Principles of Programming Languages

Lecture 05 Types

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Peter Thiemann
University of Freiburg, Germany
thiemann@informatik.uni-freiburg.de

16 May 2018

Plan

1. Types
   - Simply-Typed Lambda Calculus
   - Extensions
   - Algebraic Datatypes

Types

- Starting point: simply typed lambda calculus
- basic type constructions: products and sums
- advanced type constructions: trees (aka algebraic datatypes)

Defining a Type System

Type systems are customarily defined using **deduction systems** (i.e., a system of axioms and rules defining a relation).
The relation is phrased as a **typing judgment**, which relates a typing environment, Γ, and an expression, e, to a type, τ.

$$\Gamma \vdash e : \tau$$

A typing environment relates free variables in e to types.

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

Plan

1. Types
   - Simply-Typed Lambda Calculus
   - Extensions
   - Algebraic Datatypes

Syntax of expressions (as before)

$$\text{Expr} \ni e ::= x \mid \lambda x.e \mid e\ e \mid n \mid e{+}e \mid \texttt{if}\ e\ e\ e$$

Syntax of types

$$\text{Type} \ni \tau ::= \alpha \mid \tau \to \tau \mid \texttt{Int}$$

where

- α is a **type variable**, drawn from a denumerable set TVar,
- `Int` is a **type constant** (only applied lambda calculus), and
- $\tau' \to \tau''$ is the type of functions that map values of type τ' to values of type τ''.

Typing Rules

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau' \vdash e : \tau''}{\Gamma \vdash \lambda x.e : \tau' \to \tau''} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}$$

$$\Gamma \vdash n : \texttt{Int} \qquad \frac{\Gamma \vdash e : \texttt{Int} \qquad \Gamma \vdash e' : \texttt{Int}}{\Gamma \vdash e + e' : \texttt{Int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ e_2\ e_3 : \tau}$$

Example: A Type Derivation

For $y : \alpha \vdash (\lambda x.x)\ y : \alpha$

Properties of Simple Typing

1. If $\Gamma \vdash e : \tau$, then $FV(e) \subseteq dom(\Gamma)$.
2. (Weakening) If $\Gamma, x : \tau \vdash e : \tau'$ and $x \notin FV(e)$, then $\Gamma \vdash e : \tau'$.
3. (Substitution) If $\Gamma, x' : \tau' \vdash e : \tau$ and $\Gamma' \vdash e' : \tau'$ and $\Gamma \cup \Gamma'$ is a well-formed typing environment, then $\Gamma \cup \Gamma' \vdash e[x' \mapsto e'] : \tau$.

UNI FREIBURG

In the beginning ...

- types are just syntax
- typing is just some relation between expressions and types

Connecting Types and Semantics

In the beginning ...

- types are just syntax
- typing is just some relation between expressions and types

Connect Types and Semantics

to show that

- types describe execution and
- type avoid misinterpretation of data (type mismatches)

Key Results

1. (Type Preservation)
   If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$.
2. (Progress)
   If $\Gamma \vdash e : \tau$, then exactly one of the following holds.
   1. e is a value.
   2. There is some e' such that $e \longrightarrow e'$.

The Holy Grail

Type Soundness Theorem

If $\emptyset \vdash e : \tau$, then exactly one of the following is true.

- There exists a value v such that $e \xrightarrow{*} v$ and $\emptyset \vdash v : \tau$.
- For each e' such that $e \xrightarrow{*} e'$ there exists e'' such that $e \longrightarrow e''$.

A Surprising Property of the Simply-Typed Lambda Calculus

- Every simply-typed term has a normal form.

Strong Normalization

Suppose that $\Gamma \vdash e : \tau$.
Then there exists a term e' with $e \xrightarrow{*} e'$ such that e' is in normal form.

Plan

Recursion

- Can add a fixpoint operator without breaking the typing properties
- (Strong normalization is lost, though)

Syntax and Reduction

$$e ::= \cdots \mid \texttt{fix} \qquad\qquad \texttt{fix}\ \ e \longrightarrow e\,(\texttt{fix}\ \ e)$$

Typing

$$\Gamma \vdash \texttt{fix} : (\tau \to \tau) \to \tau$$

Example

The factorial function, again.

Product and Record Types

- A **product type** is the type of pairs or n-tuples of values where access to the components is by position. (cf. tuples in Python)
- A **record type** is the type of n-tuples of values where access to the components is by name. (cf. structs in C)
- Each component may have a different type
- The number of components and their names are fixed
- Operations
  - Construction (introduction) of a tuple/record
  - Selection (elimination) of a component

Examples: Python Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.
Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing −

```
tup1 = ();
```

Source: https://www.tutorialspoint.com/python/python_tuples.htm

Python Tuples (Cont)

Accessing Values in Tuples To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print "tup1[0]: ", tup1[0];
print "tup2[1:5]: ", tup2[1:5];
```

Source: https://www.tutorialspoint.com/python/python_tuples.htm

Example: C Structs

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. [...] each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.

```
/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title );
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);
```

Source https://www.tutorialspoint.com/cprogramming/c_structures.htm

Products/Tuples

Syntax

Tuple construction and projection

$$e ::= \cdots \mid (e, \ldots) \mid \pi_i(e) \qquad i \in \mathbf{N}$$
$$\tau ::= \cdots \mid \tau \times \cdots \times \tau$$

Products/Tuples

Syntax

Tuple construction and projection

$$e ::= \cdots \mid (e, \ldots) \mid \pi_i(e) \qquad i \in \mathbf{N}$$
$$\tau ::= \cdots \mid \tau \times \cdots \times \tau$$

Typing rules

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) : \tau_1 \times \cdots \times \tau_n} \qquad \frac{\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n}{\Gamma \vdash \pi_i(e) : \tau_i} \quad i \in \{1, \ldots, n\}$$

Reduction

Call by name

$(e_1, \ldots, e_n)$ is a **value**; no new evaluation contexts required

$$\pi_i(e_1, \ldots, e_i, \ldots) \longrightarrow e_i$$

Call by value

If $v_1, \ldots, v_n$ are values, then $(v_1, \ldots, v_n)$ is a value.

$$\pi_i(v_1, \ldots, v_i, \ldots) \longrightarrow v_i$$

Evaluation contexts

$$E_v ::= \cdots \mid (\overline{v}, E_v, \overline{e})$$

Special Case: $n = 0$ the Unit Type

Syntax

Unit construction, **no elimination, no reduction**

$$e ::= \cdots \mid () \qquad \tau ::= \cdots \mid \texttt{unit}$$

Special Case: $n = 0$ the Unit Type

Syntax

Unit construction, **no elimination, no reduction**

$$e ::= \cdots \mid () \qquad \tau ::= \cdots \mid \texttt{unit}$$

Typing rules

$$\Gamma \vdash () : \texttt{unit}$$

Records

Syntax

Record construction and selection

$$e ::= \cdots \mid \{l_1 = e_1, \ldots\} \mid e.l \qquad l \in \text{Label}$$
$$\tau ::= \cdots \mid \{l_1 : \tau_1, \ldots\}$$

Records

Syntax

Record construction and selection

$$e ::= \cdots \mid \{l_1 = e_1, \ldots\} \mid e.l \qquad l \in \text{Label}$$
$$\tau ::= \cdots \mid \{l_1 : \tau_1, \ldots\}$$

Typing rules

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n) : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}$$

Variant and Sum Types

- A **sum type** or **variant type** is the disjoint, tagged union of two or more types
- Cf. enums in Java and Rust
- Dual of product types

Example: enum in Rust

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

This enum has four variants with different types:

- Quit has no data associated with it at all.
- Move includes an anonymous struct inside it.
- Write includes a single String.
- ChangeColor includes three i32 values.

Source https://doc.rust-lang.org/book/second-edition/ch06-01-defining-an-enum.html

Enum matching in Rust

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Variants

Syntax

Variant construction and match (case)

$$e ::= \cdots \mid l(e) \mid \texttt{match}\ e\ \{l_1(x_1) \Rightarrow e_1, \ldots, l_n(x_n) \Rightarrow e_n\} \qquad l \in \text{Label}$$
$$\tau ::= \cdots \mid l_1(\tau_1) + \cdots + l_n(\tau_n)$$

Variants

Syntax

Variant construction and match (case)

$$e ::= \cdots \mid l(e) \mid \texttt{match}\ e\ \{l_1(x_1) \Rightarrow e_1, \ldots, l_n(x_n) \Rightarrow e_n\} \qquad l \in \text{Label}$$
$$\tau ::= \cdots \mid l_1(\tau_1) + \cdots + l_n(\tau_n)$$

Typing rules

$$\frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash l_i(e) : l_1(\tau_1) + \cdots + l_n(\tau_n)}$$

$$\frac{\Gamma \vdash e : l_1(\tau_1) + \cdots + l_n(\tau_n) \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \ldots \qquad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \texttt{match}\ e\ \{l_1(x_1) \Rightarrow e_1, \ldots, l_n(x_n) \Rightarrow e_n\} : \tau}$$

Variants: Special case $n = 0$ — the Empty Type

Syntax

No construction, empty match (case)

$$e ::= \cdots \mid \texttt{match}\ e\ \{\}$$
$$\tau ::= \cdots \mid 0$$

Variants: Special case $n = 0$ — the Empty Type

Syntax

No construction, empty match (case)

$$e ::= \cdots \mid \texttt{match}\ e\ \{\}$$
$$\tau ::= \cdots \mid 0$$

Typing rules

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \texttt{match}\ e\ \{\} : \tau}$$

Plan

Algebraic Datatypes

- enums in Rust can be recursive

```
pub enum BTree<T> {
    Node(T),
    Branch(Box<BTree<T>>, Box<BTree<T>>)
}
```

- instance of an **algebraic or inductive datatype**
- prototypical example: natural numbers
- operations on inductive types modeled after Church numerals

Natural Numbers as an Algebraic Datatype

UNI FREIBURG

Textbook Definition of $\mathbb{N}$

$\mathbb{N}$ is the smallest set which fulfills

1. $\texttt{zero} \in \mathbb{N}$
2. $n \in \mathbb{N} \Rightarrow \texttt{suc}\ n \in \mathbb{N}$

Natural Numbers as an Algebraic Datatype

UNI FREIBURG

Textbook Definition of $\mathbb{N}$

$\mathbb{N}$ is the smallest set which fulfills

1. $\texttt{zero} \in \mathbb{N}$
2. $n \in \mathbb{N} \Rightarrow \texttt{suc}\ n \in \mathbb{N}$

Rewritten as a Recursive Equation on Sets

$$\mathbb{N} = \{\texttt{zero}\} \cup \{\texttt{suc}\ n \mid n \in \mathbb{N}\}$$

Natural Numbers as an Algebraic Datatype

Textbook Definition of $\mathbb{N}$

$\mathbb{N}$ is the smallest set which fulfills

1. $\texttt{zero} \in \mathbb{N}$
2. $n \in \mathbb{N} \Rightarrow \texttt{suc}\ n \in \mathbb{N}$

Rewritten as a Recursive Equation on Sets

$$\mathbb{N} = \{\texttt{zero}\} \cup \{\texttt{suc}\ n \mid n \in \mathbb{N}\}$$

Rewritten as a Recursive Definition of Types

$$\mathbb{N} \cong \texttt{zero}(\texttt{unit}) + \texttt{suc}(\mathbb{N}) \qquad \text{an isomorphism}$$

Statics of the Natural Number Type

Introduction for $\mathbf{N}$

Call the isomorphism $\texttt{fold}_{nat}$

$$\frac{\Gamma \vdash e : \texttt{zero}(\texttt{unit}) + \texttt{suc}(\mathbf{N})}{\Gamma \vdash \texttt{fold}_{nat}(e) : \mathbf{N}}$$

Statics of the Natural Number Type

Introduction for $\mathbf{N}$

Call the isomorphism $\mathtt{fold}_{nat}$

$$\frac{\Gamma \vdash e : \mathtt{zero}(\mathtt{unit}) + \mathtt{suc}(\mathbf{N})}{\Gamma \vdash \mathtt{fold}_{nat}(e) : \mathbf{N}}$$

Elimination for $\mathbf{N}$

Treat a natural number as a loop / iterator (cf. Church numerals)

$$\frac{\Gamma, x : \mathtt{zero}(\mathtt{unit}) + \mathtt{suc}(\tau) \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \mathbf{N}}{\Gamma \vdash \mathtt{iter}_{nat}[x.e_1](e_2) : \tau}$$

Dynamics of the Natural Number Type

Fold is a Value (Call-by-Name); Iteration requires a value

$$v ::= \cdots \mid \texttt{fold}_{nat}(e)$$
$$E_n ::= \cdots \mid \texttt{iter}_{nat}[x.e_1](E_n)$$

Dynamics of the Natural Number Type

Fold is a Value (Call-by-Name); Iteration requires a value

$$v ::= \cdots \mid \mathtt{fold}_{nat}(e)$$
$$E_n ::= \cdots \mid \mathtt{iter}_{nat}[x.e_1](E_n)$$

Iteration

- Replaces zero by a base value supplied by $e_1[x \mapsto \mathtt{zero}]$
- Replaces suc by a function supplied by $e_1[x \mapsto \mathtt{suc}\ldots]$

$$\mathtt{iter}_{nat}[x.e_1](\mathtt{fold}_{nat}(e)) \longrightarrow \mathtt{match}\ e\ \{\ \mathtt{zero}(z) \Rightarrow e_1[x \mapsto \mathtt{zero}()];\ \mathtt{suc}(z) \Rightarrow e_1[x \mapsto \mathtt{suc}(\mathtt{iter}_{nat}[x.e_1](z))]\ \}$$

Generalization

- This construction can be generalized to any type constructed from sums and products and using a single type variable α to indicate the place of the recursion.
- The special case for $\mathbf{N}$ corresponds to $\texttt{zero}(\texttt{unit}) + \texttt{suc}(\alpha)$
- The tricky part of the semantics of `iter` can be generalized to such types.

Another Example: Binary Trees

Type Template for Binary Tree of Numbers

$$\texttt{leaf}(\texttt{nat}) + \texttt{branch}(\alpha \times \alpha)$$

Another Example: Binary Trees

Type Template for Binary Tree of Numbers

$$\texttt{leaf}(\texttt{nat}) + \texttt{branch}(\alpha \times \alpha)$$

Introduction for **BT**

Call the isomorphism $\texttt{fold}_{bt}$ (again a value)

$$\frac{\Gamma \vdash e : \texttt{leaf}(\texttt{unit}) + \texttt{branch}(\mathbf{BT} \times \mathbf{BT})}{\Gamma \vdash \texttt{fold}_{nat}(e) : \mathbf{BT}}$$

Elimination for **BT**

Treat a binary tree as an iterator (cf. Church encoding)

$$\frac{\Gamma, x : \mathtt{leaf}(\mathtt{unit}) + \mathtt{branch}(\tau \times \tau) \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \mathbf{BT}}{\Gamma \vdash \mathtt{iter}_{nat}[x.e_1](e_2) : \mathbf{BT}}$$

Dynamics for Binary Tree Elimination

Iteration

- Replaces `leaf` by a base value supplied by $e_1[x \mapsto \texttt{leaf}]$
- Replaces `branch` by a function supplied by $e_1[x \mapsto \texttt{branch}\ldots]$
- The template has two occurrences of α hence iteration splits into **two** recursive calls.

$$\texttt{iter}_{bt}[x.e_1](\texttt{fold}_{bt}(e)) \longrightarrow$$

$$\begin{array}{l}\texttt{match}\ e\ \{\ \texttt{leaf}(z) \Rightarrow e_1[x \mapsto \texttt{leaf}()];\\ \qquad\quad \texttt{branch}(z_1, z_2) \Rightarrow e_1[x \mapsto \texttt{branch}(\ \texttt{iter}_{bt}[x.e_1](z_1),\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{iter}_{bt}[x.e_1](z_2)\ )]\ \}\end{array}$$