Typed Arithmetic Expressions

Principles of Programming Languages

CSE 526

1. Typed Arithmetic Expressions
2. Simply-Typed $\lambda$-Calculus
Types

- Types are way to classify terms (programs)
- Meaningful terms (e.g. those that do not get stuck) should have a type
- A **typing relation** relates terms to types.
- Two ways to define semantics:
  - *Curry-style*: Define terms and their semantics, then define types to reject those terms whose semantics are problematic.
  - *Church-style*: Define terms and a typing relation, then define semantics only for well-typed terms.
Typed arithmetic expressions

**Terms**

\[ t ::= \text{true} \]  
\[ \quad | \quad \text{false} \]  
\[ \quad | \quad \text{if}(t,t,t) \]  
\[ \quad | \quad 0 \]  
\[ \quad | \quad \text{succ } t \]  
\[ \quad | \quad \text{pred } t \]  
\[ \quad | \quad \text{iszero } t \]  

**Types**

\[ T ::= \]  
\[ \quad \text{Bool} \]  
\[ \quad | \quad \text{Nat} \]
Typing relation for arithmetic expressions

The smallest binary relation “:” between types and terms satisfying all instances of the following inference rules:

- **T-TRUE**
  - `true : Bool`

- **T-FALSE**
  - `false : Bool`

- **T-IF**
  - `if(t_1, t_2, t_3) : T`
  - premises: `t_1 : Bool`, `t_2 : T`, `t_3 : T`

- **T-ZERO**
  - `0 : Nat`

- **T-SUCC**
  - `succ(t_1) : Nat`
  - premise: `t_1 : Nat`

- **T-PRED**
  - `pred(t_1) : Nat`
  - premise: `t_1 : Nat`

- **T-ISZERO**
  - `iszero(t_1) : Bool`
  - premise: `t_1 : Nat`
Properties of the typing relation

A term $t$ is said to be well-typed if there is a type $T$ such that $t : T$.

- **Uniqueness of types:** Each term $t$ has at most one type $T$ such that $t : T$. 
Properties of the typing relation

A term $t$ is said to be *well-typed* if there is a type $T$ such that $t : T$.

- **Uniqueness of types**: Each term $t$ has at most one type $T$ such that $t : T$.
- **Progress**: For every well-typed term $t$, either $t$ is a value or there is a $t'$ such that $t \rightarrow t'$. 

The term $t$ is said to be *progressing* if $t \rightarrow t'$.
Properties of the typing relation

A term \( t \) is said to be \textit{well-typed} if there is a type \( T \) such that \( t : T \).

- **Uniqueness of types:** Each term \( t \) has at most one type \( T \) such that \( t : T \).
- **Progress:** For every well-typed term \( t \), either \( t \) is a value or there is a \( t' \) such that \( t \rightarrow t' \).
- **Preservation:** If \( t : T \) and \( t \rightarrow t' \) then \( t' : T \).
A term \( t \) is said to be *well-typed* if there is a type \( T \) such that \( t : T \).

- **Uniqueness of types:** Each term \( t \) has at most one type \( T \) such that \( t : T \).
- **Progress:** For every well-typed term \( t \), either \( t \) is a value or there is a \( t' \) such that \( t \rightarrow t' \).
- **Preservation:** If \( t : T \) and \( t \rightarrow t' \) then \( t' : T \).
- **Safety = Progress + Preservation**
Recall booleans, numbers and operations on them can be encoded in the pure $\lambda$-calculus.

Nevertheless, it is convenient to include primitive data types in the calculus as well.

$\lambda B$ is an enriched calculus with boolean data types $\text{true}$ and $\text{false}$, and operation $\text{if}$.

$\lambda x. \lambda y. \text{if}(x, y, x)$ is a term in $\lambda B$.

$\lambda NB$ is a similarly enriched calculus with numbers and booleans.

$\lambda x. \lambda y. \text{if}(\text{iszero}(x), \text{succ}(y), x)$ is a term in $\lambda NB$.
Simply-Typed $\lambda$-Calculus

Syntax:

\[ t ::= \]

\[ \chi \quad \text{Variable} \]
\[ \lambda x : T . t \quad \text{Abstraction} \]
\[ t t \quad \text{Application} \]

Terms

Types

$T ::= \_

\text{Base types} \]
\[ T \rightarrow T \quad \text{type of functions} \]

Contexts

$\Gamma ::= \_

\emptyset \quad \text{Empty Context} \]
\[ \Gamma , x : T \quad \text{Variable Binding} \]

Programming Languages

Typed Arithmetic Expressions

CSE 526 7 / 11
Simply-Typed λ-Calculus

Syntax:

\[
\begin{align*}
  t &::= &\text{Terms} \\
  \quad & x \text{ Variable} \\
  & | \lambda x : T. t \text{ Abstraction} \\
  & | t \; t \text{ Application} \\

  T &::= &\text{Types} \\
  \quad & A \text{ Base types} \\
  & | T \to T \text{ type of functions}
\end{align*}
\]
Simply-Typed $\lambda$-Calculus

Syntax:

\[
\begin{align*}
t & ::= \\
    & \quad x \quad \text{Variable} \\
    & \quad \lambda x : T. \ t \quad \text{Abstraction} \\
    & \quad \ t \ t \quad \text{Application} \\

T & ::= \\
    & \quad A \quad \text{Base types} \\
    & \quad T \to T \quad \text{type of functions} \\

\Gamma & ::= \\
    & \quad \emptyset \quad \text{Empty Context} \\
    & \quad \Gamma, x : T \quad \text{Variable Binding}
\end{align*}
\]
Evaluation (Call-By-Value)

Small-Step Evaluation Relation for simply-typed $\lambda$-calculus:

\[
\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \quad \text{E-APP1}
\]

\[
\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2} \quad \text{E-ABS2}
\]

\[
(\lambda x : T. \ t_1) \ v_2 \rightarrow [x \mapsto v_2]t_1 \quad \text{E-APPABS}
\]
Typing Relation

\[
\Gamma 
\begin{array}{c}
  x : T 
  \in \Gamma
\end{array}
\Rightarrow
\Gamma \vdash x : T
\]

\[T-VAR\]
Typing Relation

\[
\begin{align*}
\Gamma & \vdash x : T \\
\text{T-VAR} & \\
\Gamma, x : T_1 & \vdash t_2 : T_2 \\
\Gamma & \vdash \lambda x : T_1. \ t_2 : T_1 \to T_2 \\
\text{T-ABS} &
\end{align*}
\]
Typing Relation

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{T-VAR}
\]

\[
\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. \ t_2 : T_1 \to T_2} \quad \text{T-ABS}
\]

\[
\frac{\Gamma \vdash s : T_1 \to T_2 \quad \Gamma \vdash t : T_1}{\Gamma \vdash (s \ t) : T_2} \quad \text{T-APP}
\]
Properties of the typing relation

A term $t$ is said to be well-typed in context $\Gamma$ if there is a type $T$ such that $t : T$.

- **Uniqueness of types:** In a context $\Gamma$, each term $t$ has at most one type $T$ such that $t : T$.
- **Progress:** For every closed, well-typed term $t$, either $t$ is a value or there is a $t'$ such that $t \rightarrow t'$.
- **Preservation under substitution:** If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$
- **Preservation:** If $\Gamma \vdash t : T$ and $t \rightarrow t'$ then $\Gamma \vdash t' : T$.
- **Safety** = Progress + Preservation
Erasure and Typability

*erase* is a function that maps simply-typed $\lambda$-terms to untyped $\lambda$-terms.

\[
\begin{align*}
erase(x) &= x \\
erase(\lambda x : T. t) &= \lambda x. \ erase(t) \\
erase(t_1 t_2) &= erase(t_1) \ erase(t_2)
\end{align*}
\]

- If $t \rightarrow t'$ under typed evaluation relation, then $erase(t) \rightarrow erase(t')$

An untyped term $m$ is *typable* if there is some simply-typed term $t$ and type $T$ and context $\Gamma$ such that $erase(t) = m$ and $\Gamma \vdash t : T$.

Not every untyped lambda term is typable!
Erasure and Typability

*erase* is a function that maps simply-typed \( \lambda \)-terms to untyped \( \lambda \)-terms.

\[
\begin{align*}
erase(x) &= x \\
erase(\lambda x : T. t) &= \lambda x. erase(t) \\
erase(t_1 t_2) &= erase(t_1) \ erase(t_2)
\end{align*}
\]

- If \( t \rightarrow t' \) under typed evaluation relation, then \( erase(t) \rightarrow erase(t') \)
- If \( erase(t) \rightarrow m' \), then there is a simply-typed term \( t' \) such that \( t \rightarrow t' \) (under typed evaluation relation) and \( erase(t') = m' \)
Erasure and Typability

`erase` is a function that maps simply-typed \( \lambda \)-terms to untyped \( \lambda \)-terms.

\[
erase(x) = x \\
erase(\lambda x : T. \ t) = \lambda x. \ erase(t) \\
erase(t_1 \ t_2) = erase(t_1) \ erase(t_2)
\]

- If \( t \rightarrow t' \) under typed evaluation relation, then \( erase(t) \rightarrow erase(t') \)
- If \( erase(t) \rightarrow m' \), then there is a simply-typed term \( t' \) such that \( t \rightarrow t' \) (under typed evaluation relation) and \( erase(t') = m' \)
- An untyped term \( m \) is **typable** if there is some simply-typed term \( t \) and type \( T \) and context \( \Gamma \) such that \( erase(t) = m \) and \( \Gamma \vdash t : T \).
Erasure and Typability

erase is a function that maps simply-typed \( \lambda \)-terms to untyped \( \lambda \)-terms.

\[
\begin{align*}
erase(x) &= x \\
erase(\lambda x : T. t) &= \lambda x. erase(t) \\
erase(t_1 t_2) &= erase(t_1) \ erase(t_2)
\end{align*}
\]

- If \( t \rightarrow t' \) under typed evaluation relation, then \( erase(t) \rightarrow erase(t') \)
- If \( erase(t) \rightarrow m' \), then there is a simply-typed term \( t' \) such that \( t \rightarrow t' \) (under typed evaluation relation) and \( erase(t') = m' \)
- An untyped term \( m \) is **typable** if there is some simply-typed term \( t \) and type \( T \) and context \( \Gamma \) such that \( erase(t) = m \) and \( \Gamma \vdash t : T \).
- Not every untyped lambda term is typable!

Example: \((x \ x)\)