Types and Programming Languages

Lecture 1. Untyped arithmetic expressions

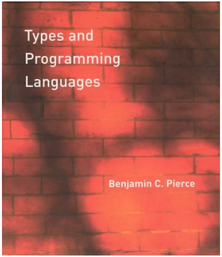
Xiaojuan Cai

cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Spring, 2016

Course overview



Benjamin C. Pierce

We will discuss in this course:

1. theories of types and PLs, including
   a. Operational semantics
   b. Call-by-value λ-calculus
   c. simple type systems and safety
   d. universal and existential polymorphism
   e. type reconstruction
   f. subtyping
   g. recursive types
   h. type operators ...
2. implementation issues, including
   a. the design and analysis of type checking algorithms
   b. implementation an interpreter of a simple functional language with OCaml

Course policy

- Final exam: 60%
- Homework: 20%
- Projects: 20%
- TA:
  Mingzhang Huang, `mingzhanghuang@gmail.com`
- Course homepage:
  `http://basics.sjtu.edu.cn/~xiaojuan/tapl2016`

Outline

Types in Computer Science

Type systems is the most popular and best established lightweight formal methods.

Definition

A *type system* is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

Brief history

Types system (type theory) refers to a much broader field.

- 1900. Formalized, Russell's paradox
- 1925. Simple theory of types, Ramsey
- 1940. Simply typed λ-calculus, Church
- 1973. Constructive type theory, Martin Löf
- 1992. Pure type theory, Barendregt
- ...

Some definitions

- Static type system. Type checking during compile-time
- Dynamic type system. Type checking during run-time
- Static $\Rightarrow$ Conservative $\Rightarrow$ prove the absence of bad behaviours
- Incapable of finding all undesired program behavirous, e.g. divide by zero
- Type checkers
  - automatic: no manual interaction
  - type annotations

What types good for

- Detecting errors early.
- Maintenance tools.
- Abstracting
- Documentation
- Efficiency

Applications: network security, program analysis, theorem prover, database, xml, ...

Language design goes hand-in-hand with type system design.

Outline

Relations

- An *n-place relation* is a set $R \subseteq S_1 \times S_2 \times \cdots \times S_n$.
- A two-place relation R on sets S and T is called a *binary relation*. We often write $s\,R\,t$ instead of $(s, t) \in R$.
- The "mixfix" concrete syntax, e.g, $\Gamma \vdash s : T$ means "the triple (Γ, s, T) in the typing relation".
- P is preserved by R if whenever we have $s\ R\ t$ and $P(s)$, we also have $P(t)$.

Functions

- $dom(R)$: the domain of a relation R on sets S and T is the set of elements $s \in S$ such that $(s, t) \in R$ for some t.
- A relation R on sets S and T is called a partial function if, whenever $(s, t_1) \in R$ and $(s, t_2) \in R$, we have $t_1 = t_2$. If $dom(R) = S$, then R is a total function.
- We write $f(x) \uparrow$ to mean "f is undefined on x," and $f(x) \downarrow$ to mean "f is defined on x."

Ordered sets

A binary relation R on a set S is

- *Reflexive*: $\forall x \in S. x\, R\, x$.
- *Transitive*: $x\, R\, y \;\wedge\; y\, R\, z$ implies $x\, R\, z$.
- *Symmetric*: $x\, R\, y$ implies $y\, R\, x$.
- *Antisymmetric*: $x\, R\, y \;\wedge\; y\, R\, x$ implies $x = y$.

1. *Preorder* (or Quasi order): Reflexive + Transitive
2. *Equivalence*: Preorder + Symmetric
3. *Partial order*: Preorder + Antisymmetric
4. *Total order*: Partial oder + $(\forall x, y \in S. x\, R\, y \vee y\, R\, x)$
5. *Well quasi order*: Preorder + (Any infinite sequence contains an increasing pair)
6. *Well founded order*: Preorder + (No infinite decreasing sequences)

Quiz: 1. Can Transitivity + Symmetry indicate Reflexivity?
2. Give examples to differentiate these orders.

Inductions

- Ordinary induction on natural numbers

  If $P(0)$
  and for all i, $P(i)$ implies $P(i+1)$,
  then $P(n)$ holds for all n.

- Complete induction on natural numbers

  If, for each natural number k,
  given $P(i)$ for all $i < k$
  we can show $P(k)$
  then $P(n)$ holds for all n.

Outline

Untyped systems

- Untyped arithmetic expressions
- Untyped λ-calculus
- ML implementations

Introduction

t ::=	`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

- BNF grammar
- t is metavariable.
- For simplicity, we use arabic numbers, e.g. 3 stands for (`succ` (`succ` (`succ 0`)))
- Currently, `if (succ 0) then true else (pred 0)` is a valid term.

Other ways to give syntax definition

The set of *terms* is the smallest set T such that

- Inductively.
  - $\{\texttt{true}, \texttt{false}, \texttt{0}\} \subseteq T$;
  - if $t_1 \in T$, then $\{\texttt{succ}\ t_1, \texttt{pred}\ t_1, \texttt{iszero}\ t_1\} \subseteq T$;
  - if $t_1, t_2, t_3 \in T$, then $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\ \in T$
- By inference rules

$$\frac{}{\texttt{true} \in T} \qquad \frac{}{\texttt{false} \in T} \qquad \frac{}{\texttt{0} \in T}$$

$$\frac{t_1 \in T}{\texttt{succ}\ t_1\ \in T} \qquad \frac{t_1 \in T}{\texttt{pred}\ t_1\ \in T} \qquad \frac{t_1 \in T}{\texttt{iszero}\ t_1\ \in T}$$

$$\frac{t_1, t_2, t_3 \in T}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\ \in T}$$

Other ways to give syntax definition, cont'd

- Concretely.
  For each natural number i, define S_i as follows:

$$\begin{array}{rcl} S_0 & = & \emptyset \\ S_{i+1} & = & \{\texttt{true}, \texttt{false}, \texttt{0}\} \\ & & \cup \{\texttt{succ}\ t_1, \texttt{pred}\ t_1, \texttt{iszero}\ t_1 \mid t_1 \in S_i\} \\ & & \cup \{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mid t_1, t_2, t_3 \in S_i\} \end{array}$$

$$S = \bigcup_i S_i$$

Lemma. S = T.

Quiz. What if we change the concrete definition of S to

$$\begin{array}{rcl} S_0 & = & \{\texttt{true}, \texttt{false}, \texttt{0}\} \\ S_{i+1} & = & \{\texttt{succ}\ t_1, \texttt{pred}\ t_1, \texttt{iszero}\ t_1 \mid t_1 \in S_i\} \\ & & \cup \{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mid t_1, t_2, t_3 \in S_i\} \end{array}$$

Inductive structure

For any $t \in T$, one of three things must be true about t:

1. t is constant
2. t has form $\texttt{succ}\ t_1, \texttt{pred}\ t_1$, or $\texttt{iszero}\ t_1$
3. t has form $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$.

Two ways to use this observation: inductive definition and inductive proof.

Inductive defintion

$$
\begin{array}{rcl}
\mathit{consts}(\texttt{true}) & = & \{\texttt{true}\} \\
\mathit{consts}(\texttt{false}) & = & \{\texttt{false}\} \\
\mathit{consts}(0) & = & \{0\} \\
\mathit{consts}(\texttt{succ}\ t_1) & = & \mathit{consts}(t_1) \\
\mathit{consts}(\texttt{pred}\ t_1) & = & \mathit{consts}(t_1) \\
\mathit{consts}(\texttt{iszero}\ t_1) & = & \mathit{consts}(t_1) \\
\mathit{consts}(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) & = & \mathit{consts}(t_1) \cup \mathit{consts}(t_2) \cup \mathit{consts}(t_3)
\end{array}
$$

Quiz. 1. Give an inductive definition of *size*, which is the size of the syntax tree of a term t.
2. Give an inductive definition of *depth*, which is the height of the syntax tree of a term t.

$$
\begin{array}{rcl}
\mathit{size}(\mathtt{true}) & = & 1 \\
\mathit{size}(\mathtt{false}) & = & 1 \\
\mathit{size}(0) & = & 1 \\
\mathit{size}(\mathtt{succ}\ t_1) & = & \mathit{size}(t_1) + 1 \\
\mathit{size}(\mathtt{pred}\ t_1) & = & \mathit{size}(t_1) + 1 \\
\mathit{size}(\mathtt{iszero}\ t_1) & = & \mathit{size}(t_1) + 1 \\
\mathit{size}(\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3) & = & \mathit{size}(t_1) + \mathit{size}(t_2) + \mathit{size}(t_3) + 1
\end{array}
$$

Lemma. $|\mathit{consts}(t)| \leq \mathit{size}(t)$.

Proof. By induction on the structure of t.

Principles of induction on terms

- Induction on depth:
  If, for each term s,
  given $P(r)$ for all r such that $depth(r) < depth(s)$,
  we can show $P(s)$,
  then $P(s)$ holds for all s.
- Induction on size:
  If, for each term s,
  given $P(r)$ for all r such that $size(r) < size(s)$,
  we can show $P(s)$,
  then $P(s)$ holds for all s.
- Structural Induction:
  If, for each term s,
  given $P(r)$ for all immediate subterms r of s,
  we can show $P(s)$,
  then $P(s)$ holds for all s.

Outline

Semantics of languages

- Operational semantics. It specifies the behavior of PL by defining an *abstract machine*.
- Denotational semantics. The meaning of a term is taken to be some mathematical object (a number or a function).
- Axiomatic semantics. It takes the laws themselves as the definition of the language.

A toy language – Booleans

Syntax

t	$::=$		*terms*
		`true`	*constant true*
		`false`	*constant false*
		`if` t `then` t `else` t	*conditional*
v	$::=$		*values*
		`true`	*true value*
		`false`	*false value*

Evaluation rules for Booleans

Evaluation

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

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

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

E-IfTrue and E-IfFalse are also called computation rules and E-If is called congruence rule.

Quiz. Evaluate the following terms:

- `true`
- `if true then (if false then false else false) else true`

Derivation tree of One-step evaluation

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

$$
\textsc{E-If}\,\dfrac{\textsc{E-If}\,\dfrac{\textsc{E-IfTrue}\,\dfrac{}{s \longrightarrow \mathit{false}}}{t \longrightarrow u}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}
$$

Theorem 3.5.4 [Determinacy of one-step evaluation]: If $t \longrightarrow t'$ and $t \longrightarrow t''$, then $t' = t''$.

Proof. By induction on the depth of the derivation tree.

Normal form and multi-step evaluation

- A term t is in normal form if no evaluation rule can appliy to it.

Theorem 3.5.7: Every value is in normal form.
Theorem 3.5.8: If t is in normal form, then it is a value.

- The multi-step evaluation relation $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$.

Theorem 3.5.11 [Uniqueness of normal forms]: If $t \longrightarrow^* u$ and $t \longrightarrow^* u'$ where u, u' are normal forms, then $u = u'$.
Theorem 3.5.12 [Termination of evaluation]: For every term t there is some normal form u such that $t \longrightarrow^* u$.

Arithmatic Expression

Syntax

t	$::=$		*terms*
		$\ldots$	
		0	*constant zero*
		$\texttt{succ}\ t$	*successor*
		$\texttt{pred}\ t$	*predecessor*
		$\texttt{iszero}\ t$	*zero test*
v	$::=$		*values*
		$\ldots$	
		$\texttt{nv}$	*numeric value*
nv	$::=$		*numeric values*
		0	*zero value*
		$\texttt{succ}\ \texttt{nv}$	*successor value*

Evaluation rules

Quiz. Give the definition of evaluation rules to guarantee

[Determinacy of one-step evaluation]: *If $t \longrightarrow t'$ and $t \longrightarrow t''$, then $t' = t''$.*

Evaluation

$$\textsc{E-PredZero}\ \frac{}{\texttt{pred}\ 0 \ \longrightarrow\ 0} \qquad \textsc{E-IszeroZero}\ \frac{}{\texttt{iszero}\ 0 \ \longrightarrow\ \texttt{true}}$$

$$\textsc{E-PredSucc}\ \frac{}{\texttt{pred}\ (\texttt{succ}\ \texttt{nv}) \ \longrightarrow\ \texttt{nv}}$$

$$\textsc{E-IszeroSucc}\ \frac{}{\texttt{iszero}\ (\texttt{succ}\ \texttt{nv}) \ \longrightarrow\ \texttt{false}}$$

$$\textsc{E-Pred}\ \frac{t_1 \ \longrightarrow\ t_1'}{\texttt{pred}\ t_1 \ \longrightarrow \texttt{pred}\ t_1'} \qquad \textsc{E-Iszero}\ \frac{t_1 \ \longrightarrow\ t_1'}{\texttt{iszero}\ t_1 \ \longrightarrow \texttt{iszero}\ t_1'}$$

$$\textsc{E-Succ}\ \frac{t_1 \ \longrightarrow\ t_1'}{\texttt{succ}\ t_1 \ \longrightarrow \texttt{succ}\ t_1'}$$

Normal form and stuckness

- What if we change E-PredSucc to `pred (succ t)` $\longrightarrow$ `t`? Does it still satisfy [Determinacy of one-step transition]?
- Note there are meaningless terms, such as `if 0 then (succ true) else (iszeoro false)`.
- A term t is *stuck* if it is in normal form but not a value.

Conclusion

- Types are very important for PLs.
- This course will give a full view of type systems from the simplest one to full-fledged one.
- Fundamental concepts for PLs:
  - syntax, defined inductively, concretely, ...
  - inductive proofs are very important for PLs, especially, *structural induction*.
  - operational semantics plays more and more important roles. We define evaluation rules by using operational transitions.
- Properties such as [**Determinacy of one-step evaluation**], [**Uniqueness of normal forms**], and [**Termination**] are important for a good language design.

Homework

- 3.3.4, 3.5.10, 3.5.13, 3.5.17, 3.5.18.
- Install **OCaml** and get familiar with this language.
  `http://ocaml.org/`