T-501-FMAL Programming Languages
Lectures 14-15

Tarmo Uustalu
Reykjavik University
tarmo@ru.is

Spring 2021

Basis of FP: Lambda-calculus

Lambda-calculus

- FP is based on *lambda-calculus*, a formalism for computing with functions by Alonzo Church and Haskell Curry.
- It comes in two basic versions:
  - *untyped lambda-calculus*
  - *typed lambda-calculus*.
- In untyped lambda-calculus, one can write *terms* (expressions). The terms can be simplified or compute according to certain rules.
- In typed lambda-calculus, there are also *types*. Good terms can be assigned types.

Lambda-calculus ctd

- In untyped lambda-calculus, it is possible to write terms that are meaningless (at least if one attempts at a naive interpretation), but can nevertheless be useful.

  Eg they make it possible to express recursively defined functions. These can fail to terminate.
  Untyped lambda-calculus is Turing-complete.

- Typed lambda-calculus is more disciplined.

  In particular, computations in typed lambda-calculus always terminate (so it cannot be Turing-complete).

- Actual FP languages use variants and extensions of untyped and typed lambda-calculus.

Anonymous functions

- A central idea of lambda-calculus:

  To work with a function, ie to define it, to apply it, to pass it as an argument, you don't have to give it a name.
- A function can be *anonymous* (nameless).

- Instead of defining $f\,x = 17 * x$ and then working with f, one can directly work with $\lambda x.\,17 * x$.
- This is an expression for the same function.

Untyped lambda-calculus

Untyped lambda calculus: Terms

- There is exactly one syntactic category: *terms*.
- Terms are built from *variables*.
- There are 3 term forms:
  - *variables-as-terms*: x,
  - *lambda-abstraction*: $\lambda x.\, t$ (for defining functions)
  - *application*: $t\ u$ (for applying functions to arguments, or calling them with actual parameters)
- Application associates to the left, ie $t\ u\ v$ means $(t\ u)\ v$.
- Lambda extends as far to the right as possible (until the closest closing parenthesis), ie $\lambda x.\, t\ u$ means $\lambda x.\,(t\ u)$.
- One can write "nonsense" terms like $x\, x$ (self-application of a function). (Typed lambda-calculus forbids such terms.)

- In a real FP language, there would also be some *constants*.

What about let?

- Let is definable:

$$\text{let } x = t \text{ in } u \quad = \quad (\lambda x.\, u)\, t$$

$$\text{let } f\ y = t \text{ in } u \quad = \quad \text{let } f = \lambda y.\, t \text{ in } u \quad = \quad (\lambda f.\, u)\, (\lambda y.\, t)$$

Bound variables, α-conversion

- The lambda-abstraction $\lambda x.\, t$ *binds* the occurrences of x in t.
- If a variable occurrence in a term is not bound by any lambda, it is *free*.

- Terms differing only by the names of bound variables (such terms are called α-*convertible*) are considered equal.

- A bound variable x of $\lambda x.\, t$ cannot be replaced by a variable that is free in t. This is *capture* and must be avoided.
- Eg $\lambda x.\, x\, y = \lambda z.\, z\, y$, but $\lambda x.\, x\, y \neq \lambda y.\, y\, y$.

Examples of α-convertibility

$$\lambda x.\, y\ x\, z = \lambda x'.\, y\, x'\, z$$

$$\lambda x.\, x\, x = \lambda x'.\, x'\, x'$$

$$\lambda x.\, x\, (\lambda z.\, x\, z) = \lambda x'.\, x'\, (\lambda z.\, x'\, z)$$

$$\lambda x.\, y = \lambda x'.\, y$$

Normalization

- Computation amounts to simplifying a term in small steps, called β-*reduction* steps, until there is no opportunity left.
- Terms that cannot be β-reduced are called β-*normal*.
- The process of reducing a term in multiple steps to a β-normal form is called *normalization*.
- Eg

$$(\lambda x.\, \lambda y.\; y\, x)\, v\, (w\, v) \to (\lambda y.\, y\, v)\, (w\, v) \to w\, v\, v$$

- Several β-reduction steps may be applicable to one and the same term, one then has to choose which step to apply.

- In a real FP language, there would also be reduction rules for the constants.
- One does not necessarily normalize terms fully, eg one may not reduce under λ.
- The reduction step to take is deterministically chosen by an evaluation strategy.

β-reduction

- The *(single-step) β-reduction* relation $\rightarrow$ is defined by these rules:

$$\frac{}{(\lambda x.\, t)\, u \rightarrow t[u/x]}$$

$$\frac{t \rightarrow t'}{\lambda x.\, t \rightarrow \lambda x.\, t'} \quad \frac{t \rightarrow t'}{t\, u \rightarrow t'\, u} \quad \frac{u \rightarrow u'}{t\, u \rightarrow t\, u'}$$

- $t[u/x]$ is a notation for substituting u for all occurrences of x in t. This requires some care...
- Terms of the form $(\lambda x.\, t)\, u$ are called *β-redexes*. A β-reduction step simplifies one of the possible multiple β-redexes in a term.
- A *β-normal form* is a term without β-redexes.
- *Multi-step β-reduction* $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$, ie zero, one or multiple single-step β-reductions.

Substitution

- Substitution $t[v/x]$ is defined by these equations by structural recursion on t:

$$\begin{array}{rcll} x[v/x] & = & v & \\ y[v/x] & = & y & \text{if } y \neq x \\ (\lambda x.\, t)[v/x] & = & \lambda x.\, t & \\ (\lambda y.\, t)[v/x] & = & \lambda y.\, (t[v/x]) & \text{if } y \text{ not free in } v \text{ and } y \neq x \\ (t\, u)[v/x] & = & (t[v/x])\, (u[v/x]) & \end{array}$$

- Note the side-condition of the 4th equation. It is again to avoid *capture*. A free variable should not become bound when a substitution is made.
- If the 4th equation does not apply outright, the bound variable y of $\lambda y.\, t$ must be renamed to a variable not free in v and different from x.

Examples of substitution

$$(\lambda x.\, y\, x)[z\, w/y] = \lambda x.\, z\, w\, x$$

$$(y\, x\, (\lambda z.\, y))[\lambda w.\, w\, y/x] = y\, (\lambda w.\, w\, y)\, (\lambda z.\, y)$$

$$(\lambda x.\, x\, y)[\lambda x.\, x\, y/y] = \lambda x.\, x\, (\lambda x.\, x\, y)$$

$$(\lambda x.\, x\, y)[z/y] = \lambda x.\, x\, z$$

$$(\lambda x.\, x\, y)[z/x] = \lambda x.\, x\, y$$

No substitution for bound variables!

$$(\lambda x.\, x\, y)[x/y] \neq \lambda x.\, x\, x$$

$$(\lambda x.\, x\, y)[x/y] = (\lambda z.\, z\, y)[x/y] = \lambda z.\, z\, x$$

Capture is not allowed.

Need to rename bound variable.

Confluence

- Untyped lambda-calculus is *confluent* in the sense that: If $t \to^* u$ and $t \to^* v$, then there exists w such that $u \to^* w$ and $v \to^* w$.
- (Intuitively: You cannot reduce in an irrepairable direction.)

Uniqueness of normal forms

- An immediate consequence of confluence is *uniqueness of normal forms*:
  Given a term t, there can be at most one u in β-normal form such that $t \to^* u$.
- (But notice that there need not be a normal form for t; we are only saying that there cannot be two different normal forms.)

- In a real FP language with effects, however, different evaluation strategies can give different effects and also a different normal form for the same term.

Failure of normalization in general

- Not every term has a normal form.
- Eg $\Omega = (\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$ is without a normal form.
- Indeed, the only possible reduction step is $(\lambda x.\, x\, x)\,(\lambda x.\, x\, x) \to (\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$, ie one is back at Ω in one step.

Evaluation strategies

- If a term has a normal form, some evaluating strategies may not lead to it, by failing to terminate.
- But, for any term with a normal form, the *normal* (ie leftmost outermost) evaluation strategy finds it.
- The *applicative* (leftmost innermost) evaluation strategy may generally fail to terminate.

- Consider eg $(\lambda x.\, y)\,\Omega$.
- With the normal evaluation strategy, the first step is $(\lambda x.\, y)\,\Omega \to y$ and y is a normal form.
- With the applicative evaluation strategy, the first step is $(\lambda x.\, y)\,\Omega \to (\lambda x.\, y)\,\Omega$ and this is repeated forever.

Typed lambda-calculus

(Simply) typed lambda-calculus: Types

- In addition to terms, (simply) typed lambda-calculus has *types*.
- A term can be *typed* with a type, depending on a context of types for its free variables.
- In simply typed lambda-calculus, there are just two forms of types:
    - *type variables*: X,
    - *function types*: $A \to B$
- We can eg type $\lambda x.\, x$ with $A \to A$ for any type A or $\lambda x.\, \lambda y. x$ with $A \to B \to A$ for any types A, B.
- $\to$ associates to the right; we will write $A \to B \to C$ instead of $A \to (B \to C)$.

- In a real FP language, there would additionally be some *base types* and usually also some type constructors beyond $\to$.

Type assignment

- *Assignment of types to terms* wrt a typing context is defined by these rules:

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}$$

- These rules derive *typing judgements* of the form $\Gamma \vdash t : A$, stating that a term t is of a type A in a context Γ.
- The *typing context* Γ is a list $x_1 : A_1, \ldots, x_n : A_n$ of typings for variables.
  The list $x_1, \ldots, x_n$ must contain all free variables of the term t (and may contain further variables unused in t).
- (The symbol $\vdash$ is called "turnstile".)

Type derivations

- A *type derivation* for a term is built by applying the typing rules, like eg this:

$$\dfrac{\overline{x : A \vdash x : A}}{\vdash \lambda x.\, x : A \to A}$$

- Or this:

$$\dfrac{\dfrac{\dfrac{\overline{x : A, f : A \to B \vdash f : A \to B} \quad \overline{x : A, f : A \to B \vdash x : A}}{x : A, f : A \to B \vdash f\, x : B}}{x : A \vdash \lambda f.\, f\, x : (A \to B) \to B}}{\vdash \lambda x.\, \lambda f.\, f\, x : A \to ((A \to B) \to B)}$$

- “Nonsense” terms like $\Omega = (\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$ are not typable.

More examples of type assignment

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash x : A \to B \to C} \quad \overline{\Gamma \vdash z : A}}{\Gamma \vdash x\,z : B \to C} \qquad \dfrac{\overline{\Gamma \vdash y : A \to B} \quad \overline{\Gamma \vdash z : A}}{\Gamma \vdash y\,z : B}}{x : A \to B \to C, y : A \to B, z : A \vdash x\,z\,(y\,z) : C}}{x : A \to B \to C, y : A \to B \vdash \lambda z.\, x\,z\,(y\,z) : C}}{x : A \to B \to C \vdash \lambda y.\, \lambda z.\, x\,z\,(y\,z) : (A \to B) \to A \to C}}{\vdash \lambda x.\, \lambda y.\, \lambda z.\, x\,z\,(y\,z) : (A \to B \to C) \to (A \to B) \to A \to C}$$

where $\Gamma = x : A \to B \to C, y : A \to B, z : A$.

Subject reduction (type preservation)

- Typed lambda calculus has the *subject reduction* or *type preservation* property: β-reduction preserves the type of a term.
- Ie, if $\Gamma \vdash t : A$ and $t \to u$, then also $\Gamma \vdash u : A$.

- This property is required also in any reasonable real typed FP language.

Strong normalization

- Differently from untyped lambda-calculus, in typed lambda-calculus every term has a normal form.
- Moreover, it is *strongly normalizing*, which means that any evaluation strategy will compute that normal form.

- But again, in a real typed FP language, some terms may reduce infinitely (because such languages often include a general recursor as a primitive).
- Also, in the presence of effects, different evaluation strategies give different effects and also a different normal form for the same term.

Polymorphically typed lambda-calculus

- Polymorphically typed lambda calculus
  (a.k.a. System F)
  adds a new type form:
  - *universally quantified types*: $\forall X.\, A$
- It also adds two new non-syntax-directed typing rules of *generalization* and *instantiation (specialization)*:

$$\frac{\Gamma \vdash t : A \quad X \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash t : \forall X.\, A} \qquad \frac{\Gamma \vdash t : \forall X.\, A}{\Gamma \vdash t : A[B/X]}$$

  Here $\mathrm{FV}(\Gamma)$ refers to free type variables in the types in Γ.
- (Substitution must be carried out correctly, avoiding type variable capture!)
- In real FP languages, it is restricted where in a type derivation you may generalize and instantiate.

More is typable

- With polymorphic types, one can type more terms, e.g., $\lambda x.\, x\, x$.

$$
\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}}{x : A \vdash x : A \to A} \quad \dfrac{}{x : A \vdash x : A}}{\dfrac{x : A \vdash x\, x : A}{\vdash \lambda x.\, x\, x : A \to A}}
$$

  where $A = \forall X.\, X \to X$.
- Notice that $(X \to X)[A/X] = A \to A$.
- Yet one still cannot type $\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$.

- Type inference in real FP languages does not use the full expressive power of polymorphically typed lambda calculus and does not type as many terms.

Subject reduction and strong normalization

- Subject reduction still holds:
  If $\Gamma \vdash t : A$ and $t \to^* u$, then $\Gamma \vdash u : A$.
- Strong normalization also still holds:
  If $\Gamma \vdash t : A$, then any reduction sequence of t terminates at a normal form.
- As a consequence, every term has a unique normal form.

Encoding datatypes

Encoding booleans

- Datatypes like the types of booleans or natural numbers can be coded up in polymorphically typed lambda-calculus.
- These encodings are known as the Church encodings.

- The Boolean type is encoded like this:

$$
\begin{aligned}
\mathsf{Bool} &= \forall X.\, X \to X \to X \\
\mathsf{tt} &= \lambda t.\, \lambda f.\, t \\
\mathsf{ff} &= \lambda t.\, \lambda f.\, f \\
\mathsf{ite} &= \lambda b.\, \lambda t.\, \lambda f.\, b\; t\; f
\end{aligned}
$$

Encoding booleans ctd

$$\dfrac{\dfrac{\dfrac{\dfrac{}{t : X, f : X \vdash t : X}}{t : X \vdash \lambda f.\, t : X \to X}}{\vdash \lambda t.\, \lambda f.\, t : X \to X \to X}}{\vdash \lambda t.\, \lambda f.\, t : \mathrm{Bool}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{t : X, f : X \vdash f : X}}{t : X \vdash \lambda f.\, f : X \to X}}{\vdash \lambda t.\, \lambda f.\, f : X \to X \to X}}{\vdash \lambda t.\, \lambda f.\, f : \mathrm{Bool}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash b : \mathrm{Bool}}}{\Gamma \vdash b : A \to A \to A} \quad \dfrac{}{\Gamma \vdash t : A}}{\Gamma \vdash b\, t : A \to A} \quad \dfrac{}{\Gamma \vdash f : A}}{b : \mathrm{Bool}, t : A, f : A \vdash b\, t\, f : A}}{b : \mathrm{Bool}, t : A \vdash \lambda f.\, b\, t\, f : A \to A}}{b : \mathrm{Bool} \vdash \lambda t.\, \lambda f.\, b\, t\, f : A \to A \to A}}{\vdash \lambda b.\, \lambda t.\, \lambda f.\, b\, t\, f : \mathrm{Bool} \to A \to A \to A}$$

where $\Gamma = b : \mathrm{Bool}, t : A, f : A$.

Encoding booleans ctd

$$
\begin{array}{rcl}
\mathrm{ite}\;\mathrm{tt}\;u\;v & = & (\lambda b.\,\lambda t.\,\lambda f\,.\,b\;t\;f)\;\mathrm{tt}\;u\;v \\
& \to & (\lambda t.\,\lambda f\,.\,\mathrm{tt}\;t\;f)\;u\;v \\
& \to & (\lambda f\,.\,\mathrm{tt}\;u\;f)\;v \\
& \to & \mathrm{tt}\;u\;v \\
& = & (\lambda t.\,\lambda f\,.\,t)\;u\;v \\
& \to & (\lambda f\,.\,u)\;v \\
& \to & u
\end{array}
$$

$$
\mathrm{ite}\;\mathrm{ff}\;u\;v \quad \to^* \quad v
$$

Encoding booleans ctd

- Using ite, we can define eg

$$
\begin{array}{rcl}
\mathsf{not} & = & \lambda b.\, \mathsf{ite}\, b\, \mathsf{ff}\, \mathsf{tt} \\
\mathsf{and} & = & \lambda b.\ \lambda b'.\, \mathsf{ite}\, b\, b'\, \mathsf{ff} \\
\mathsf{or} & = & \lambda b.\ \lambda b'.\, \mathsf{ite}\, b\, \mathsf{tt}\, b'
\end{array}
$$

Encoding naturals

- The natural number type is encoded like this:

$$
\begin{array}{rcl}
\mathsf{Nat} & = & \forall X.\, X \to (X \to X) \to X \\
\mathsf{Z} & = & \lambda z.\, \lambda s.\, z \\
\mathsf{S} & = & \lambda n.\, \lambda z.\, \lambda s.\, s\,(n\, z\, s) \\
\mathsf{fold} & = & \lambda n.\, \lambda z.\, \lambda s.\, n\, z\, s
\end{array}
$$

- fold behaves as an iterator. One has:

$$
\begin{array}{rcl}
\mathsf{fold}\, \mathsf{Z}\, u\, v & \to^* & u \\
\mathsf{fold}\, (\mathsf{S}\, t)\, u\, v & \to^* & v\, (\mathsf{fold}\, t\, u\, v)
\end{array}
$$

Encoding naturals ctd

- A number n is encoded by

$$\underline{n} = \lambda z.\ \lambda s.\ \underbrace{s\,(\ldots(s}_{n \text{ times}}\ z))$$

- Addition, multiplication, checking equality to 0 can be defined by

$$\begin{array}{rcl} \mathsf{add} & = & \lambda n.\,\lambda m.\,\mathsf{fold}\,n\,m\,\mathsf{S} \\ \mathsf{mult} & = & \lambda n.\,\lambda m.\,\mathsf{fold}\,n\,\mathsf{Z}\,(\mathsf{add}\,m) \\ \mathsf{isZ} & = & \lambda n.\,\mathsf{fold}\,n\,\mathsf{tt}\,(\lambda b.\,\mathsf{ff}) \end{array}$$

Encoding general recursion

- Let

$$\mathsf{Y} f \quad = \quad (\lambda x.\, f\ (x\, x))\, (\lambda x.\, f\, (x\, x))$$

- We have

$$\begin{array}{rcl} \mathsf{Y}\, f & = & (\lambda x.\, f\ (x\, x))\, (\lambda x.\, f\, (x\, x)) \\ & \to & f\, ((\lambda x.\, f\ (x\, x))\, (\lambda x.\, f\, (x\, x))) \\ & = & f\, (\mathsf{Y}\, f) \\ & \to & f\, (f\, (\mathsf{Y}\, f)) \\ & \to & f\, (f\, (f\, (\mathsf{Y}\, f))) \\ & \to & \ldots \end{array}$$

  Y allows for nonterminating reduction sequences.
- Y is not typable. So Y is available only in untyped lambda calculus.

Encoding general recursion ctd

- Y is a general recursion combinator:
  If we want to obtain a term g such that $g \to^* f\, g$ for some fixed f, we can define $g = \mathrm{Y}\, f$.
- E.g., we can define

$$
\begin{array}{rcl}
f & = & \lambda \mathit{fact}'.\, \lambda n.\, \mathrm{ite}\,(n > 0)\, 1\, (n * \mathit{fact}'\,(n-1)) \\
\mathrm{fact} & = & \mathrm{Y}\, f
\end{array}
$$

  Then

$$
\begin{array}{rcl}
\mathrm{fact} & \to & f\, \mathrm{fact} \\
 & = & (\lambda \mathit{fact}'.\, \lambda n.\, \mathrm{ite}\,(n > 0)\, 1\, (n * \mathit{fact}'\,(n-1)))\, \mathrm{fact} \\
 & \to & \lambda n.\, \mathrm{ite}\,(t > 0)\, 1\, (n * \mathrm{fact}\,(n-1))
\end{array}
$$

Encoding general recursion ctd

- Y can be added to typed lambda calculus with typing rule

$$\frac{}{\vdash \mathsf{Y} : (A \to A) \to A}$$

  and reduction rule

$$\mathsf{Y}\, f \to f\, (\mathsf{Y}\, f)$$

- The resulting system no longer has strong normalizability.