IA014: Advanced Functional Programming

3. Simply Typed Lambda Calculus

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Motivation

Untyped λ-calculus:

- does not distinguish between functions and arguments
  everything is a function!
- the evaluation may get *stuck*
  - $(x\ y) \not\to$ *not a value*
  - $(\lambda x.\,\mathsf{succ}\,x)\,\mathsf{true} \to \mathsf{succ}\,\mathsf{true} \not\to$
  - $42 + \lambda x.x \not\to$

Solution? Types!

In this lecture we will:

- define the *simply typed λ-calculus* $\lambda^{\to}$
- be working with λ-calculus extended with Booleans
- use $\mathrm{t}, \mathrm{t}' \ldots$ for terms (instead of $M, M', N, \ldots$)

Type systems

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

B. Pierce, Types and Programming Languages

Desirable properties

1. *type safe*
   well-typed programs “do not go wrong”
2. *not too conservative*
   most useful programs should be typeable

Revision: Call-By-Value λ-calc.

syntax

$$
\begin{array}{rll}
\mathrm{t} ::= & x & \text{variable} \\
\mid & \mathrm{t}\ \mathrm{t}' & \text{application} \\
\mid & \lambda x.\mathrm{t} & \text{abstraction} \\
\mathrm{v} ::= & \lambda x.\mathrm{t} & \text{abstraction } \textit{value}
\end{array}
$$

semantics

$$\frac{}{(\lambda x.\mathrm{t})\ \mathrm{v} \to \mathrm{t}[x := \mathrm{v}]}$$

$$\frac{\mathrm{t}_1 \to \mathrm{t}_1'}{\mathrm{t}_1\ \mathrm{t}_2 \to \mathrm{t}_1'\ \mathrm{t}_2} \qquad \frac{\mathrm{t} \to \mathrm{t}'}{\mathrm{v}\ \mathrm{t} \to \mathrm{v}\ \mathrm{t}'}$$

Adding types – syntax of $\lambda^{\to}$

Terms and Values

$\mathrm{t} ::=$	x	variable
$\mid$	$\mathrm{t}\ \mathrm{t}'$	application
$\mid$	$\lambda x{:}\mathrm{T}.\mathrm{t}$	abstraction
$\mid$	true	constant true
$\mid$	false	constant false
$\mid$	$\mathrm{if\ t\ then\ t'\ else\ t''}$	conditional
$\mathrm{v} ::=$	$\lambda x{:}\mathrm{T}.\mathrm{t}$	abstraction *value*
$\mid$	true	true value
$\mid$	false	false value

Types

$\mathrm{T} ::=$	$\mathrm{T} \to \mathrm{T}$	function *type*
$\mid$	Bool	Boolean type

Basic type terminology

$$\begin{array}{rcll} \mathrm{T} & ::= & \mathrm{T} \to \mathrm{T} & \text{function type} \\ & | & \mathrm{Bool} & \text{Boolean type} \end{array}$$

The grammar above defines the *set of simple types over* Bool:

- Bool is a *base type* (also *atomic type*)
- $\mathrm{T}_1 \to \mathrm{T}_2$ is a *function type*: type of functions expecting arguments of type T_1 and returning results of T_2
- "$\to$" is a *type constructor*
  (creates a new type based on T_1 and T_2)

Notation

- $\mathrm{t} : \mathrm{T}$ means t *is of type* T

Typing relation, context

Consider a term $\mathrm{t} := \lambda x.\mathrm{t}'$.

- the type of the body t' depends on the type of the argument x
- type of t depends on the type of both t' and x
- when typing t we need to "know" the type of x (a context)

Typing context Γ

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

Term t has type T in the typing context Γ.

- *binding* $x : \mathrm{T}$ is a pair variable+type
- Γ is a *sequence* of bindings
- we can assume that all bindings in Γ are distinct
- for empty context, we write just $\vdash \mathrm{t} : \mathrm{T}$

Typing rules

$$\Gamma \vdash \mathrm{true} : \mathrm{Bool} \quad \text{(T-True)}$$

$$\Gamma \vdash \mathrm{false} : \mathrm{Bool} \quad \text{(T-False)}$$

$$\frac{x : \mathrm{T} \in \Gamma}{\Gamma \vdash x : \mathrm{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : \mathrm{T}_1 \vdash \mathrm{t} : \mathrm{T}_2}{\Gamma \vdash \lambda x.\mathrm{t} : \mathrm{T}_1 \to \mathrm{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{T}_1 \to \mathrm{T}_2 \qquad \Gamma \vdash \mathrm{t}_2 : \mathrm{T}_1}{\Gamma \vdash \mathrm{t}_1\ \mathrm{t}_2 : \mathrm{T}_2} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{Bool} \qquad \Gamma \vdash \mathrm{t}_2 : \mathrm{T} \qquad \Gamma \vdash \mathrm{t}_3 : \mathrm{T}}{\Gamma \vdash \mathrm{if}\ \mathrm{t}_1\ \mathrm{then}\ \mathrm{t}_2\ \mathrm{else}\ \mathrm{t}_3 : \mathrm{T}} \quad \text{(T-If)}$$

Typing derivation trees

What is the type of $\lambda x : \mathrm{Bool} \to \mathrm{Bool}.\lambda y : \mathrm{Bool}.x\ y$?

$$\dfrac{\dfrac{\dfrac{x : \mathrm{Bool} \to \mathrm{Bool} \in \{x : \mathrm{Bool} \to \mathrm{Bool}, y : \mathrm{Bool}\}}{x : \mathrm{Bool} \to \mathrm{Bool}, y : \mathrm{Bool} \vdash x : \mathrm{Bool} \to \mathrm{Bool}} \quad \dfrac{y : \mathrm{Bool} \in \{x : \mathrm{Bool} \to \mathrm{Bool}, y : \mathrm{Bool}\}}{x : \mathrm{Bool} \to \mathrm{Bool}, y : \mathrm{Bool} \vdash y : \mathrm{Bool}}}{x : \mathrm{Bool} \to \mathrm{Bool}, y : \mathrm{Bool} \vdash x\ y : \mathrm{Bool}}}{x : \mathrm{Bool} \to \mathrm{Bool} \vdash (\lambda y : \mathrm{Bool}.x\ y) : \mathrm{Bool} \to \mathrm{Bool}}}{\vdash (\lambda x : \mathrm{Bool} \to \mathrm{Bool}.\lambda y : \mathrm{Bool}.x\ y) : (\mathrm{Bool} \to \mathrm{Bool}) \to \mathrm{Bool} \to \mathrm{Bool}}$$

Term t is *well typed* if there exists a type T s.t. $\vdash \mathrm{t} : \mathrm{T}$

Evaluation rules for $\lambda^{\to}$

$$\frac{}{(\lambda x : \mathrm{T}.\mathrm{t})\mathrm{v} \to \mathrm{t}[x := \mathrm{v}]} \text{ (E-Abs)}$$

$$\frac{\mathrm{t}_1 \to \mathrm{t}_1'}{\mathrm{t}_1\ \mathrm{t}_2 \to \mathrm{t}_1'\ \mathrm{t}_2} \text{ (E-App1)}$$

$$\frac{\mathrm{t} \to \mathrm{t}'}{\mathrm{v}\ \mathrm{t} \to \mathrm{v}\ \mathrm{t}'} \text{ (E-App2)}$$

Types have no effect on the evaluation!

Type safety (soundness)

Type safety: Well-typed terms do not "go wrong".
(E.g. do not get stuck.)

Safety = progress + preservation

Progress A well-typed term is not stuck.
(It's either a value, or it can take one more 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.

Progress for $\lambda^{\to}$

Lemma (Canonical forms)

1. *if* v *is a value of type* Bool*, then* v *is either* true *or* false
2. *if* v *is a value of type* $\mathrm{T}_1 \to \mathrm{T}_2$*, then* $\mathrm{v} = \lambda x : \mathrm{T}_1.\mathrm{t}$

Theorem (Progress)

Let t *be a closed, well-typed term (i.e.* $\exists \mathrm{T}$ *s.t.* $\vdash \mathrm{t} : \mathrm{T}$*). Then either* t *is a value, or there exists* t' *such that* $\mathrm{t} \to \mathrm{t}'$*.*

Preservation for $\lambda^{\to}$

Lemma (Preservation of types under substitution)

If $\Gamma, x : \mathrm{S} \vdash \mathrm{t} : \mathrm{T}$ *and* $\Gamma \vdash \mathrm{s} : \mathrm{S}$*, then* $\Gamma \vdash \mathrm{t}[x := \mathrm{s}] : \mathrm{T}$

Theorem (Preservation)

If $\Gamma \vdash \mathrm{t} : \mathrm{T}$ *and* $\mathrm{t} \to \mathrm{t}'$*, then* $\Gamma \vdash \mathrm{t}' : \mathrm{T}$

Normalization for $\lambda^{\to}$

Term is *normalizable* iff its evaluation is guaranteed to halt in a finite number of steps.

Theorem ((Weak) Normalization for $\lambda^{\to}$)

If $\vdash \mathrm{t} : \mathrm{T}$, *then* t *is normalizable.*

- the theorem above is stated for the CBV semantics
- holds also for full β-reduction (*strong normalization*)

Consequences

- $\boldsymbol{\omega} = \lambda x.x\ x$ is neither strongly nor weakly normalizing
- . . . $\boldsymbol{\omega}$ is not typeable (as any term with an infinite computation), therefore
- . . . $\lambda^{\to}$is not Turing complete!

From λ-calculus to functional programming II

Recursion

Revision (untyped lambda calculus)

$$G := \lambda f.\lambda n.\, \text{if } n = 0 \text{ then } 1 \text{ else } n * (f\ (n-1))$$

$$\text{fact} := \mathbf{FIX}\ G = G\ (\mathbf{FIX}\ G)$$

Recursion in $\lambda^{\to}$

$$\begin{aligned} H :=& \lambda f : \mathrm{Nat} \to \mathrm{Bool}.\lambda x : \mathrm{Nat}. \\ & \text{if iszero } x \text{ then true} \\ & \text{else if iszero } (\mathrm{pred}\, x) \text{ then false} \\ & \text{else } f\, \text{iszero } (\mathrm{pred}(\mathrm{pred}\, x)) \end{aligned}$$

$H : (\mathrm{Nat} \to \mathrm{Bool}) \to \mathrm{Nat} \to \mathrm{Bool}$

$\text{iseven} := \mathbf{FIX}\, H \qquad \text{iseven} : \mathrm{Nat} \to \mathrm{Bool}$

Typing **FIX**

some combinators for **FIX**

- $\boldsymbol{Y} := \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$
- $\boldsymbol{\Theta} := (\lambda x f.f(x\ x\ f))(\lambda x f.f(x\ x\ f))$

What is the type of $\boldsymbol{Y}$?

- problem: self-application: $\lambda x : \mathrm{T}.x\ x$
- the type T would need to be
  - a function type
  - an argument type (of the same type!)
- **FIX** *is not typeable in* $\lambda^{\to}$
- also follows from normalization for $\lambda^{\to}$

Recursion in $\lambda^{\to}$

problem: $\mathbf{FIX}$ is not typeable in $\lambda^{\to}$
solution: extend $\lambda^{\to}$ with a new primitive fix

syntax

$\mathrm{t} ::=$	$\ldots$	*terms*
$\mid$	$\mathrm{fix}\ \mathrm{t}$	fixed point of t

typing

$$\frac{\Gamma \vdash \mathrm{t} : \mathrm{T} \to \mathrm{T}}{\Gamma \vdash \mathrm{fix}\,\mathrm{t} : \mathrm{T}} \text{ (T-Fix)}$$

evaluation

$$\mathrm{fix}(\lambda x : \mathrm{T}_1.\mathrm{t}_2) \to \mathrm{t}_2[x := (\mathrm{fix}(\lambda x : \mathrm{T}_1.\mathrm{t}_2))] \text{ (E-FixBeta)}$$

$$\frac{\mathrm{t} \to \mathrm{t}'}{\mathrm{fix}\,\mathrm{t} \to \mathrm{fix}\,\mathrm{t}'} \text{ (E-Fix)}$$

Note: adding fix breaks strong normalization!

Unit type

syntax

$t ::=$	$\ldots$	*terms*
$\mid$	unit	constant unit
$v ::=$	$\ldots$	*values*
$\mid$	unit	constant unit
$T ::=$	$\ldots$	*types*
$\mid$	Unit	unit type

typing rule

$$\Gamma \vdash \mathrm{unit} : \mathrm{Unit} \text{ (T-Unit)}$$

- "the most simple" base type
- similar to `void` in C or Java
- uses: languages with side effects
  - if we do not care about the returned value
  - value is secondary to the side-effect

Sequencing

syntax

$$\begin{array}{rcll} t & ::= & \ldots & \textit{terms} \\ & | & \mathrm{t}_1; \mathrm{t}_2 & \text{sequence} \end{array}$$

typing rule

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{Unit} \qquad \Gamma \vdash \mathrm{t}_2 : \mathrm{T}_2}{\Gamma \vdash \mathrm{t}_1; \mathrm{t}_2 : \mathrm{T}_2} \text{ (T-Seq)}$$

evaluation rules

$$\frac{\mathrm{t}_1 \to \mathrm{t}_1'}{\mathrm{t}_1; \mathrm{t}_2 \to \mathrm{t}_1'; \mathrm{t}_2} \text{ (E-Seq1)} \qquad \mathrm{unit}; \mathrm{t}_2 \to \mathrm{t}_2 \text{ (E-Seq2)}$$

different approach

- $\mathrm{t}_1; \mathrm{t}_2$ is an *abbreviation* for $(\lambda x : \mathrm{Unit}.\ \mathrm{t}_2)\ \mathrm{t}_1$
- sequencing is therefore a *derived form:* typing and evaluation rules can be derived
- derived forms are *syntactic sugar*

Let bindings

Is `let` just a derived form?

$$\text{let } x = \text{t}_1 \text{ in } \text{t}_2 \quad := \quad (\lambda x : \text{T}_1.\text{t}_2)\ \text{t}_1$$

problem: where to get T_1 from? (when desugaring)
answer: the typechecker

$$\frac{\vdots \qquad \vdots}{\Gamma \vdash \text{t}_1 : \text{T}_1 \qquad \Gamma, x : \text{T}_1 \vdash \text{t}_2 : \text{T}_2}\ \text{(T-Let)}$$
$$\Gamma \vdash \text{let } x = \text{t}_1 \text{ in } \text{t}_2 : \text{T}_2$$

converts to

$$\frac{\dfrac{\vdots}{\Gamma \vdash \text{t}_1 : \text{T}_1} \qquad \dfrac{\dfrac{\vdots}{\Gamma, x : \text{T}_1 \vdash \text{t}_2 : \text{T}_2}}{\Gamma \vdash \lambda x : \text{T}_1.\text{t}_2 : \text{T}_1 \to \text{T}_2}\ \text{(T-Abs)}}{\Gamma \vdash (\lambda x : \text{T}_1.\text{t}_2)\ \text{t}_1 : \text{T}_2}\ \text{(T-App)}$$

Let bindings II

- `let` is not a derived form in $\lambda^{\to}$!
- we therefore have to add it externally

syntax

$$\begin{array}{rll} \mathrm{t} ::= & \ldots & \textit{terms} \\ \mid & \mathrm{let}\ \ x = \mathrm{t}\ \ \mathrm{in}\ \ \mathrm{t} & \text{let binding} \end{array}$$

typing rule

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{T}_1 \qquad \Gamma, x : \mathrm{T}_1 \vdash \mathrm{t}_2 : \mathrm{T}_2}{\Gamma \vdash \mathrm{let}\ \ x = \mathrm{t}_1\ \ \mathrm{in}\ \ \mathrm{t}_2 : \mathrm{T}_2}\ \text{(T-Let)}$$

evaluation rules

$$\mathrm{let}\ \ x = \mathrm{v}\ \ \mathrm{in}\ \ \mathrm{t} \quad \to \quad \mathrm{t}[x := \mathrm{v}]\ \text{(E-LetV)}$$

$$\frac{\mathrm{t}_1 \to \mathrm{t}_1'}{\mathrm{let}\ \ x = \mathrm{t}_1\ \ \mathrm{in}\ \ \mathrm{t}_2 \quad \to \quad \mathrm{let}\ \ x = \mathrm{t}_1'\ \ \mathrm{in}\ \ \mathrm{t}_2}\ \text{(E-Let)}$$

Recursive let bindings

Example

```
let rec iseven : Nat → Bool =
  λx : Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
  iseven 7;
```

Note: `let` in Haskell/ML

	non-recursive	recursive
ML	`let`	`let rec`
HASKELL	–	`let`

`let rec` is a derived form:

$$\text{let rec } x : T_1 = t_1 \text{ in } t_2 \quad := \quad \text{let } x = \text{fix}(\lambda x : T_1.t_1) \text{ in } t_2$$

Pairs

syntax

$t ::=$	$\ldots$	*terms*
$\mid$	(t, t)	pair
$\mid$	$t.1$	first projection
$\mid$	$t.2$	second projection
$v ::=$	$\ldots$	*values*
$\mid$	(t, t)	pair value
$T ::=$	$\ldots$	*types*
$\mid$	$T_1 \times T_2$	product type

typing rules

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \quad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.1 : T_1} \quad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.2 : T_2} \quad \text{(T-Proj2)}$$

Note: $\times$ is a new *type constructor*

Pairs – evaluation

evaluation rules

$$(v_1, v_2).1 \to v_1 \quad \text{(E-PairV1)}$$

$$(v_1, v_2).2 \to v_2 \quad \text{(E-PairV2)}$$

$$\frac{t \to t'}{t.1 \to t'.1} \quad \text{(E-Proj1)}$$

$$\frac{t \to t'}{t.2 \to t'.2} \quad \text{(E-Proj2)}$$

$$\frac{t_1 \to t_1'}{(t_1, t_2) \to (t_1', t_2)} \quad \text{(E-Pair1)}$$

$$\frac{t_2 \to t_2'}{(v_1, t_2) \to (v_1, t_2')} \quad \text{(E-Pair2)}$$

Note: Pairs can be naturally extended to tuples and records.

Lists

syntax

$t ::=$	$\ldots$	*terms*
	$\mid \; \mathrm{nil}[T]$	empty list
	$\mid \; \mathrm{cons}[T]\ t\ t$	list constructor
	$\mid \; \mathrm{null}[T]\ t$	test for empty list
	$\mid \; \mathrm{hd}[T]\ t$	head of a list
	$\mid \; \mathrm{tl}[T]\ t$	tail of a list
$v ::=$	$\ldots$	*values*
	$\mid \; \mathrm{nil}[T]$	empty list
	$\mid \; \mathrm{cons}[T]\ v\ v$	list constructor
$T ::=$	$\ldots$	*types*
	$\mid \; \mathrm{List}\ T$	type of lists

some typing rules

$$\Gamma \vdash \mathrm{nil}[T] : \mathrm{List}\ T \quad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : \mathrm{List}\ T}{\Gamma \vdash \mathrm{cons}[T]\ t_1\ t_2 : \mathrm{List}\ T} \quad \text{(T-Cons)}$$

Try the rest yourselves!

Type Ascription

syntax

$$\begin{array}{rcll} \mathrm{t} & ::= & \ldots & \textit{terms} \\ & | & \mathrm{t}\ \mathrm{as}\ \mathrm{T} & \text{ascription} \end{array}$$

typing rule

$$\frac{\Gamma \vdash \mathrm{t} : \mathrm{T}}{\Gamma \vdash \mathrm{t}\ \mathrm{as}\ \mathrm{T} : \mathrm{T}}\ \text{(T-Asc)}$$

evaluation rules

$$\frac{\mathrm{t} \to \mathrm{t}'}{\mathrm{t}\ \mathrm{as}\ \mathrm{T} \to \mathrm{t}'\ \mathrm{as}\ \mathrm{T}}\ \text{(E-Asc1)} \qquad \mathrm{v}\ \mathrm{as}\ \mathrm{T} \to \mathrm{v}\ \text{(E-Asc2)}$$

- $\mathrm{t}\ \mathrm{as}\ \mathrm{T}$ is an assertion that t has type T
- usefull typechecking and documentation purposes