldots & \text{terms:} \\
 & | & 0 & \text{constant zero} \\
 & | & \texttt{succ}\ t & \text{successor}
\end{array}
$$

Additional Evaluation Rules

Stuckness

Definition (Stuckness)

A closed term is *stuck* if it is in normal form but not a value.

Consider the extension of our Booleans language to natural numbers:

Additional Syntax

$$
\begin{array}{llll}
t & ::= & \ldots & \text{terms:} \\
  & \mid & 0 & \text{constant zero} \\
  & \mid & \mathtt{succ}\ t & \text{successor}
\end{array}
\qquad
\begin{array}{llll}
v & ::= & \ldots & \text{values:} \\
  & \mid & nv & \text{numeric value}
\end{array}
$$

Additional Evaluation Rules

Stuckness

Definition (Stuckness)

A closed term is *stuck* if it is in normal form but not a value.

Consider the extension of our Booleans language to natural numbers:

Additional Syntax

$$
\begin{array}{llll}
t & ::= & \ldots & \text{terms:} \\
 & \mid & 0 & \text{constant zero} \\
 & \mid & \mathtt{succ}\ t & \text{successor}
\end{array}
\qquad
\begin{array}{llll}
v & ::= & \ldots & \text{values:} \\
 & \mid & nv & \text{numeric value}
\end{array}
\qquad
\begin{array}{llll}
nv & ::= & & \text{numeric values} \\
 & \mid & 0 & \text{zero value} \\
 & \mid & \mathtt{succ}\ nv & \text{successor value}
\end{array}
$$

Additional Evaluation Rules

Stuckness

Definition (Stuckness)

A closed term is *stuck* if it is in normal form but not a value.

Consider the extension of our Booleans language to natural numbers:

Additional Syntax

$$\begin{array}{rcll} t & ::= & \dots & \text{terms:} \\ & | & 0 & \text{constant zero} \\ & | & \mathtt{succ}\ t & \text{successor} \end{array} \qquad \begin{array}{rcll} v & ::= & \dots & \text{values:} \\ & | & nv & \text{numeric value} \end{array} \qquad \begin{array}{rcll} nv & ::= & & \text{numeric values} \\ & | & 0 & \text{zero value} \\ & | & \mathtt{succ}\ nv & \text{successor value} \end{array}$$

Additional Evaluation Rules

$$\frac{t \longrightarrow t'}{\mathtt{succ}\ t \longrightarrow \mathtt{succ}\ t'} \ \textsc{E-Succ}$$

Stuckness

Definition (Stuckness)

A closed term is *stuck* if it is in normal form but not a value.

Consider the extension of our Booleans language to natural numbers:

Additional Syntax

t	$::=$	$\dots$	terms:
	$\vert$	0	constant zero
	$\vert$	$\texttt{succ}\ t$	successor

v	$::=$	$\dots$	values:
	$\vert$	nv	numeric value

nv	$::=$		numeric values
	$\vert$	0	zero value
	$\vert$	$\texttt{succ}\ nv$	successor value

Additional Evaluation Rules

$$\frac{t \longrightarrow t'}{\texttt{succ}\ t \longrightarrow \texttt{succ}\ t'}\ \textsc{E-Succ}$$

Can you spot the stuck terms?

Defining the Semantics of Programming Languages

Operational semantics Two flavors exist:

Small-step What we have done so far when defining $\longrightarrow$.

Defining the Semantics of Programming Languages

Operational semantics Two flavors exist:

- **Small-step** What we have done so far when defining $\longrightarrow$.
- **Big-step** Essentially, what we have defined so far as $\longrightarrow^*$ but directly defined.

$$\boxed{t \Downarrow v}$$

$$\frac{}{v \Downarrow v}\ \textsc{B-Value}$$

$$\frac{t_1 \Downarrow \texttt{true} \qquad t_2 \Downarrow v_2}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow v_2}\ \textsc{B-IfTrue} \qquad \frac{t_1 \Downarrow \texttt{false} \qquad t_3 \Downarrow v_3}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow v_3}\ \textsc{B-IfFalse}$$

$$\frac{t \Downarrow nv}{\texttt{succ}\ t \Downarrow \texttt{succ}\ nv}\ \textsc{B-Succ}$$

Defining the Semantics of Programming Languages

Operational semantics Two flavors exist:

- **Small-step** What we have done so far when defining $\longrightarrow$.
- **Big-step** Essentially, what we have defined so far as $\longrightarrow^*$ but directly defined.

Denotational Semantics Maps the terms of the language to elements of another domain.

Additional Syntax

$$\begin{array}{rcll} t & ::= & \ldots & \text{terms:} \\ & | & \texttt{add}\ t\ t & \text{addition} \end{array}$$

Additional Evaluation

$$\frac{t_1 \longrightarrow t_1'}{\texttt{add}\ t_1\ t_2 \longrightarrow \texttt{add}\ t_1'\ t_2} \text{ E-AddLeft} \qquad \frac{t_2 \longrightarrow t_2'}{\texttt{add}\ nv_1\ t_2 \longrightarrow \texttt{add}\ nv_1\ t_2'} \text{ E-AddRight}$$

$$\frac{nv_1 + nv_2 = nv_3}{\texttt{add}\ nv_1\ nv_2 \longrightarrow nv_3} \text{ E-AddRight}$$

Mapping `add` onto the addition of natural numbers.

Defining the Semantics of Programming Languages

Operational semantics Two flavors exist:

Small-step What we have done so far when defining $\longrightarrow$.

Big-step Essentially, what we have defined so far as $\longrightarrow^*$ but directly defined.

Denotational Semantics Maps the terms of the language to elements of another domain.

We will come back to this in the final part of this lecture when we want to prove properties such as semantic preservation of compiler transformations etc.

Defining the Semantics of Programming Languages

Operational semantics Two flavors exist:

- Small-step What we have done so far when defining $\longrightarrow$.
- Big-step Essentially, what we have defined so far as $\longrightarrow^*$ but directly defined.

Denotational Semantics Maps the terms of the language to elements of another domain.

Axiomatic Semantics Instead of proving properties additionally, the properties are defined right in the system. The most prominent example is *Hoare Logic*.

Defining Relations

Haskell

Gallina (Coq)

Defining Relations

Haskell

recursive

```
data Nat = O | S Nat
```

Gallina (Coq)

```
Inductive nat : Type := O | S (n : nat).
```

Defining Relations

Haskell

recursive

```
data Nat = O | S Nat
```

(GADT)

```
data Nat where
    O :: Nat
    S :: Nat -> Nat
```

Gallina (Coq)

```
Inductive nat : Type := O | S (n : nat).
```

```
Inductive nat : Type :=
| O : nat
| S : nat -> nat.
```

Defining Relations

Haskell

recursive

```haskell
data Nat = O | S Nat
```

(GADT)

```haskell
data Nat where
    O :: Nat
    S :: Nat -> Nat
```

parameterized

```haskell
data List a :: Type where
    Nil :: List a
    Cons :: a -> List a -> List a
```

Gallina (Coq)

```coq
Inductive nat : Type := O | S (n : nat).
```

```coq
Inductive nat : Type :=
| O : nat
| S : nat -> nat.
```

```coq
Inductive list (A:Type) : Set :=
| nil : list A
| cons : A -> list A -> list A.
```

Defining Relations

Haskell

recursive

```
data Nat = O | S Nat
```

(GADT)

```
data Nat where
    O :: Nat
    S :: Nat -> Nat
```

parameterized

```
data List a :: Type where
    Nil :: List a
    Cons :: a -> List a -> List a
```

annotated/indexed

```
data Ev :: Nat -> Type where
    EvO :: Ev 'O
    EvSS :: Ev n -> Ev ('S ('S n))
```

Gallina (Coq)

```
Inductive nat : Type := O | S (n : nat).

Inductive nat : Type :=
| O : nat
| S : nat -> nat.

Inductive list (A:Type) : Set :=
| nil : list A
| cons : A -> list A -> list A.

Inductive ev : nat -> Prop :=
| ev_0 : ev O
| ev_SS : (n : nat), ev n -> ev (S (S n)).
```

What we have learned

- The definition of programming languages is built upon relations.
- We know the evaluation relation and
- know how to implement it?! (See the next exercise.)