Midterm Exam

Examsolutions on webpage.

Look at your exam in Cheryl Hickey’s office.

You can pick up your exam from Cheryl after October 26.

Submit regrade request (in writing) before October 26.

Typos

Midterm Exam

Fall 2005

Software Foundations

CIS 500

19 October, 2005

CIS 500, 19 October, 2005
Type Systems

Currently, active and successful topic in PL research

A "light-weight" formal methods "enabling technology" for all sorts of other things, e.g.
language-based security

The skeleton" around which modern programming languages are often designed

Approaches to Typing

- Strongly typed languages prevent programs from accessing private data, corrupting memory, crashing the machine, etc.
- Weakly typed languages do not.
- Statically typed languages perform type-consistency checks at program entry.
- Dynamically typed languages delay these checks until programs are executed.

Strictly speaking, Java should be called "mostly static".

Plan

For today, we'll go back to the simple language of arithmetic and booleans.

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 $T$ that classifies terms according to the shapes of the values that result from evaluating them.
4. Check that the typing relation is sound in the sense that, if $T$, then evaluation of the term $t$ will not get stuck.

We'll spend a good part of the rest of the semester adding features to this type system.

We'll develop a simple type system for the lambda-calculus next week.

Follow-up TAPL Ch. 9.

For today, we'll go back to the simple language of arithmetic and booleans.

For today, we'll go back to the simple language of arithmetic and booleans.

Plan

1. Follow-up TAPL Ch. 9.
2. We'll spend a good part of the rest of the semester adding features to this type system.
3. Next week, we'll develop a simple type system for the lambda-calculus.
4. We'll develop a simple type system for the lambda-calculus.
### Arithmetic Expressions

**Syntax**

- \( t \) ::= \( \text{terms} \)
- \( \text{true} \)
- \( \text{false} \)
- \( \text{if} \text{then} t \text{else} t \)
- \( \text{constant} t \)
- \( \text{conditional} \)
- \( \text{0} \)
- \( \text{successor} \)
- \( \text{predecessor} \)
- \( \text{iszero} t \)

**Evaluation Rules**

- \( \text{E-IfTrue} \): \( \text{if} \text{true} \text{then} t \text{else} t \)  
  \[ t \]
- \( \text{E-IfFalse} \): \( \text{if} \text{false} \text{then} t \text{else} t \)  
  \[ \neg t \]
- \( \text{E-If} \): \( \text{if} t_1 \text{then} t \text{else} t \)  
  \[ t_1 \]
- \( \text{E-Succ} \): \( \text{successor} \)  
  \[ \text{succt} \]
- \( \text{E-Pred} \): \( \text{predecessor} \)  
  \[ \text{predt} \]
- \( \text{E-Iszero} \): \( \text{iszero} \)  
  \[ \text{iszerot} \]

**Types**

In this language, values have two possible "shapes": they are either booleans or numbers.
Typing Rules

(T-True)
\[
\begin{aligned}
\text{succ } t_1 : \text{Nat} \\
\text{if } t_1 : \text{Nat} \\
\end{aligned}
\]

(T-False)
\[
\begin{aligned}
0 : \text{Nat} \\
\end{aligned}
\]

(T-Zero)

Typtie Rules

(T-If)
\[
\begin{aligned}
\text{if } t_1 : \text{Nat} \text{ then } t_2 \text{ else } t_3 : T \\
\end{aligned}
\]

(T-False)
\[
\begin{aligned}
\text{false : Boot} \\
\end{aligned}
\]

(T-True)
\[
\begin{aligned}
\text{true : Boot} \\
\end{aligned}
\]
Typing Rules

0: \text{Nat} (T-Zero)

1: \text{Nat} (T-Succ)

\text{succt} 1: \text{Nat} (T-Succ)

\text{predt} 1: \text{Nat} (T-Pred)

\text{iszerot} 1: \text{Bool} (T-IsZero)

Typing Derivations

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

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. Proofsof properties about the typing relation often proceed by induction on typing derivations.

\begin{align*}
\text{T-Zero} & : \text{Nat} \\
\text{T-IsZero} & : \text{Bool} \\
\text{T-Zero} & : \text{Nat} \\
\text{T-If} & : \text{Nat} \\
\text{T-If} & : \text{T} \\
\text{T-If} & : \text{T} \\
\text{T-Zero} & : \text{T} \\
\text{T-Zero} & : \text{T} \\
\end{align*}

Using this rule, we cannot assign a type to
\[
\text{if } \text{true } \text{then } 0 \text{ else } 0.
\]

\begin{align*}
\text{T-Zero} & : \text{Nat} \\
\text{T-Pred} & : \text{Nat} \\
\text{T-Zero} & : \text{T} \\
\text{T-Zero} & : \text{T} \\
\end{align*}

Using this rule, we cannot assign a type to
\[
\text{if true } \text{then } 0 \text{ else } \text{true}.
\]
Properties of the Typing Relation

Type Safety

Theorem: If \( t : T \) and \( t \rightarrow t' \) then \( t' \in \{ \text{val} \} \).

We usually prove type safety by showing the following two properties:

1. Progress: A well-typed term is not stuck.
2. Preservation: A well-typed term is not stuck.

Inversion

1. If \( \text{true} : R \), then \( R = \text{Bool} \).
2. If \( \text{false} : R \), then \( R = \text{Bool} \).
3. If \( \text{if} t_1 \text{then} t_2 \text{else} t_3 : R \), 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 \( t_1 : \text{Nat} \).

Proof:

This leads directly to a recursive algorithm for calculating the type of a term...
Inversion Lemma:
1. If true, then $R = \text{Bool}$.
2. If false, then $R = \text{Bool}$.
3. If if t1 then t2 else t3, then $t_1: \text{Bool}$, $t_2: R$, and $t_3: R$.
4. If 0, then $R = \text{Nat}$.
5. If pred t1, then $t_1: \text{Nat}$ and $t_1: \text{Nat}$.
6. If succ t1, then $t_1: \text{Nat}$ and $t_1: \text{Nat}$.
7. If itzero t1, then $t_1: \text{Nat}$ and $t_1: \text{Nat}$.

Proof:
... This leads directly to a recursive algorithm for calculating the type of a term.

Typechecking Algorithm:
$\text{typeof}(t) = \begin{cases} \text{Bool} & \text{if } t = \text{true} \\ \text{else if } t = \text{false} \\ \text{if } \text{typeof}(t_1) = \text{Bool} \text{ and typeof}(t_2) = \text{typeof}(t_3) \text{ then } \text{typeof}(t_2) \\ \text{else "not typable"} & \text{otherwise} \end{cases}$

Canonical Forms Lemma:
1. If $v$ is a value of type $\text{Bool}$, then $v$ is either true or false.
2. If $v$ is a value of type $\text{Nat}$, then $v$ is a numeric value.

Proof:
...
Theorem: Suppose \( t \) is a well-typed term (that is, for some \( T \), \( T \rightarrow t \)). Then

Proof: By induction on a derivation of \( T \).

1. \( t \) is a value or else there is some \( T \) with \( T \rightarrow t \).
2. \( t \) is a value or else there is some \( T \) with \( T \rightarrow t \).

The in these cases are immediate, since

\( T \rightarrow t \)

then, by
t

\( T \rightarrow t \):

Suppose \( t \) is a well-typed term (that is, for some \( T \), \( T \rightarrow t \)). Then

Proof: By induction on a derivation of \( T \).

1. \( t \) is a value or else there is some \( T \) with \( T \rightarrow t \).
2. \( t \) is a value or else there is some \( T \) with \( T \rightarrow t \).

Theorem: Suppose \( t \) is a well-typed term (that is, for some \( T \), \( T \rightarrow t \)). Then

Proof: By induction on a derivation of \( T \).

1. \( t \) is a value or else there is some \( T \) with \( T \rightarrow t \).
2. \( t \) 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 $T$). Then

either $t$ is a value or else there is some $t_0$ with $t \neq t_0$.

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

Case $T$-I: $t = t_1 \text{ then } t_2 \text{ else } t_3$.

CASE $T$-I: If $t_1$ is a value, then the canonical forms lemma tells us that $t$ is a value.

Otherwise, there is some $t_0$ with $t \neq t_0$.

Theorem: Suppose $t$ is a well-typed term (that is, $t : T$ for some $T$). Then

either $t$ is a value or else there is some $t_0$ with $t \neq t_0$.

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

Case $T$-I: $t = t_1 \text{ then } t_2 \text{ else } t_3$.

Case $T$-I: If $t_1$ is a value, then the canonical forms lemma tells us that $t$ is a value.

Otherwise, there is some $t_0$ with $t \neq t_0$.

Preservation

Theorem: If $t : T$ and $t \neq t_0$, then $t_0 : T$.

Proof: ...