Advanced Topics in Programming Languages

Spring semester, 2012

Extensions to Typed Lambda Calculus

Lecturer: Mooly Sagiv *Scribe: Orr Tamir and Bar Avidan*

9.1 Lecture Overview

In this lecture we will extend the typed lambda calculus with some interesting features in our somehow shortest path to Java.

9.2 Typed Lambda Calculus (Review)

9.2.1 Derivation Rules

$t ::=$		terms
	x	variable
	$\lambda x : T.t$	abstraction
	$t\ t$	application

$v ::=$		values
	$\lambda x : T.t$	abstraction values

$T ::=$		types
	$T \to T$	types of functions

$\Gamma ::=$		context
	$\emptyset$	empty context
	$\Gamma, x : T$	term variable binding

9.2.2 Evaluation Rules

$$\frac{t_1 \to t_1'}{t_1 t_t 2 \to t_1' t_2} \text{ (E-APP1)}$$

$$\frac{t_2 \to t_2'}{v_1 t_2 \to v_1 t_2'} \text{ (E-APP2)}$$

$$\frac{(\lambda x : T_{11}.t_{12})v_2}{[x \mapsto v_2]t_{12}} \text{ (E-APPABS)}$$

9.2.3 Type Rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2} \text{ (T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \wedge \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \;\; t_2 : T_{12}} \text{ (T-ABS)}$$

Properties of the type system:

- Uniqueness of types
- Linear time type checking: using the derivation tree backwards(inversion lemma).
- Type Safety
  - Well typed programs cannot go wrong, meaning that there is no undefined semantics and there is no need for runtime checks
- Progress Property: If t is well typed then t is a value or there is an evaluation step $t \to t'$.
- Preservation Property: If t is well typed and there is an evaluation step $t \to t'$ that is also well typed.

9.3 Unit type

This type is equal to the void type that is popular in many languages.

9.3.1 Derivation Rules

t ::= ...		terms
	unit	constant unit

v ::= ...		values
	unit	constant unit

9.3.2 Type Rules

$\Gamma \vdash unit : UNIT$ (T-Unit)

$T ::= ...$		types
	$Unit$	unit type

9.3.3 Derivation Rules

$$\frac{t_1 : T_1; t_2}{(\lambda x : T_1.Unit\ t_2)t_1 \land x \notin FV(t_2)} \text{ (T-ABS)}$$

9.4 Ascription

This extension allows us to ascribe a particular type to a given term.

9.4.1 Derivation Rules

$T ::=$		types
	t as T	

9.4.2 Evaluation Rules

$$\frac{v \text{ as } T}{v} \text{ (E-ASCRIBE)}$$

$$\frac{t \rightarrow t'}{t \text{ as } T \rightarrow t'} \text{ (E-ASCRIBE1)}$$

9.4.3 Type Rules

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \text{ as } T_1 : T} \text{ (T-ASCRIBE)}$$

Note that this extension is only for the comfort of the user and the compiler doesn't benefit from it. However the compiler can warn the user if the declared type is not equal to the proofed type.

There are some situations where ascription can be useful in programming. One common one is documentation. It can sometimes become difficult for a reader to keep track of the types of the subexpressions of a large compound expression. Judicious use of ascription can make such programs much easier to follow. Another use of ascription is as a mechanism for abstraction. In systems where a given term t may have many different types (for example, systems with subtyping), ascription can be used to "hide" some of these types by telling the typechecker to treat t as if it had only a smaller set of types.

9.5 Let Binding

When writing a complex expression, it is often useful - both for avoiding repetition and for increasing readability - to give names to some of its subexpressions. The next extension is similar to local variables definition.

9.5.1 Derivation Rules

$t ::= ...$		terms
	$\text{let } x = t_1 \text{ in } t_2$	let binding

9.5.2 Evaluation Rules

$$\frac{\text{let } x = v_1 \text{ in } t_2}{[x \mapsto v_1]t_2} \text{ (E-LETV)}$$

$$\frac{t_1 \to t_1'}{\text{let } x = t_1 \text{ in } t_2 \to \text{let } x = t_1' \text{ in } t_2} \text{ (E-LET)}$$

9.5.3 Type Rules

$$\frac{\Gamma \vdash t_1 : t \wedge \Gamma.x : T \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2 \to \text{let } x = t_1' \text{ in } t_2} \text{ (E-LET)}$$

In the untyped lambda calculus we would define the let using a lambda abstraction (syntactic sugar). But in the typed lambda calculus we need the type of the bounded variable in the lambda abstraction.

9.6 Pairs

Most programming languages provide a variety of ways of building compound data structures. The simplest of these is pairs, or more generally tuples, of values. We treat pairs in this section, then do the more general cases of tuples and labeled records. In order to support pairs we add two new forms of term, pairing, written $\{t_1, t_2\}$, and projection, written $t.1$ for the first projection from t and $t.2$ for the second projection. In addition we add one new type constructor, $T_1 \times T_2$.

9.6.1 Derivation Rules

t ::= ...		terms
	$\{t, t\}$	pair
	$t.1$	first projection
	$t.2$	second projection

v ::= ...		values
	$\{v, v\}$	pair value

T ::= ...		types
	$T_1 \times T_2$	pair type

9.6.2 Evaluation Rules

$$\frac{\{v_1, v_2\}.1}{v_1} \text{ (E-PairBeta1)}$$

$$\frac{\{v_1, v_2\}.1}{v_2} \text{ (E-PairBeta2)}$$

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

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

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

$$\frac{t_2 \to t_2'}{\{v_1, t_2\} \to \{v_1, t_2'\}} \text{ (E-Pair1)}$$

9.6.3 Type Rules

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

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

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

Note that the evaluation order defined in this semantics implies an eager evaluation. We could have define a lazy evaluation but with a cost of non-determinism and harder proofs.

9.6.4 Examples

- {pred 4, if true then false else false}.1
  - {pred 4, if true then false else false}.1
  - {3, if true then false else false}.1
  - {3, false}.1
  - 3
- (λx:Nat × Nat. x.2) {pred 4, pred 5}
  - (λx:Nat × Nat. x.2) {pred 4, pred 5}
  - (λx:Nat × Nat. x.2) {3, pred 5}
  - (λx:Nat × Nat. x.2) {3,4}
  - {3,4}.2
  - 4

9.7 Tuples

This extension is a generalization of Pairs.

9.7.1 Derivation Rules

$t ::= ...$		terms
	$\{_{t_i}\}_{i \in 1..n}$	tuple
	$t.i$	projection

$v ::= ...$		values
	$\{v_i\}_{i \in 1..n}$	tuple value

$T ::= ...$		types
	$T_1 \times T_2 ... \times T_n$	tuple type

9.7.2 Evaluation Rules

$$\frac{\{v_i\}_{i \in 1..n}.j}{v_j} \text{ (E-ProjTuple)}$$

$$\frac{t \to t'}{t.j \to t.j'} \text{ (E-Proj)}$$

$$\frac{t_j \to t'_j}{\{v_1, .., v_{j-1}, t_j, ..t_n\} \to \{v_1, .., v_{j-1}, t'_j, ..t_n\}} \text{ (E-Tuple)}$$

9.7.3 Type Rules

$$\frac{\forall i, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_i\}_{1..n} : T_1 \times T_2 ... \times T_n} \text{ (T-Tuple)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2 ... \times T_n}{\Gamma \vdash t.i : T_i} \text{ (T-Proj)}$$

For example, {1,2,true} is a 3-tuple containing two numbers and a boolean. Its type is written {Nat,Nat,Bool}.

9.8 Records

Records allow an elegant to organize data. In C# or Java they are very popular (Map,Dictionary). They are also standard in ML, Haskel and Scala. We could also define them in the untyped lambda calculus.

9.8.1 Derivation Rules

$t ::= ...$		terms
	$\{l_i = t_i\}_{i \in 1..n}$	record
	$t.l$	projection

$v ::= ...$		values
	$\{l_i = v_i\}_{i \in 1..n}$	tuple value

$T ::= ...$		types
	$l_1 \to T_1 \times l_2 \to T_2 ... \times l_n \to T_n$	Record type

9.8.2 Evaluation Rules

$$\frac{\{l_i = v_i\}_{i \in 1..n}.l_j}{v_j} \text{ (E-ProjRCD)}$$

$$\frac{t \to t'}{t.l \to t.l'} \text{ (E-Proj)}$$

$$\frac{t_j \to t'_j}{\{l_1 = v_1, .., l_{j-1} = v_{j-1}, l_j = t_j, ..l_n = t_n\} \to \{l_1 = v_1, .., l_{j-1} = v_{j-1}, l_j = t_j, ..l_n = t_n\}} \text{ (E-RCD)}$$

9.8.3 Type Rules

$$\frac{\forall i, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i\}_{1..n} : l_1 \to T_1 \times l_2 \to T_2 ... \times l_n \to T_n} \text{ (T-RCD)}$$

$$\frac{\Gamma \vdash t : l_1 \to T_1 \times l_2 \to T_2 ... \times l_n \to T_n}{\Gamma \vdash t.l_i : T_i} \text{ (T-ProjRCD)}$$

The generalization from $n - ary$ tuples to labelled records is equally straightforward. We simply annotate each field t_j with a label l_i drawn from some predetermined set L. For example, {x=5} and {partno=5524,cost=30.27} are both record values, their types are {x:Nat} and {partno:Nat,cost:Float}. We require that all the labels in a given record term or type be distinct.

9.9 Pattern Matching

- An elegant way to access records
- Checked by the compiler
- Shortens the code
- Standard in ML, Haskel, Scala
- Can be expressed in the untyped lambda calculus

We do not discuss the subject here.

9.10 Sums

Many programs need to deal with heterogeneous collections of values. For example, a list cell can be either *nil* or a *cons* cell carrying a head and a tail, a node of an abstract syntax tree in a compiler can represent a variable, an abstraction, an application, etc. The type-theoretic mechanism that supports this kind of programming is variant types. Before introducing variants in full generality, let us consider the simpler case of binary sum types. A sum type describes a set of values drawn from exactly two given types.

9.10.1 Derivation Rules

$t ::= ...$		terms
	$inl \;\; t$	tagging(left)
	$inr \;\; t$	tagging(right)
	$case\ t \;\; \text{of} \;\; inl \;\; x \Rightarrow t \mid inr \;\; x \Rightarrow t$	case

$v ::= ...$ values
$inl\ \ v$ tagged value(left)
$inr\ \ v$ tagged value(right)

$T ::= ...$ types
$T_1 + T_2$ sum type

9.10.2 Evaluation Rules

$$\frac{case\ (inl\ \ v)\ \text{ of }\ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2}{[x_1 \mapsto v]t_1} \text{ (E-CaseINL)}$$

$$\frac{case\ (inr\ \ v)\ \text{ of }\ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2}{[x_2 \mapsto v]t_2} \text{ (E-CaseINR)}$$

$$\frac{t \rightarrow t'}{case\ t\ \text{ of }\ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2 \rightarrow case\ t'\ \text{ of }\ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2} \text{ (E-Case)}$$

$$\frac{t_1 \rightarrow t_1'}{inl\ \ t_1 \rightarrow inl\ t_1'} \text{ (E-INL)}$$

$$\frac{t_1 \rightarrow t_1'}{inr\ \ t_1 \rightarrow inl\ t_1'} \text{ (E-INR)}$$

9.10.3 Type Rules

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash inl\ \ t_1 : T_1 + T_2} \text{ (T-INL)}$$

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash inr\ \ t_1 : T_1 + T_2} \text{ (T-INR)}$$

$$\frac{\Gamma, x_1 : T1 \vdash t_1 : T \wedge \Gamma, x_2 : T_2 \vdash t_1 : T \wedge \Gamma \vdash t : T_1 + T_2}{\Gamma \vdash case\ t\ \text{ of }\ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2 : T} \text{ (T-CASE)}$$

9.10.4 Example

Suppose we are using the types
PhysicalAddr = {firstlast:String, addr:String};
VirtualAddr = {name:String, email:String};
to represent different sorts of address-book records. If we want to manipulate both sorts of records uniformly we can introduce the sum type:
Addr = PhysicalAddr + VirtualAddr;
each of whose elements is either a $PhysicalAddr$ or a $VirtualAddr$.
We create elements of this type by tagging elements of the component types $PhysicalAddr$

and $VirtualAddr$. For example, if *pa* is a $PhysicalAddr$, then *inl pa* is an $Addr$. (The names of the tags *inl* and *inr* arise from thinking of them as functions that "inject" elements of $PhysicalAddr$ or $VirtualAddr$ into the left and right components of the sum type Addr).

inl : PhysicalAddr $\rightarrow$ *Addr*
inr : VirtualAddr $\rightarrow$ *Addr*

We use a *case* construct that allows us to distinguish whether a given value comes from the left or right branch of a sum. For example, we can extract a name from an *Addr* like this:

$$
\begin{aligned}
&getName = \lambda a{:}Addr.\\
&\qquad case\ a\ of\\
&\qquad\quad inl\ x \Rightarrow x.firstlast\\
&\qquad\ \mid inr\ y \Rightarrow y.name;
\end{aligned}
$$

The type of the whole $getName$ function is *Addr* $\rightarrow$ $String$.

9.11 Sums with Unique Types

With sums, as we defined them, most of the properties of the type system are preserved, but one rule fails: the Uniqueness of Types. The difficulty arises from the tagging constructs *inl* and *inr*. The typing rule $T - INL$, for example, says that, once we have shown that t_1 is an element of T_1, we can derive that *inl* t_1 is an element of $T_1 + T_2$ for any type T_2. For example, we can derive both *inl 5 : Nat+Nat* and *inl 5 : Nat+Bool* (and infinitely many other types). The failure of uniqueness of types means that we cannot build a typechecking algorithm using the inversion lemma.
Potential solutions:

- Use type reconstruction (not shown).
- Use subtyping (not shown).
- User annotation, as we show next.

9.11.1 Derivation Rules

$t ::= ...$		terms
	$inl\ \ t\ \ as\ T$	tagging(left)
	$inr\ \ t\ \ as\ T$	tagging(right)
	$case\ t\ \ \text{of}\ \ inl\ \ x \Rightarrow t \vert inr\ \ x \Rightarrow t$	case

$v ::= ...$		values
	$inl\ \ v\ \ as\ T$	tagged value(left)
	$inr\ \ v\ \ as\ T$	tagged value(right)

$T ::= ...$ types
$T_1 + T_2$ sum type

9.11.2 Evaluation Rules

$$\frac{case\ (inl\ \ v\ as\ T\ )\ \ \text{of}\ \ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2}{[x_1 \mapsto v]t_1}\ \text{(E-CaseINL)}$$

$$\frac{case\ (inr\ \ v\ as\ T\ )\ \ \text{of}\ \ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2}{[x_2 \mapsto v]t_2}\ \text{(E-CaseINR)}$$

$$\frac{t \to t'}{case\ t\ \ \text{of}\ \ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2 \to case\ t'\ \ \text{of}\ \ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2}\ \text{(E-Case)}$$

$$\frac{t_1 \to t_1'}{inl\ \ t_1\ \ as\ T \to inl\ t_1'\ \ as\ T}\ \text{(E-INL)}$$

$$\frac{t_1 \to t_1'}{inr\ \ t_1\ \ as\ T \to inl\ t_1'\ \ as\ T}\ \text{(E-INR)}$$

9.11.3 Type Rules

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash inl\ \ t_1\ \ as\ T_1 + T_2 : T_1 + T_2}\ \text{(T-INL)}$$

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash inr\ \ t_1\ \ as\ T_1 + T_2 : T_1 + T_2}\ \text{(T-INR)}$$

$$\frac{\Gamma, x_1 : T1 \vdash t_1 : T \wedge \Gamma, x_2 : T_2 \vdash t_1 : T \wedge \Gamma \vdash t : T_1 + T_2}{\Gamma \vdash case\ t\ \ \text{of}\ \ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2 : T}\ \text{(T-CASE)}$$

9.12 Variants

Binary sums generalize to variants just as tuples generalize to records. Instead of $T_1 + T_2$, we write $< l_1 : T_1, l_2 : T2 >$, where l_1 and l_2 are field labels. Instead of *inl t* as $T_1 + T_2$, we write $< l_1 = t >$ as $< l_1 : T_1, l_2 : T_2 >$. And instead of labelling the branches of the case with *inl* and *inr*, we use the same labels as the corresponding sum type.

9.12.1 Derivation Rules

$t ::= ...$		terms
	$< l = t > \ \ as \ \ T$	tagging
	$case\ t\ \ \text{of}\ \ < l_i = x_i > \ \ x \Rightarrow t_i, i \in 1..n$	case

$v ::= ...$		values
	$< l = v > as \ \ T$	tagged value

$T ::= ...$		types
$< l_i = T_i >_{i \in 1...n}$	type of variants type	

9.12.2 Evaluation Rules

$$\frac{case\ (< l_j = v > as\ T\ )\ \ \text{of}\ \ < l_i = x_i \Rightarrow t_i, i \in 1...n \ \ x_2 \Rightarrow t_2}{[x_j \mapsto v]t_j} \text{ (E-CaseVariant)}$$

$$\frac{t \to t'}{case\ t\ \ \text{of}\ \ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2 \to case\ t'\ \ \text{of}\ \ inl\ \ x_1 \Rightarrow t_1 | inr\ \ x_2 \Rightarrow t_2} \text{ (E-Case)}$$

$$\frac{t_i \to t_i'}{< l_i = t_i > \ \ as\ \ T \to < l_i = t_i' > \ \ as\ \ T} \text{ (E-VARIANT)}$$

9.12.3 Type Rules

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash < l_j = t_j > as < l_i = T_i >, i = 1..n :< l_i = T_i >, i = 1..n} \text{ (T-VARIANT)}$$

$$\frac{\forall i, \Gamma, x_i : Ti \vdash t_i : T \wedge \Gamma \vdash t :< l_i = T_i >, i = 1..n}{\Gamma \vdash case\ t\ \ \text{of}\ \ < l_i = x_i > \Rightarrow t_i, i = 1..n : T} \text{ (T-CASE)}$$

9.12.4 Example

We now look back at the Sums' section example.
PhyisicalAddr = {firstlast: String, add:String}
VirtualAddr = {name: String, email:String}
Addr = <physical: PhisicalAddr, virtual: VirtualAddr>
 getName = λa:Addr.
  case a of
   <physical=x> ⇒ x.firstlast
   | <virtual=y> ⇒ y.name;

$getName : Addr \to String$

9.12.5 Uses of Varients

- **Options (optional values)**
  Consider this type definition:

  $OptionalNat =< none : Unit, some : Nat >;$

  An element of this type is either the trivial unit value with the tag none or else a number with the tag *some*. In other words, the type *OptionalNat* is isomorphic to Nat extended with an additional distinguished value *none*. For example, the type

  $Table = Nat \rightarrow OptionalNat;$

  represents finite mappings from numbers to numbers: the domain of such a mapping is the set of inputs for which the result is $< some = n >$ for some n.
  $emptyTable = \lambda n : Nat. < none = unit >$ as $OptionalNat;$
  $emptyTable : Table$

  This is a constant function that returns *none* for every input.

  $extendTable =$
  $\quad \lambda t : Table.m : Nat.v : Nat.$
  $\quad\quad \lambda n : Nat.$
  $\quad\quad\quad$ *if equal n m then* $<$ *some* $= v >$ *as OptionalNat*
  $\quad\quad\quad\quad$ *else t n;*

  $extendTable : Table \rightarrow Nat \rightarrow Nat \rightarrow Table$
  extendTable takes a table and adds (or overwrites) an entry mapping the input m to the output $< some = v >$.

  We can use the result that we get back from a Table lookup by wrapping a case around it. For example, if t is our table and we want to look up its entry for 5, we might write

  $x =$ *case t(5) of*
  $\quad < none = u > \ \Rightarrow 999$
  $\quad | < some = v > \ \Rightarrow v;$
  providing 999 as the default value of x in case t is undefined on 5.

- **Enumeration**
  An enumeration is a variant type in which the field type associated with each label is Unit. For example, a type representing the days of the working week might be defined as:
  $Weekday =< monday : Unit, tuesday : Unit, wednesday : Unit,$
  $\quad\quad thursday : Unit, friday : Unit >;$

Since the type Unit has only *unit* as a member, the type Weekday is inhabited by precisely five values, corresponding one-for-one with the days of the week. The *case* construct can be used to define computations on enumerations.

$$
\begin{aligned}
&nextBusinessDay = w : Weekday.\\
&\quad case\ w\ of\\
&\qquad < monday = x > \Rightarrow < tuesday = unit > \ as\ Weekday\\
&\quad\ \ | < thursday = x > \Rightarrow < wednesday = unit > \ as\ Weekday\\
&\quad\ \ | < wednesday = x > \Rightarrow < thursday = unit > \ as\ Weekday\\
&\quad\ \ | < tuesday = x > \Rightarrow < friday = unit > \ as\ Weekday\\
&\quad\ \ | < friday = x > \Rightarrow < monday = unit > \ as\ Weekday;
\end{aligned}
$$

- **Single-Field Variants**
  Another interesting special case is variant types with just a single label l:

  $V =< l : T >;$

  Although is doesn't seem very useful, it can be helpful in some situations. The main use of such types is to enforce some behaviour of the program.
  For example, suppose we are writing a program to do financial calculations in multiple currencies. Such a program might include functions for converting between dollars and euros. If both are represented as Floats, then these functions might look like this:
  $dollars2euros = \lambda d : Float.timesfloat\ d\ 1.1325;$
  $dollars2euros : Float \to Float$
  $euros2dollars = \lambda e : Float.timesfloat\ e\ 0.883;$
  $euros2dollars : Float \to Float$
  Suppose we then start with a dollar amount
  $mybankbalance = 39.50;$
  We can easily perform manipulations that make no sense at all. For example, we can convert my bank balance to euros twice:
  dollars2euros (dollars2euros mybankbalance);

  Since all our amounts are represented simply as floats, there is no way that the type system can help prevent this sort of nonsense. However, if we define dollars and euros as different variant types (whose underlying representations are floats) then we can define safe versions of the conversion functions that will only accept amounts in the correct currency:

  $DollarAmount =< dollars : Float >;$
  $EuroAmount =< euros : Float >;$

  dollars2euros =
  λd:DollarAmount.

case d of < dollars = x >⇒
< euros = timesfloat x 1.1325> as EuroAmount;

dollars2euros : DollarAmount → EuroAmount

euros2dollars =
λe:EuroAmount.
case e of < euros = x >⇒
< dollars = timesfloat x 0.883> as DollarAmount;
euros2dollars : EuroAmount → DollarAmount

Now the typechecker can track the currencies used in our calculations and remind us how to interpret the final results:

mybankbalance = < dollars = 39.50 *> as DollarAmount;*
euros2dollars (dollars2euros mybankbalance);

< dollars = 39.49990125 *> as DollarAmount : DollarAmount*

Moreover, if we write a nonsensical double-conversion, the types will fail to match and our program will (correctly) be rejected:

dollars2euros (dollars2euros mybankbalance);

Will cause an "Error: parameter type mismatch".

- **Dynamic Types**
  Even in statically typed languages, there is often the need to deal with data whose type cannot be determined at compile time. This occurs in particular when the lifetime of the data spans multiple machines or many runs of the compiler, when, for example, the data is stored in an external file system or database, or communicated across a network. To handle such situations safely, many languages offer facilities for inspecting the types of values at run time.
  We can use Varients to address this issue, we do not show it here.

9.13 General Recursion

We have seen that, in the untyped lambda-calculus, recursive functions can be defined with the aid of the fix combinator.

Recursive functions can be defined in a typed setting in a similar way. For example, here is a function iseven that returns true when called with an even argument and false otherwise:

ff = ie:NatBool.
λx:Nat.
if iszero x then true
else if iszero (pred x) then false
else ie (pred (pred x));

$ff : (Nat \to Bool) \to Nat \to Bool$

iseven = fix ff;
$iseven : Nat \to Bool$

iseven 7;
$false : Bool$

However, there is one important difference from the untyped setting: fix itself cannot be defined in the simply typed lambda-calculus.

9.14 Recursion

We add the *letrec* syntax in order to solve the problem of the fix combinator, which is missing on the typed lambda calculus, so now we can use recursion, as shown next.

9.14.1 Derivation Rules

$t ::= ...$		terms
	$fix\ \ t$	fixed point of t
	$letrec\ \ x : T_1 = t_1\ \ in\ \ t_2 \doteq let\ \ x = (fix(\lambda x : T_1.t_1))\ \ in\ \ t_2$	letrec

9.14.2 Evaluation Rules

$$\frac{fix(\lambda x : T1.t_2))}{[x \mapsto (fix(\lambda x : T_1.t_2)]t_2}\ \text{(E-FixBeta)}$$

$$\frac{t \to t'}{fix\ t \to fix\ t'}\ \text{(E-FIX)}$$

9.14.3 Type Rules

$$\frac{\Gamma \vdash t.T_1 \to T_1}{\Gamma \vdash fix\ t : T_1}\ \text{(T-REFER)}$$

9.15 Lists

The typing features we have seen can be classified into base types like *Bool* and $Unit$, and *typeconstructors* like $\to$ and $\times$ that build new types from old ones. Another useful type constructor is *List*. For every type T, the type *List* T describes finite-length lists whose elements are drawn from T. Lists are straightforward to define, but they will become more interesting with polymorphism.

9.16 References

Nearly every programming language provides some form of assignment operation that changes the contents of a previously allocated piece of storage. This mechanism is called reference and it is used to mutate the store (or a heap). With references we can also make aliases of variables, allocate memory, and free allocated memory.

9.16.1 Basics

The basic operations on references are *allocation*, *dereferencing*, and *assignment*. To allocate a reference, we use the *ref* operator, providing an initial value for the new cell.

r = ref 5;
r : Ref Nat

The response from the typechecker indicates that the value of r is a reference to a cell that will always contain a number. To read the current value of this cell, we use the dereferencing operator !.

!r;
5 : Nat

To change the value stored in the cell, we use the assignment operator.

r := 7;
unit : Unit

If we dereference r again, we see the updated value.

!r;
7 : Nat

9.16.2 References and Aliasing

It is important to bear in mind the difference between the reference that is bound to r and the *cell* in the store that is pointed to by this reference.

r = ref 13;

If we make a copy of r, for example by binding its value to another variable s,

$s = r$;
s : Ref Nat

what gets copied is only the reference, not the cell. We can verify this by assigning a new value into s:

s := 82;
unit : Unit

And if we read it out via r:

!r;
$82 : Nat$
The references r and s are said to be aliases for the same cell.

9.16.3 Dangling References

Dangling reference problem: we allocate a cell holding a number, save a reference to it in some data structure, use it for a while, then deallocate it and allocate a new cell holding a boolean, possibly reusing the same storage. Now we can have two names for the same storage cell: one with type *Ref Nat* and the other with type *Ref Bool.* Deallocation causes confusion, and violates type safety. The common way to address this problem in modern languages, is to use a garbage collector, which deallocates every memory cell which is not reachable from the program.

9.16.4 Type Rules

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash ref\ t : ref\ T} \text{ (T-REF)}$$

$$\frac{\Gamma \vdash t : ref\ T}{\Gamma \vdash !t : T} \text{ (T-DEREF1)}$$

$$\frac{\Gamma \vdash t_1 : ref\ T \wedge \Gamma t_2 : T}{\Gamma \vdash t_1 = \&t_2 : Unit} \text{ (T-REFER)}$$

This typing method is simplifying the reality, since it does not refer to memory and allocations at all. It is possible to formalize the references' operational behaviour, and use then in a type safe way, on our type system.

9.17 Summary

Typed lambda calculus can be extended to cover many programming language features. It is possible to enforce type safety in realistic situations.

Bibliography

[1] Pierce, Benjamin C., *Types and Programming Languages*, The MIT Press, 2002, pp. 118-146, 153-170.