263-2200

Types and Programming Languages
Outline

Types
   Evaluation Rules
   Typing Rules

Properties of the Typing Relation
   The Inversion Lemma
   Prolog Implementation

Reasoning Involving Types
   Progress
   Preservation
Outline

Types
  Evaluation Rules
  Typing Rules

Properties of the Typing Relation
  The Inversion Lemma
  Prolog Implementation

Reasoning Involving Types
  Progress
  Preservation
Plan

- We will now revisit the simple language of arithmetic and boolean expressions *NB* and show how to equip it with a (very simple) type system
- The key property of this type system will be *soundness*: *Well-typed programs do not get stuck*
- After that we will develop a simple type system for the \( \lambda \)-calculus
- We will spend a good part of the rest of the semester adding features to this type system
Outline

1. begin with a set of terms, a set of values, and an evaluation relation
2. define a set of *types* classifying values according to their “shapes”
3. define a typing relation $t : T$ that classifies terms according to the shape of the values that result from evaluating them
4. check that the typing relation is *sound* in the sense that,
   4.1 if $t : T$ and $t \longrightarrow^* v$, then $v : T$
   4.2 if $t : T$ then the evaluation of $t$ will not get stuck
The Language \( NB \)

\[ t ::= \]
\begin{align*}
true & \quad \text{constant true} \\
false & \quad \text{constant false} \\
if \ t \ \text{then} \ t \ \text{else} \ t & \quad \text{conditional} \\
0 & \quad \text{constant zero} \\
succ \ t & \quad \text{successor} \\
prec \ t & \quad \text{predecessor} \\
iszero \ t & \quad \text{zero test}
\end{align*}

\[ v ::= \]
\begin{align*}
true & \quad \text{true value} \\
false & \quad \text{false value} \\
nv & \quad \text{numeric value}
\end{align*}

\[ nv ::= \]
\begin{align*}
0 & \quad \text{zero value} \\
succ \ nv & \quad \text{successor value}
\end{align*}
Evaluation Rules

\[
\text{if } \text{true} \text{ then } t_2 \text{ else } t_3 \rightarrow t_2 \quad \text{(E-IfTrue)}
\]

\[
\text{if } \text{false} \text{ then } t_2 \text{ else } t_3 \rightarrow t_3 \quad \text{(E-IfFalse)}
\]

\[
t_1 \rightarrow t'_1
\]

\[
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 \quad \text{(E-If)}
\]
\[
\begin{align*}
&\frac{t_1 \rightarrow t_1'}{(E\text{-Succ})} \\
&\frac{\text{succ } t_1 \rightarrow \text{succ } t_1'}{(E\text{-Pred})} \\
&\frac{\text{pred } 0 \rightarrow 0}{(E\text{-PredZero})} \\
&\frac{\text{pred } (\text{succ } n v_1) \rightarrow n v_1}{(E\text{-PredSucc})} \\
&\frac{t_1 \rightarrow t_1'}{(E\text{-Pred})} \\
&\frac{\text{pred } t_1 \rightarrow \text{pred } t_1'}{(E\text{-Pred})} \\
&\frac{\text{iszero } 0 \rightarrow \text{true}}{(E\text{-IsZeroZero})} \\
&\frac{\text{iszero } (\text{succ } n v_1) \rightarrow \text{false}}{(E\text{-IsZeroSucc})} \\
&\frac{t_1 \rightarrow t_1'}{(E\text{-IsZero})} \\
&\frac{\text{iszero } t_1 \rightarrow \text{iszero } t_1'}{(E\text{-IsZero})}
\end{align*}
\]
Types

In this language, values have two possible “shapes”: they are either booleans or numbers.

\[ T ::= \]

- \( \text{Bool} \) type of booleans
- \( \text{Nat} \) type of numbers
Typing Rules

true : Bool  \hspace{2cm} (T-True)

false : Bool  \hspace{2cm} (T-False)

\[
t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T
\]
\[
\frac{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}{(T-\text{If})}
\]

0 : Nat  \hspace{2cm} (T-Zero)

\[
t_1 : \text{Nat}
\]
\[
\frac{\text{succ } t_1 : \text{Nat}}{(T-\text{Succ})}
\]

\[
t_1 : \text{Nat}
\]
\[
\frac{\text{pred } t_1 : \text{Nat}}{(T-\text{Pred})}
\]

\[
t_1 : \text{Nat}
\]
\[
\frac{\text{iszero } t_1 : \text{Bool}}{(T-\text{IsZero})}
\]
Typing Derivations

Every pair \((t, T)\) in the typing relation can be justified by a \textit{derivation tree} built from instances of the inference rules.

$$
\begin{align*}
\frac{0 : \text{Nat}}{
\text{T-Zero}
}
\end{align*}
\quad
\begin{align*}
\frac{\text{iszero } 0 : \text{Bool}}{
\text{T-IsZero}
}
\end{align*}
\quad
\begin{align*}
\frac{0 : \text{Nat}}{
\text{T-Zero}
}
\end{align*}
\quad
\begin{align*}
\frac{\text{pred } 0 : \text{Nat}}{
\text{T-Pred}
}
\end{align*}
\quad
\begin{align*}
\frac{\text{if iszero } 0 \text{ then } 0 \text{ else pred } 0 : \text{Nat}}{
\text{T-If}
}
\end{align*}
$$

Proofs of properties about the typing relation often proceed by induction on typing derivations.
Imprecision of Typing

Like other static program analyses, type systems are generally imprecise: they do not predict exactly what kind of value will be returned by every program, but just a conservative (safe) approximation.

\[
\begin{align*}
t_1 &: \text{Bool} \quad t_2 &: T \quad t_3 &: T \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 &: T
\end{align*}
\] (T-If)

Using this rule, we cannot assign a type to

\[
\text{if true then 0 else false}
\]

even though this term will certainly evaluate to a number.
Outline

Types
  Evaluation Rules
  Typing Rules

Properties of the Typing Relation
  The Inversion Lemma
  Prolog Implementation

Reasoning Involving Types
  Progress
  Preservation
The safety (or soundness) of this type system can be expressed by two properties:

1. **Progress**: A well-typed term is not stuck
   
   If \( t : T \), then either \( t \) is a value or else \( t \rightarrow t' \) for some \( t' \).

2. **Preservation**: Types are preserved by one-step evaluation
   
   If \( t : T \) and \( t \rightarrow t' \), then \( t' : T \).
Inversion

Lemma:

1. If $true : R$, then $R = \text{Bool}$.
2. If $false : R$, then $R = \text{Bool}$.
3. If $if \; t_1 \; \text{then} \; t_2 \; \text{else} \; t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$.
4. If $0 : R$, then $R = \text{Nat}$
5. if $\text{succ} \; t_1 : R$, then $R = \text{Nat}$ and $t_1 : \text{Nat}$.
6. if $\text{pred} \; t_1 : R$, then $R = \text{Nat}$ and $t_1 : \text{Nat}$.
7. if $\text{iszero} \; t_1 : R$, then $R = \text{Bool}$ and $t_1 : \text{Nat}$.
Inversion

Lemma:

1. If true : \( R \), then \( R = \text{Bool} \).
2. If false : \( R \), then \( R = \text{Bool} \).
3. If if \( t_1 \) then \( t_2 \) else \( t_3 \) : \( R \), then \( t_1 : \text{Bool} \), \( t_2 : R \), and \( t_3 : R \).
4. If 0 : \( R \), then \( R = \text{Nat} \)
5. if succ \( t_1 \) : \( R \), then \( R = \text{Nat} \) and \( t_1 : \text{Nat} \).
6. if pred \( t_1 \) : \( R \), then \( R = \text{Nat} \) and \( t_1 : \text{Nat} \).
7. if iszero \( t_1 \) : \( R \), then \( R = \text{Bool} \) and \( t_1 : \text{Nat} \).

Proof: ...
Lemma:

1. If \textit{true} : R, then \( R = \text{Bool} \).
2. If \textit{false} : R, then \( R = \text{Bool} \).
3. If \textit{if} \( t_1 \text{ then } t_2 \text{ else } t_3 \) : R, then \( t_1 : \text{Bool}, t_2 : R, \) and \( t_3 : R \).
4. If 0 : R, then \( R = \text{Nat} \)
5. if \textit{succ} \( t_1 \) : R, then \( R = \text{Nat} \) and \( t_1 : \text{Nat} \).
6. if \textit{pred} \( t_1 \) : R, then \( R = \text{Nat} \) and \( t_1 : \text{Nat} \).
7. if \textit{iszero} \( t_1 \) : R, then \( R = \text{Bool} \) and \( t_1 : \text{Nat} \).

Proof: \hspace{1cm} ...

This leads directly to a recursive algorithm for calculating the type of a term...
Step 1: Design a Mapping to a Term Algebra

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>map (\rightarrow) true</td>
</tr>
<tr>
<td>false</td>
<td>map (\rightarrow) false</td>
</tr>
<tr>
<td>if (t_1) then (t_2) else (t_3)</td>
<td>map (\rightarrow) cond((t_1, t_2, t_3))</td>
</tr>
<tr>
<td>0</td>
<td>map (\rightarrow) 0</td>
</tr>
<tr>
<td>succ (t)</td>
<td>map (\rightarrow) succ((t))</td>
</tr>
<tr>
<td>pred (t)</td>
<td>map (\rightarrow) pred((t))</td>
</tr>
<tr>
<td>iszero (t)</td>
<td>map (\rightarrow) iszero((t))</td>
</tr>
<tr>
<td>Bool</td>
<td>map (\rightarrow) bool</td>
</tr>
<tr>
<td>Nat</td>
<td>map (\rightarrow) nat</td>
</tr>
</tbody>
</table>
Step 2: Implement the Typing Relation

typeOf(true, bool).
typeOf(false, bool).

typeOf(0, nat).
typeOf(succ(T1), nat) :- ofType(T1, nat).

typeOf(cond(T1, T2, T3), T) :- ofType(T1, bool),
                             ofType(T2, T),
                             ofType(T3, T).

typeOf(pred(T1), nat) :- ofType(T1, nat).
typeOf(iszero(T1), bool) :- ofType(T1, nat).
Step 3: Develop Testing Framework

genericTest(T) :-
    typeof(T, Type),
    write('Result = '),
    write(T), write(': '), write(Type),
    fail.

test_01 :-
    genericTest(succ(succ(0))).

test_02 :-
    genericTest(succ(pred(succ(pred(0))))).

batchTest :-
    write('test_01:
'),
    not(test_01), nl, nl,
    write('test_02:
'),
    not(test_02), nl, nl.
Outline

Types
- Evaluation Rules
- Typing Rules

Properties of the Typing Relation
- The Inversion Lemma
- Prolog Implementation

Reasoning Involving Types
- Progress
- Preservation
Lemma:

1. If \( v \) is a value of type \( \text{Bool} \), then \( v \) is either \( \text{true} \) or \( \text{false} \).
2. If \( v \) is a value of type \( \text{Nat} \), then \( v \) is a numeric value.

Proof:
### Part 1

Recall the syntax of values:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v$</td>
<td>::= values</td>
</tr>
<tr>
<td></td>
<td>$true$</td>
</tr>
<tr>
<td></td>
<td>$false$</td>
</tr>
<tr>
<td>$nv$</td>
<td>::= numeric values</td>
</tr>
<tr>
<td></td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$succ \ nv$</td>
</tr>
</tbody>
</table>

For part 1,

If $v$ is $true$ or $false$, the result is immediate. Also note that in this case $v$ cannot be $0$ or $succ \ nv$, since the inversion lemma tells us that $v$ would then have type $Nat$, not $Bool$. Part 2 is similar.
Part 1

Recall the syntax of values:

\[ v ::= \]
\[ \text{true} \]
\[ \text{false} \]
\[ nv \]

\[ nv ::= \]
\[ 0 \]
\[ \text{succ } nv \]

For part 1, if \( v \) is \textit{true} or \textit{false}, the result is immediate.
Part 1

Recall the syntax of values:

\[ v ::= \]
\[ \quad \text{true value} \]
\[ \quad \text{false value} \]
\[ \quad \text{numeric value} \]

\[ nv ::= \]
\[ \quad 0 \quad \text{zero value} \]
\[ \quad \text{succ } nv \quad \text{successor value} \]

For part 1, if \( v \) is \textit{true} or \textit{false}, the result is immediate. Also note that in this case \( v \) cannot be \( 0 \) or \textit{succ } \( nv \), since the inversion lemma tells us that \( v \) would then have type \textit{Nat}, not \textit{Bool}. 
Part 1

Recall the syntax of values:

\[ v ::= \]
\[ true \]
\[ false \]
\[ nv \]

\[ nv ::= \]
\[ 0 \]
\[ succ \ nv \]

For part 1, if \( v \) is \textit{true} or \textit{false}, the result is immediate. Also note that in this case \( v \) cannot be \textit{0} or \textit{succ} \( nv \), since the inversion lemma tells us that \( v \) would then have type \textit{Nat}, not \textit{Bool}. Part 2 is similar.
Theorem: Suppose $t$ is a well-typed term (that is, $t : T$ for some type $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$. 
**Progress**

**Theorem:** Suppose $t$ is a well-typed term (that is, $t : T$ for some type $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

**Proof:**
**Theorem:** Suppose $t$ is a well-typed term (that is, $t : T$ for some type $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

**Proof:**

By induction on a derivation of $t : T$. 

---

**Progress**
Aside

Recall that our typing relation consists of the following typing rules:

\[ T_{\text{rule}} = \{ \]
\[ \quad \text{T-True, T-False, T-If,} \]
\[ \quad \text{T-Zero, T-Succ, T-Pred, T-IsZero} \]
\[ \} \]

Also recall that we have the following evaluation rules:

\[ E_{\text{rule}} = E_{\text{bool}} \cup E_{\text{nat}} \]

\[ E_{\text{bool}} = \{ \text{E-IfTrue, E-IfFalse, E-If} \} \]

\[ E_{\text{nat}} = \{ \]
\[ \quad \text{E-Succ, E-PredZero, E-PredSucc, E-Pred,} \]
\[ \quad \text{E-IsZeroZero, E-IsZeroSucc, E-IsZero} \]
\[ \} \]
Aside

For each typing rule \( r \in T_{\text{rule}} \), we will consider the implications of a typing derivation ending with \( r \).

\[
\begin{array}{ccc}
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
(r_i \in T_{\text{rule}}) & (r_j \in T_{\text{rule}}) & (r) \\
\hline
t : T
\end{array}
\]

In particular, we will need to consider which evaluation rules (if any) can be applied to \( t \). Essentially, our analysis covers:

\[ T_{\text{rule}} \times E_{\text{rule}} \]
Theorem: Suppose \( t \) is a well-typed term (that is, \( t : T \) for some type \( T \)). Then either \( t \) is a value or else there is some \( t' \) with \( t \rightarrow t' \).

Case \( r = T-True \). This implies that \( t = true \) (and no evaluation rules apply). Since \( t \) is value, the theorem holds in this case.
**Theorem:** Suppose \( t \) is a well-typed term (that is, \( t : T \) for some type \( T \)). Then either \( t \) is a value or else there is some \( t' \) with \( t \rightarrow t' \).

**Case** \( r = T-False \). This implies that \( t = false \) (and no evaluation rules apply). Since \( t \) is value, the theorem holds in this case.
Theorem: Suppose $t$ is a well-typed term (that is, $t : T$ for some type $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

Case $r = T$-Zero. This implies that $t = 0$ (and no evaluation rules apply). Since $t$ is value, the theorem holds in this case.
**Progress Proof**

*Theorem:* Suppose $t$ is a well-typed term (that is, $t : T$ for some type $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

**Case** $r = \text{T-If}$. This implies (by the inversion lemma) that following must hold for $t$:

$$
t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3
$$

$$
t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T
$$
**Progress Proof**

**Theorem:** Suppose $t$ is a well-typed term (that is, $t : T$ for some type $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

**Case** $r = T$-If. This implies (by the inversion lemma) that following must hold for $t$:

$$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$$

$$t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$$

There are three *evaluation rules* that need to be considered: E-IfTrue, E-IfFalse, and E-If.

We assume (by the induction hypothesis) that the theorem holds for $t_1$. This allows us to conclude that (1) $t_1$ is a value or (2) there exists a $t_1'$ such that $t_1 \rightarrow t_1'$. 
Progress Proof

Theorem: Suppose \( t \) is a well-typed term (that is, \( t : T \) for some type \( T \)). Then either \( t \) is a value or else there is some \( t' \) with \( t \rightarrow t' \).

Case \( r = T-If \). This implies (by the inversion lemma) that following must hold for \( t \):

\[
\begin{align*}
t &= \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\
t_1 &: \text{Bool} \\
t_2 &: T \\
t_3 &: T
\end{align*}
\]

Subcase \( t_1 \) is a value. We also know that \( t_1 : \text{Bool} \) and hence \( t_1 \) must either be \text{true} or \text{false} (by the canonical forms lemma). In this case, the reduction \( t \rightarrow t' \) is accomplished via the evaluation rule E-IfTrue or E-IfFalse.
**Theorem:** Suppose $t$ is a well-typed term (that is, $t : T$ for some type $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

**Case** $r = T$-If. This implies (by the inversion lemma) that following must hold for $t$:

$$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$$

$t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

**Subcase** There exists a $t'_1$ such that $t_1 \rightarrow t'_1$. In this case, the reduction $t \rightarrow t'$ is accomplished via the *evaluation rule* E-If. Thus, for the case T-If, our analysis accounts for all evaluation rules.
Remaining Cases

*Theorem:* Suppose $t$ is a well-typed term (that is, $t : T$ for some type $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

Case $r = T$-Zero. Here $t = 0$ so no evaluation rules apply.

Case $r = T$-Succ. Here $t = \text{succ } t_1$ so only the evaluation rule E-Succ must be considered (in the case where $t_1$ is not a value).

Case $r = T$-Pred. Here $t = \text{pred } t_1$ and the evaluation rules E-PredZero, E-PredSucc and E-Pred must be considered.

Case $r = T$-IsZero. Here $t = \text{iszero } t_1$ and the evaluation rules E-IsZeroZero, E-IsZeroSucc and E-IsZero must be considered.
Preservation

*Theorem:* If $t : T$ and $t \rightarrow t'$, then $t' : T$. 
Preservation

**Theorem:** If $t : T$ and $t \rightarrow t'$, then $t' : T$.

**Proof:** By induction on a typing derivation of $t : T$. 
Preservation

*Theorem:* If $t : T$ and $t \rightarrow t'$, then $t' : T$.

*Proof:* By induction on a typing derivation of $t : T$.

Case $r = T$-True. In this case, $t = true$. Therefore, $t \rightarrow t'$ is not possible, and the theorem is vacuously true.
Preservation

**Theorem:** If $t : T$ and $t \rightarrow t'$, then $t' : T$.

**Proof:** By induction on a typing derivation of $t : T$.

**Case** $r = T\text{-False}$. In this case, $t = \text{false}$. Therefore, $t \rightarrow t'$ is not possible, and the theorem is vacuously true.
**Preservation**

*Theorem:* If $t : T$ and $t \rightarrow t'$, then $t' : T$.

*Proof:* By induction on a typing derivation of $t : T$.

**Case $r = T-\text{If}$:** In this case, it follows that:

\[
\begin{align*}
  t &= \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\
  t_1 & : \text{Bool} \\
  t_2 & : T \\
  t_3 & : T
\end{align*}
\]
**Preservation**

*Theorem:* If $t : T$ and $t \rightarrow t'$, then $t' : T$.

*Proof:* By induction on a typing derivation of $t : T$.

**Case $r = T-If.$** In this case, it follows that:

$$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$$

$t_1 : \text{Bool}$  
$t_2 : T$  
$t_3 : T$

In this context, there are three evaluation rule by which $t \rightarrow t'$ can be derived: E-IfTrue, E-IfFalse, E-If.
Preservation

**Theorem:** If $t : T$ and $t \rightarrow t'$, then $t' : T$.

**Proof:** By induction on a typing derivation of $t : T$.

**Case** $r = T$-$If$. In this case, it follows that:

$$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$$

$t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

**Subcase** Evaluation rule $= E$-$If$True. In this case, $t_1 = \text{true}$ and $t' = t_2$. We also know that $t_2 : T$, and hence $t' : T$ as required.
Preservation

Theorem: If $t : T$ and $t \rightarrow t'$, then $t' : T$.

Proof: By induction on a typing derivation of $t : T$.

Case $r = T$-$\text{If}$. In this case, it follows that:

$$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$$

$t_1 : \text{Bool}$, $t_2 : T$, $t_3 : T$

Subcase Evaluation rule $= \text{E-IfFalse}$. In this case,

$t_1 = \text{false}$ and $t' = t_3$. We also know that $t_3 : T$,
and hence $t' : T$ as required.
Preservation

**Theorem:** If $t : T$ and $t \rightarrow t'$, then $t' : T$.

**Proof:** By induction on a typing derivation of $t : T$.

**Case** $r = T$-$If$. In this case, it follows that:

\[
 t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\
 t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T
\]

**Subcase** Evaluation rule = E-If. In this case, $t_1 \rightarrow t'_1$ and

\[
 t' = \text{if } t'_1 \text{ then } t_2 \text{ else } t_3
\]

Combining the inductive assumption with $t_1 : \text{Bool}$ lets us conclude $t'_1 : \text{Bool}$. We also know $t_2 : T$ and $t_3 : T$. Therefore we can conclude that $t' : T$. 

Remaining Cases

**Theorem:** If $t : T$ and $t \rightarrow t'$, then $t' : T$.

**Case** $r = T-Zero$. Here $t = 0$ so no evaluation rules apply.

**Case** $r = T-Succ$. Here $t = succ \ t_1$ so only the evaluation rule $E-Succ$ must be considered (and only in the case where $t_1$ is not a value).

**Case** $r = T-Pred$. Here $t = pred \ t_1$ and the evaluation rules $E-PredZero$, $E-PredSucc$ and $E-Pred$ must be considered.

**Case** $r = T-IsZero$. Here $t = iszero \ t_1$ and the evaluation rules $E-IsZeroZero$, $E-IsZeroSucc$ and $E-IsZero$ must be considered.