Types and Programming Languages

Lecture 4. Types, the simply typed λ-calculus

Xiaojuan Cai

cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Spring, 2016

Outline

Typed arithmetic expressions

- Typing relation
- Safety = Progress + Preservation

Simply typed λ-calculus

- Function types

By anonymous

Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.

By anonymous

Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.

A: The definitions are almost always wrong.

Arithmetic expressions

t	::=		*terms*
		`true`	*constant true*
		`false`	*constant false*
		`if` t `then` t `else` t	*conditional*
		0	*constant zero*
		`succ` t	*successor*
		`pred` t	*predecessor*
		`iszero` t	*zero test*
v	::=		*values*
		`true`	*true value*
		`false`	*false value*
		`nv`	*numeric value*
nv	::=		*numeric values*
		0	*zero value*
		`succ nv`	*successor value*

Types

- Recall that evaluating a term can either result in a value or else get stuck at some stage, by reaching a term like `pred false`.

Types

- Recall that evaluating a term can either result in a value or else get stuck at some stage, by reaching a term like `pred false`.
- In fact, we can tell *stuck terms* without actually evaluating it.

Types

- Recall that evaluating a term can either result in a value or else get stuck at some stage, by reaching a term like `pred false`.
- In fact, we can tell *stuck terms* without actually evaluating it.
- Coming soon: If a term is well typed, i.e., it has some type T, then it never get stuck (never goes *wrong*).

Typing relation

The typing relation for arithmetic expressions, written "$t : T$, is defined by a set of inference rules assigning types to terms.

`T`	::=		*types*
		`Bool`	*type of booleans*
		`Nat`	*type of natural numbers*

Typing relation

The typing relation for arithmetic expressions, written "$t : T$, is defined by a set of inference rules assigning types to terms.

$$
\begin{array}{lll}
\mathtt{T} ::= & & \textit{types} \\
& \mathtt{Bool} & \textit{type of booleans} \\
& \mathtt{Nat} & \textit{type of natural numbers}
\end{array}
$$

Typing rules:

$$
\text{T-True}\ \frac{}{\mathtt{true} : \mathtt{Bool}} \qquad \text{T-False}\ \frac{}{\mathtt{false} : \mathtt{Bool}} \qquad \text{T-Zero}\ \frac{}{\mathtt{0} : \mathtt{Nat}}
$$

$$
\text{T-Succ}\ \frac{t_1 : \mathtt{Nat}}{\mathtt{succ}\ t_1 : \mathtt{Nat}} \qquad \text{T-Pred}\ \frac{t_1 : \mathtt{Nat}}{\mathtt{pred}\ t_1 : \mathtt{Nat}}
$$

$$
\text{T-If}\ \frac{t_1 : \mathtt{Bool} \quad t_2 : \mathtt{T} \quad t_3 : \mathtt{T}}{\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : \mathtt{T}} \qquad \text{T-IsZero}\ \frac{t_1 : \mathtt{Nat}}{\mathtt{iszero}\ t_1 : \mathtt{Bool}}
$$

Uniqueness of types

When reasoning about the typing relation, we will often inverse the typing relation.

Uniqueness of types

When reasoning about the typing relation, we will often inverse the typing relation.

Lemma 8.2.2:

1. If $\texttt{true} : \texttt{R}$ or $\texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$;
2. If $\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Bool}$, $\texttt{t}_2 : \texttt{R}$, and $\texttt{t}_3 : \texttt{R}$.
3. If $\texttt{0} : \texttt{R}$, or $\texttt{succ } \texttt{t}_1 : \texttt{R}$, or $\texttt{pred } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
4. If $\texttt{iszero } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $\texttt{t}_1 : \texttt{Nat}$.

Uniqueness of types

When reasoning about the typing relation, we will often inverse the typing relation.

Lemma 8.2.2:

1. If $\mathtt{true} : \mathtt{R}$ or $\mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$;
2. If $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then $\mathtt{t}_1 : \mathtt{Bool}$, $\mathtt{t}_2 : \mathtt{R}$, and $\mathtt{t}_3 : \mathtt{R}$.
3. If $\mathtt{0} : \mathtt{R}$, or $\mathtt{succ}\ \mathtt{t}_1 : \mathtt{R}$, or $\mathtt{pred}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$.
4. If $\mathtt{iszero}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$ and $\mathtt{t}_1 : \mathtt{Nat}$.

Thoerem 8.2.4 [Uniqueness of types]: Each term t has at most one type.

Uniqueness of types

When reasoning about the typing relation, we will often inverse the typing relation.

Lemma 8.2.2:

1. If $\mathtt{true} : \mathtt{R}$ or $\mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$;
2. If $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then $\mathtt{t}_1 : \mathtt{Bool}$, $\mathtt{t}_2 : \mathtt{R}$, and $\mathtt{t}_3 : \mathtt{R}$.
3. If $\mathtt{0} : \mathtt{R}$, or $\mathtt{succ}\ \mathtt{t}_1 : \mathtt{R}$, or $\mathtt{pred}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$.
4. If $\mathtt{iszero}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$ and $\mathtt{t}_1 : \mathtt{Nat}$.

Thoerem 8.2.4 [Uniqueness of types]: Each term t has at most one type.

Above theorem does not hold for languages with subtyping rules.

The most basic property of type system

Safety = Progress + Preservation

- Progress: A well-typed term is not stuck (either it is a value or it can take a step according to the evaluation rules).
- Preservation: If a well-typed term takes a step of evaluation, then the resulting term is also well typed.

Porgress

Lemma 8.3.1 [Canonical forms]:

- If v is a value of type `Bool`, then v is either `true` or `false`.
- If v is a value of type `Nat`, then v is a numeric value according to the grammar.

Theorem 8.3.2 [Progress]: Suppose t is a well-typed term (that is, $t : \mathtt{T}$ for some $\mathtt{T}$). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof. By induction on a derivation of $t : \mathtt{T}$.

Preservation

Theorem 8.3.3 [Preservation]: If $t : \mathtt{T}$ and $t \longrightarrow t'$, then $t' : \mathtt{T}$.

Proof. Either by induction on a derivation of $t : \mathtt{T}$, or by induction on a derivation of $t \longrightarrow t'$.

Outline

Typed arithmetic expressions

- Typing relation
- Safety = Progress + Preservation

Simply typed λ-calculus

- Function types

Add types to λ-calculus

Coming soon: A typing relation for variables, abstractions, and applications that

- maintain type safety: satisfy the type progress and preservation;
- are not to conservative: they should assign types to most of the programs we actually care about writing.

Add types to λ-calculus

Coming soon: A typing relation for variables, abstractions, and applications that

- maintain type safety: satisfy the type progress and preservation;
- are not to conservative: they should assign types to most of the programs we actually care about writing.

Turing completeness of λ-calculus implies that there is no hope of giving an exact type analysis for these primitives. For example:

$$\texttt{if}\ \langle\texttt{long and tricky computation}\rangle\ \texttt{then true else}\ (\lambda \mathtt{x}.\mathtt{x})$$

Arrow type

For a function,

1. we care about the types of both arguments and results:

$$\text{arrow type } \mathtt{T} \rightarrow \mathtt{T}$$

Note the difference between $\mathtt{T} \rightarrow \mathtt{T} \rightarrow \mathtt{T}$ and $(\mathtt{T} \rightarrow \mathtt{T}) \rightarrow \mathtt{T}$

Arrow type

For a function,

1. we care about the types of both arguments and results:

   $$\text{arrow type } \mathtt{T} \to \mathtt{T}$$

   Note the difference between $\mathtt{T} \to \mathtt{T} \to \mathtt{T}$ and $(\mathtt{T} \to \mathtt{T}) \to \mathtt{T}$
2. the type of an abstraction relies on the type of argument, e.g.

   $$\lambda x.x : \mathtt{Bool} \to \mathtt{Bool} \textit{ or } \lambda x.x : \mathtt{Nat} \to \mathtt{Nat}$$

Arrow type

For a function,

1. we care about the types of both arguments and results:

   $$\text{arrow type } \mathtt{T} \to \mathtt{T}$$

   Note the difference between $\mathtt{T} \to \mathtt{T} \to \mathtt{T}$ and $(\mathtt{T} \to \mathtt{T}) \to \mathtt{T}$
2. the type of an abstraction relies on the type of argument, e.g.

   $$\lambda x.x : \mathtt{Bool} \to \mathtt{Bool} \textit{ or } \lambda x.x : \mathtt{Nat} \to \mathtt{Nat}$$

3. Hence, the typing relation on abstractions should be written as

   $$\lambda \mathtt{x} : \mathtt{T_1} . \mathtt{t_2} \;:\; \mathtt{T_1} \to \mathtt{T_2}$$

Arrow type

For a function,

1. we care about the types of both arguments and results:

$$\text{arrow type } \mathtt{T} \to \mathtt{T}$$

   Note the difference between $\mathtt{T} \to \mathtt{T} \to \mathtt{T}$ and $(\mathtt{T} \to \mathtt{T}) \to \mathtt{T}$

2. the type of an abstraction relies on the type of argument, e.g.

$$\lambda x.x : \mathtt{Bool} \to \mathtt{Bool} \textit{ or } \lambda x.x : \mathtt{Nat} \to \mathtt{Nat}$$

3. Hence, the typing relation on abstractions should be written as

$$\lambda \mathtt{x} : \mathtt{T_1} . \mathtt{t_2} \;\; : \;\; \mathtt{T_1} \to \mathtt{T_2}$$

   But how do we derive $\mathtt{T_2}$? We assume $\mathtt{x} : \mathtt{T_1}$!!! So we need an environment (context) for our typing relation:

$$\Gamma \vdash t : \mathtt{T}$$

Pure simply typed λ-calculus ($\lambda_{\rightarrow}$)

Terms $t ::= x \mid \lambda x : \mathtt{T}\,.t \mid t\ t$

Values $v ::= \lambda x : \mathtt{T}\,.t$

Types $\mathtt{T} ::= \mathtt{T} \rightarrow \mathtt{T}$

Contexts $\Gamma ::= \emptyset \mid \Gamma, x : \mathtt{T}$

Typing

$$\textsc{T-Var}\ \frac{x : \mathtt{T} \ \in \Gamma}{\Gamma \vdash x : \mathtt{T}} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : \mathtt{T}_1 \vdash t_2 : \mathtt{T}_2}{\Gamma \vdash \lambda x : \mathtt{T}_1\,.t_2 \ :\ \mathtt{T}_1 \rightarrow \mathtt{T}_2}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash t_1 : \mathtt{T}_1 \rightarrow \mathtt{T}_2 \quad \Gamma \vdash t_2 \ :\ \mathtt{T}_1}{\Gamma \vdash t_1\ t_2 \ :\ \mathtt{T}_2}$$

Quiz: 1. Please draw the type derivation tree of the term
$(\lambda \mathtt{x} : \mathtt{Bool} \rightarrow \mathtt{Nat.x\ true})(\lambda \mathtt{x} : \mathtt{Bool.if\ x\ then\ 0\ else\ (succ\ 0)})$.

Pure simply typed λ-calculus ($\lambda_{\rightarrow}$)

Terms $t ::= x \mid \lambda x : \mathtt{T}\,.t \mid t\ t$

Values $v ::= \lambda x : \mathtt{T}\,.t$

Types $\mathtt{T} ::= \mathtt{T} \rightarrow \mathtt{T}$

Contexts $\Gamma ::= \emptyset \mid \Gamma, x : \mathtt{T}$

Typing

$$\textrm{T-VAR}\ \frac{x : \mathtt{T} \ \in \Gamma}{\Gamma \vdash x : \mathtt{T}} \qquad \textrm{T-ABS}\ \frac{\Gamma, x : \mathtt{T}_1 \vdash t_2 : \mathtt{T}_2}{\Gamma \vdash \lambda x : \mathtt{T}_1\,.t_2 \ : \ \mathtt{T}_1 \rightarrow \mathtt{T}_2}$$

$$\textrm{T-APP}\ \frac{\Gamma \vdash t_1 : \mathtt{T}_1 \rightarrow \mathtt{T}_2 \quad \Gamma \vdash t_2 \ : \ \mathtt{T}_1}{\Gamma \vdash t_1\ t_2 \ : \ \mathtt{T}_2}$$

Quiz: 1. Please draw the type derivation tree of the term $(\lambda\mathtt{x} : \mathtt{Bool} \rightarrow \mathtt{Nat.x\ true})(\lambda\mathtt{x} : \mathtt{Bool.if\ x\ then\ 0\ else\ (succ\ 0)})$.
2. What about this term $\lambda x : \mathit{Bool}.x\ x$?

Properties of typing

Lemma 9.3.1 [Inversion of the Typing Relation]:

1. If $\Gamma \vdash x : \mathtt{R}$, then $x : \mathtt{R} \in \Gamma$.
2. If $\Gamma \vdash \lambda x : \mathtt{T}_1 .t_2 : \mathtt{R}$, then $\mathtt{R} = \mathtt{T}_1 \to \mathtt{R}_2$ for some $\mathtt{R}_2$ with $\Gamma, x : \mathtt{T}_1 \vdash t_2 : \mathtt{R}_2$.
3. If $\Gamma \vdash t_1\, t_2 : \mathtt{R}$, then there is some type $\mathtt{T}_1$ such that $t_1 : \mathtt{T}_1 \to \mathtt{R}$ and $\Gamma \vdash t_2 : \mathtt{T}_1$.
4. for booleans $\cdots$

Properties of typing

Lemma 9.3.1 [Inversion of the Typing Relation]:

1. If $\Gamma \vdash x : \mathtt{R}$, then $x : \mathtt{R} \in \Gamma$.
2. If $\Gamma \vdash \lambda x : \mathtt{T}_1 . t_2 : \mathtt{R}$, then $\mathtt{R} = \mathtt{T}_1 \to \mathtt{R}_2$ for some $\mathtt{R}_2$ with $\Gamma, x : \mathtt{T}_1 \vdash t_2 : \mathtt{R}_2$.
3. If $\Gamma \vdash t_1\ t_2 : \mathtt{R}$, then there is some type $\mathtt{T}_1$ such that $t_1 : \mathtt{T}_1 \to \mathtt{R}$ and $\Gamma \vdash t_2 : \mathtt{T}_1$.
4. for booleans $\cdots$

Theorem 9.3.3 [Uniqueness of types]: In a given typing context Γ, a term t (with free variables all in the domain of Γ) has at most one type.

Progress

Lemma 9.3.4 [Canonical forms]:

- If v is a value of type `Bool`, then v is either `true` or `false`.
- If v is a value of type $\mathtt{T}_1 \to \mathtt{T}_2$, then $v = \lambda x : \mathtt{T}_1 . t2$.

Theorem 9.3.5 [Progress]: Suppose t is a closed, well-typed term (that is, $\Gamma \vdash t : \mathtt{T}$ for some $\mathtt{T}$). Then either t is a value or else there is some t with $t \longrightarrow t$.

Preservation

Theorem 9.3.9 [Preservation]: If $\Gamma \vdash t : \mathtt{T}$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \mathtt{T}$.

Preservation

Theorem 9.3.9 [Preservation]: If $\Gamma \vdash t : \mathtt{T}$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \mathtt{T}$.

Quiz. Please try to prove above theorem and figure out what lemmas we need.

Preservation

Theorem 9.3.9 [Preservation]: If $\Gamma \vdash t : \mathtt{T}$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \mathtt{T}$.
Quiz. Please try to prove above theorem and figure out what lemmas we need.

Lemma 9.3.6 [Permutation]: If $\Gamma \vdash t : \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash t : \mathtt{T}$. Moreover, the latter derivation has the same depth as the former.

Theorem 9.3.7 [Weakening]: If $\Gamma \vdash t : \mathtt{T}$ and $x \notin dom(\Gamma)$, then $\Gamma, x : \mathtt{S} \vdash t : \mathtt{T}$. Moreover, the latter derivation has the same depth as the former.

Theorem 9.3.8 [Preservation of types under substitution]: If $\Gamma, x : \mathtt{S} \vdash t : \mathtt{T}$ and $\Gamma \vdash s : \mathtt{S}$, then $\Gamma \vdash [x \mapsto s]t : \mathtt{T}$.

The Curry-Howard correspondence

Other names for typing rules from a logic view.
The $\rightarrow$ type constructor comes with typing rules of two kinds:

- an introduction rule (T-Abs) describing how elements of the type can be created, and
- an elimination rule (T-App) describing how elements of the type can be used.

Curry-Howard Correspondence, or isomorphism

Logic	Programming languages
proposition	types
proposition $P \supset Q$	type $P \rightarrow Q$
proposition $P \wedge Q$	type $P \times Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)

Erasure and Typability

Type annotations are be used during *type checking*, and will be erased before evaluation.

Definition 9.5.1 [Erasure]: The erasure of a simply typed term t is defined as follows:

$$\begin{array}{rcl} erase(x) & = & x \\ erase(\lambda x : \mathtt{T}_1 . t_2) & = & \lambda x . erase(t_2) \\ erase(t_1\ t_2) & = & erase(t_1) erase(t_2) \end{array}$$

Definition 9.5.3 [Typability]: A term m in the untyped λ-calculus is said to be typable in $\lambda_{\rightarrow}$ if there are some simply typed term t, type $\mathtt{T}$, and context Γ such that $erase(t) = m$ and $\Gamma \vdash t : \mathtt{T}$.

Conclusion

- Typing system $\Gamma \vdash t : T$ can remove some terms before they run into *stuck* states.

Conclusion

- Typing system $\Gamma \vdash t : T$ can remove some terms before they run into *stuck* states.
- However, it also removes well-behaviored terms.

Conclusion

- Typing system $\Gamma \vdash t : T$ can remove some terms before they run into *stuck* states.
- However, it also removes well-behaviored terms.
- Type safety $=$ progress $+$ preservation

Conclusion

- Typing system $\Gamma \vdash t : T$ can remove some terms before they run into *stuck* states.
- However, it also removes well-behaviored terms.
- Type safety $=$ progress $+$ preservation
- For proving safety, some other properties such as *canonical forms, uniqueness of type* are needed.

Conclusion

- Typing system $\Gamma \vdash t : T$ can remove some terms before they run into *stuck* states.
- However, it also removes well-behaviored terms.
- Type safety $=$ progress $+$ preservation
- For proving safety, some other properties such as *canonical forms, uniqueness of type* are needed.
- Simply typed λ-calculus is non-Turing-complete.

Homework

- 8.3.4, 8.3.6, 8.3.7, 9.2.2, 9.2.3, 9.3.2, 9.4.1

Projects. Extend `arith` with

- Untyped lambda calculus (Chapter 7), due on Apr. 14 *(Thursday of Week 8)*
- Simple typed lambda calculus (Chapter 10) , due on May. 12 *(Thursday of Week 12)*
- Subtyping (Chapter 17) , due on Jun. 9 *(Thursday of Week 16)*