

Formal Systems and their Applications

Course nr: H04H8A

Frank Piessens
(Frank.Piessens@cs.kuleuven.be)

Acknowledgment: these slides are based in part on slides from Benjamin Pierce

Course Overview

- Introduction
- Formal models of programming languages
- Simple type systems
 - Typing an expression language
 - Simply typed lambda calculus
 - Typing simple programming language features: records, variants, lists, references, exceptions, ...
- Subtyping
- Conclusion

Simple Type Systems

Typing an expression language

- Recap: arithmetic expressions
 - Syntax, semantics
 - Run-time errors
- The Typing Relation
- Type soundness
- Wrap-up

Recap: syntax

<code>t ::=</code>	<code>true</code> <code>false</code> <code>if t then t else t</code> <code>0</code> <code>succ t</code> <code>pred t</code> <code>iszero t</code>	<i>terms</i> <i>constant true</i> <i>constant false</i> <i>conditional</i> <i>constant zero</i> <i>successor</i> <i>predecessor</i> <i>zero test</i>
<code>v ::=</code>	<code>true</code> <code>false</code> <code>nv</code>	<i>values</i> <i>true value</i> <i>false value</i> <i>numeric value</i>
<code>nv ::=</code>	<code>0</code> <code>succ nv</code>	<i>numeric values</i> <i>zero value</i> <i>successor value</i>

Recap: semantics (1)

if true then t_2 else $t_3 \longrightarrow t_2$ (E-IFTRUE)

if false then t_2 else $t_3 \longrightarrow t_3$ (E-IFFALSE)

$$\frac{t_1 \longrightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3} \quad (\text{E-IF})$$

Recap: semantics (2)

$$\frac{t_1 \longrightarrow t'_1}{\text{succ } t_1 \longrightarrow \text{succ } t'_1} \quad (\text{E-SUCC})$$

$$\text{pred } 0 \longrightarrow 0 \quad (\text{E-PREDZERO})$$

$$\text{pred } (\text{succ } nv_1) \longrightarrow nv_1 \quad (\text{E-PREDSUCC})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{pred } t_1 \longrightarrow \text{pred } t'_1} \quad (\text{E-PRED})$$

$$\text{iszero } 0 \longrightarrow \text{true} \quad (\text{E-ISZEROZERO})$$

$$\text{iszero } (\text{succ } nv_1) \longrightarrow \text{false} \quad (\text{E-ISZEROSUCC})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{iszero } t_1 \longrightarrow \text{iszero } t'_1} \quad (\text{E-ISZERO})$$

Run-time errors

- The goal of a type system is to avoid “getting stuck”
 - E.g. never reach: succ true
- The type system will do this by
 - Statically classifying terms according to “the type of value” they compute
 - Ensuring that only the right “types” of values are used in each part of the program
 - E.g. only a boolean is used as condition in a test

The Typing Relation

- We will distinguish two “types of values” (i.e. two *types*): natural numbers and Booleans

$T ::=$	<i>types</i>
Bool	<i>type of booleans</i>
Nat	<i>type of numbers</i>

- And we will define a relation between terms and types
 - $t:T$ *-- read as “t is of type T”*
 - This relation is defined inductively, using inference rules

The Typing Rules

$\text{true} : \text{Bool}$ (T-TRUE)

$\text{false} : \text{Bool}$ (T-FALSE)

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$
 (T-IF)

$0 : \text{Nat}$ (T-ZERO)

$$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}}$$
 (T-SUCC)

$$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}}$$
 (T-PRED)

$$\frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}}$$
 (T-ISZERO)

Example of a type derivation tree

$$\frac{\frac{\frac{}{0 : \text{Nat}} \text{T-ZERO}}{\text{iszero } 0 : \text{Bool}} \text{T-ISZERO} \quad \frac{\frac{}{0 : \text{Nat}} \text{T-ZERO}}{\text{pred } 0 : \text{Nat}} \text{T-PRED}}{\text{if iszero } 0 \text{ then } 0 \text{ else pred } 0 : \text{Nat}} \text{T-IF}$$

Type Soundness

- The type system is designed to avoid run-time errors
 - Now, we should prove it does this correctly: A well-typed program does not get stuck
- Note that the converse is not generally true
 - Programs that do not get stuck are not necessarily well-typed
 - E.g. if true then 1 else succ false
 - Typically unavoidable because of the static nature of typing

Proving Type Soundness

The safety (or soundness) of this type system can be expressed by two properties:

1. *Progress*: A well-typed term is not stuck

If $t : T$, then either t is a value or else $t \longrightarrow t'$ for some t' .

2. *Preservation*: Types are preserved by one-step evaluation

If $t : T$ and $t \longrightarrow t'$, then $t' : T$.

Lemma: Inversion

Lemma:

1. If `true` : R , then $R = \text{Bool}$.
2. If `false` : R , then $R = \text{Bool}$.
3. If `if` t_1 `then` t_2 `else` t_3 : R , then t_1 : Bool , t_2 : R , and t_3 : R .
4. If `0` : R , then $R = \text{Nat}$.
5. If `succ` t_1 : R , then $R = \text{Nat}$ and t_1 : Nat .
6. If `pred` t_1 : R , then $R = \text{Nat}$ and t_1 : Nat .
7. If `iszero` t_1 : R , then $R = \text{Bool}$ and t_1 : Nat .

Progress and Preservation

- Canonical Forms Lemma:

Lemma:

1. If v is a value of type `Bool`, then v is either `true` or `false`.
2. If v is a value of type `Nat`, then v is a numeric value.

- Progress:

Theorem: Suppose t is a well-typed term (that is, $t : T$ for some T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

- Preservation:

Theorem: If $t : T$ and $t \longrightarrow t'$, then $t' : T$.

Wrap-up

- “Type-checking” consists of a static check of the program, such that runtime errors are avoided
- Proving a type system sound:
 - Generalize “well-typed” to program states
 - Is trivial here
 - Prove that well-typed program states do not get stuck (Progress)
 - Prove that “being well-typed” is preserved by evaluation steps (Preservation)

Course Overview

- Introduction
- Formal models of programming languages
- Simple type systems
 - Typing an expression language
 - Simply typed lambda calculus
 - Typing simple programming language features: records, variants, lists, references, exceptions, ...
- Subtyping
- Conclusion

Simply Typed Lambda Calculus

- We will study a series of type systems for the lambda calculus.
- This first one is more or less the “simplest one possible”
 - We need at least a “type of functions”: \rightarrow
 - But in order to distinguish higher-order functions, we need more than just a “type of functions”

$T ::=$	<i>types</i>
Bool	<i>type of booleans</i>
$T \rightarrow T$	<i>types of functions</i>

Lambda calculus + Booleans

<code>t ::=</code>	<i>terms</i>
<code> x</code>	<i>variable</i>
<code> $\lambda x.t$</code>	<i>abstraction</i>
<code> t t</code>	<i>application</i>
<code> true</code>	<i>constant true</i>
<code> false</code>	<i>constant false</i>
<code> if t then t else t</code>	<i>conditional</i>
<code>v ::=</code>	<i>values</i>
<code> $\lambda x.t$</code>	<i>abstraction value</i>
<code> true</code>	<i>true value</i>
<code> false</code>	<i>false value</i>

The Typing Relation

- Typing lambda-calculus expressions introduces several challenges:
 1. In an abstraction: $\lambda x. t_2$
 - What is the type of x ?
 - Two options: infer or annotate
 - For the moment we go for the simpler option, annotate: $\lambda x:T_1. t_2$
 2. How do we deal with terms with free variables?

Dealing with free variables...

`true : Bool` (T-TRUE)

`false : Bool` (T-FALSE)

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-IF})$$

$$\frac{\text{???}}{\lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

... by introducing context information.

The context Γ (Gamma) is a (finite) mapping of variables to types

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

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

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-IF})$$

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

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

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \quad (\text{T-APP})$$

Properties

- The two key properties are:
 - Progress:

A closed, well-typed term is not stuck
*If $\vdash t : T$, then either t is a value or else $t \longrightarrow t'$
for some t' .*

- Preservation:

If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

- To prove them, we proceed in a similar way as for expressions

Inversion Lemma

1. If $\Gamma \vdash \text{true} : R$, then $R = \text{Bool}$.
2. If $\Gamma \vdash \text{false} : R$, then $R = \text{Bool}$.
3. If $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $\Gamma \vdash t_1 : \text{Bool}$ and $\Gamma \vdash t_2, t_3 : R$.
4. If $\Gamma \vdash x : R$, then $x : R \in \Gamma$.
5. If $\Gamma \vdash \lambda x : T_1. t_2 : R$, then $R = T_1 \rightarrow R_2$ for some R_2 with $\Gamma, x : T_1 \vdash t_2 : R_2$.
6. If $\Gamma \vdash t_1 \ t_2 : R$, then there is some type T_{11} such that $\Gamma \vdash t_1 : T_{11} \rightarrow R$ and $\Gamma \vdash t_2 : T_{11}$.

Uniqueness and canonical forms

- Uniqueness:
 - In a given context Γ , if a term is typable, then it is only in one way
- 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 $T_1 \rightarrow T_2$, then v has the form $\lambda x:T_1. t_2$.

Progress

- Progress theorem:

Theorem: Suppose t is a closed, well-typed term (that is, $\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

- Proof is by induction on the typing derivation
- Note: if the term is not closed, progress can fail

Preservation

- Substitution Lemma:

Lemma: Types are preserved under substitution.

That is, if $\Gamma, x:S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.

- Preservation Theorem:

Theorem: If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

- Proof by induction on typing derivation

Side note: Curry-Howard correspondence

- There is a very precise connection between formal proofs in logic and type checking

LOGIC	PROGRAMMING LANGUAGES
propositions	types
proposition $P \supset Q$	type $P \rightarrow Q$
proposition $P \wedge Q$	type $P \times Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term) evaluation

Erase and typability

- Erasure: stripping type information from a term

$$\begin{aligned} \text{erase}(x) &= x \\ \text{erase}(\lambda x:T_1. t_2) &= \lambda x. \text{erase}(t_2) \\ \text{erase}(t_1 t_2) &= \text{erase}(t_1) \text{erase}(t_2) \end{aligned}$$

- Typability: can we add type information such that a term becomes well-typed?
- Proving that “evaluation commutes with erasure” shows that typing information can be forgotten after type checking

Course Overview

- Introduction
- Formal models of programming languages
- Simple type systems
 - Typing an expression language
 - Simply typed lambda calculus
 - Typing simple programming language features: records, variants, lists, references, exceptions, ...
- Subtyping
- Conclusion

Adding Other Base Types

- We discussed the simply typed lambda calculus with Booleans.
- Adding other base types such as natural numbers, strings, or floats is straightforward

The Unit Type

- Similar to “void” in Java

$t ::= \dots$	<i>terms</i>
unit	<i>constant unit</i>
$v ::= \dots$	<i>values</i>
unit	<i>constant unit</i>
$T ::= \dots$	<i>types</i>
Unit	<i>unit type</i>

New typing rules

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

$\Gamma \vdash \text{unit} : \text{Unit}$

(T-UNIT)

Sequential Composition

$t ::= \dots$
 $t_1; t_2$

terms

$$\frac{t_1 \longrightarrow t'_1}{t_1; t_2 \longrightarrow t'_1; t_2} \quad (\text{E-SEQ})$$

$$\text{unit}; t_2 \longrightarrow t_2 \quad (\text{E-SEQNEXT})$$

$$\frac{\Gamma \vdash t_1 : \text{Unit} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2} \quad (\text{T-SEQ})$$

Can be seen as “syntactic sugar”, or as a “derived form”:

$$t_1; t_2 \stackrel{\text{def}}{=} (\lambda x : \text{Unit}. t_2) t_1$$

where $x \notin FV(t_2)$

Type Ascription

New syntactic forms

$t ::= \dots$
 $t \text{ as } T$

New evaluation rules

$v_1 \text{ as } T \longrightarrow v_1$

$$\frac{t_1 \longrightarrow t'_1}{t_1 \text{ as } T \longrightarrow t'_1 \text{ as } T}$$

New typing rules

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T}$$

terms

ascription

$t \longrightarrow t'$

(E-ASCRIIBE)

(E-ASCRIIBE1)

$\Gamma \vdash t : T$

(T-ASCRIIBE)

Let Bindings

New syntactic forms

$t ::= \dots$
 $\text{let } x=t \text{ in } t$

terms

let binding

New evaluation rules

$t \longrightarrow t'$

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

$$\frac{t_1 \longrightarrow t'_1}{\text{let } x=t_1 \text{ in } t_2 \longrightarrow \text{let } x=t'_1 \text{ in } t_2}$$
 (E-LET)

New typing rules

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2}$$
 (T-LET)

Pairs

$t ::= \dots$	<i>terms</i>
$\{t, t\}$	<i>pair</i>
$t.1$	<i>first projection</i>
$t.2$	<i>second projection</i>
$v ::= \dots$	<i>values</i>
$\{v, v\}$	<i>pair value</i>
$T ::= \dots$	<i>types</i>
$T_1 \times T_2$	<i>product type</i>

Pairs (ctd)

$$\{v_1, v_2\}.1 \longrightarrow v_1 \quad (\text{E-PAIRBETA1})$$

$$\{v_1, v_2\}.2 \longrightarrow v_2 \quad (\text{E-PAIRBETA2})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1.1 \longrightarrow t'_1.1} \quad (\text{E-PROJ1})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1.2 \longrightarrow t'_1.2} \quad (\text{E-PROJ2})$$

$$\frac{t_1 \longrightarrow t'_1}{\{t_1, t_2\} \longrightarrow \{t'_1, t_2\}} \quad (\text{E-PAIR1})$$

$$\frac{t_2 \longrightarrow t'_2}{\{v_1, t_2\} \longrightarrow \{v_1, t'_2\}} \quad (\text{E-PAIR2})$$

Pairs (ctd)

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

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \quad (\text{T-PROJ1})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \quad (\text{T-PROJ2})$$

Straightforward to extend to n-tuples, or records (where order matters)

Disjoint Sums

<code>t ::= ...</code>	<i>terms</i>
<code>inl t</code>	<i>tagging (left)</i>
<code>inr t</code>	<i>tagging (right)</i>
<code>case t of inl x⇒t inr x⇒t</code>	<i>case</i>
<code>v ::= ...</code>	<i>values</i>
<code>inl v</code>	<i>tagged value (left)</i>
<code>inr v</code>	<i>tagged value (right)</i>
<code>T ::= ...</code>	<i>types</i>
<code>T+T</code>	<i>sum type</i>

Disjoint Sums (ctd)

$$\begin{array}{l} \text{case (inl } v_0) \\ \text{of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \end{array} \longrightarrow [x_1 \mapsto v_0]t_1 \quad (\text{E-CASEINL})$$
$$\begin{array}{l} \text{case (inr } v_0) \\ \text{of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \end{array} \longrightarrow [x_2 \mapsto v_0]t_2 \quad (\text{E-CASEINR})$$
$$\frac{t_0 \longrightarrow t'_0}{\text{case } t_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \longrightarrow \text{case } t'_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2} \quad (\text{E-CASE})$$
$$\frac{t_1 \longrightarrow t'_1}{\text{inl } t_1 \longrightarrow \text{inl } t'_1} \quad (\text{E-INL})$$
$$\frac{t_1 \longrightarrow t'_1}{\text{inr } t_1 \longrightarrow \text{inr } t'_1} \quad (\text{E-INR})$$

Disjoint Sums (ctd)

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

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

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \quad \Gamma, x_1 : T_1 \vdash t_1 : T \quad \Gamma, x_2 : T_2 \vdash t_2 : T}{\Gamma \vdash \text{case } t_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 : T} \quad (\text{T-CASE})$$

NOTE: Uniqueness of types is lost!

Recovering Uniqueness

$t ::= \dots$	<i>terms</i>
$\text{inl } t \text{ as } T$	<i>tagging (left)</i>
$\text{inr } t \text{ as } T$	<i>tagging (right)</i>
$v ::= \dots$	<i>values</i>
$\text{inl } v \text{ as } T$	<i>tagged value (left)</i>
$\text{inr } v \text{ as } T$	<i>tagged value (right)</i>

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

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

Evaluation rules can just ignore the type annotations

Variants

- Sums can be generalized easily to variants...

$t ::= \dots$	<i>terms</i>
$\langle l=t \rangle \text{ as } T$	<i>tagging</i>
$\text{case } t \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i \quad i \in 1..n$	<i>case</i>
$T ::= \dots$	<i>types</i>
$\langle l_i : T_i \quad i \in 1..n \rangle$	<i>type of variants</i>

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash \langle l_j=t_j \rangle \text{ as } \langle l_i : T_i \quad i \in 1..n \rangle : \langle l_i : T_i \quad i \in 1..n \rangle} \text{ (T-VARIANT)}$$

$$\frac{\begin{array}{c} \Gamma \vdash t_0 : \langle l_i : T_i \quad i \in 1..n \rangle \\ \text{for each } i \quad \Gamma, x_i : T_i \vdash t_i : T \end{array}}{\Gamma \vdash \text{case } t_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i \quad i \in 1..n : T} \text{ (T-CASE)}$$

Variants (ctd)

$$\text{case } \langle l_j = v_j \rangle \text{ as } T \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{i \in 1..n} \quad (\text{E-CASEVARIANT})$$
$$\longrightarrow [x_j \mapsto v_j] t_j$$
$$\frac{t_0 \longrightarrow t'_0}{\text{case } t_0 \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{i \in 1..n} \longrightarrow \text{case } t'_0 \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{i \in 1..n}} \quad (\text{E-CASE})$$
$$\frac{t_i \longrightarrow t'_i}{\langle l_i = t_i \rangle \text{ as } T \longrightarrow \langle l_i = t'_i \rangle \text{ as } T} \quad (\text{E-VARIANT})$$

Options and Enumerations

- Optional values of type T can be encoded as
 - Disjoint sums of Unit and T
- Enumerated types can be simulated as
 - Variant that includes Unit for each enumerated value

```
Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit,  
          thursday:Unit, friday:Unit>;
```

```
nextBusinessDay = λw:Weekday.
```

```
  case w of <monday=x>    ⇒ <tuesday=unit> as Weekday  
           | <tuesday=x>  ⇒ <wednesday=unit> as Weekday  
           | <wednesday=x> ⇒ <thursday=unit> as Weekday  
           | <thursday=x> ⇒ <friday=unit> as Weekday  
           | <friday=x>   ⇒ <monday=unit> as Weekday;
```

General Recursion

- In none of the typed versions of the lambda-calculus that we have seen up to now is it possible to write a non-terminating program
- In particular, the fix combinator from the untyped lambda calculus is not typable
- But we can re-introduce fix as a typed primitive...

General Recursion

$t ::= \dots$ *terms*
 $\text{fix } t$ *fixed point of } t*

$$\text{fix } (\lambda x:T_1.t_2) \longrightarrow [x \mapsto (\text{fix } (\lambda x:T_1.t_2))]t_2 \quad (\text{E-FIXBETA})$$
$$\frac{t_1 \longrightarrow t'_1}{\text{fix } t_1 \longrightarrow \text{fix } t'_1} \quad (\text{E-FIX})$$
$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash \text{fix } t_1 : T_1} \quad (\text{T-FIX})$$
$$\text{letrec } x:T_1=t_1 \text{ in } t_2 \stackrel{\text{def}}{=} \text{let } x = \text{fix } (\lambda x:T_1.t_1) \text{ in } t_2$$

Example

```
ff = λie:Nat→Bool.  
    λx:Nat.  
      if iszero x then true  
      else if iszero (pred x) then false  
      else ie (pred (pred x));  
  
iseven = fix ff;  
  
iseven 7;
```

```
letrec 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;
```

References and Assignment

References and Assignments

- Dealing with references and assignment will be a significantly more complex extension
- In particular because we can no longer model the state of an abstract machine as just the program text
 - We need an additional component that models the heap or the store
 - As a consequence, the typing relation will need to track information about the store

Basic Operations

- Allocation:
 - `r = ref 5; // allocates and initializes a memory cell`
- Dereferencing:
 - `!r; // reads the contents of cell r`
- Assignment:
 - `r := 7; // stores a new value in cell r`
- NOTE: in many languages, dereferencing and some kinds of allocation are implicit

Complications

- Evaluation of terms has “side-effects”
 - $r := \text{succ}(!r);$
- Many *aliases* to the same cell can exist:
 - $r = \text{ref } 5; s = r;$
 - $s := 7; !r;$
- Garbage collection or deallocation
 - We will not model deallocation and hence assume “transparent” garbage collection at run-time.

Syntax and Typing

$t ::=$

`unit`

`x`

`$\lambda x:T.t$`

`t t`

`ref t`

`!t`

`t:=t`

terms

unit constant

variable

abstraction

application

reference creation

dereference

assignment

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{ref } t_1 : \text{Ref } T_1} \quad (\text{T-REF})$$

$$\frac{\Gamma \vdash t_1 : \text{Ref } T_1}{\Gamma \vdash !t_1 : T_1} \quad (\text{T-DEREF})$$

$$\frac{\Gamma \vdash t_1 : \text{Ref } T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \text{Unit}} \quad (\text{T-ASSIGN})$$

Evaluation

- To model allocation, we need to extend the state of our abstract machines
 - With a *store* or *heap*
 - We model the store as a finite function from *locations* to *values*
 - Allocation adds a new location to the domain of the function – this new location is the result of the ref operator
 - Dereference reads the function
 - Assignment changes (updates) the function for one location

Extended language to model intermediary machine states

<code>v ::=</code>	<i>values</i>
<code>unit</code>	<i>unit constant</i>
<code>$\lambda x:T.t$</code>	<i>abstraction value</i>
<code>/</code>	<i>store location</i>

<code>t ::=</code>	<i>terms</i>
<code>unit</code>	<i>unit constant</i>
<code>x</code>	<i>variable</i>
<code>$\lambda x:T.t$</code>	<i>abstraction</i>
<code>t t</code>	<i>application</i>
<code>ref t</code>	<i>reference creation</i>
<code>!t</code>	<i>dereference</i>
<code>t:=t</code>	<i>assignment</i>
<code>/</code>	<i>store location</i>

Evaluation Rules

$$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{t_1 := t_2 \mid \mu \longrightarrow t'_1 := t_2 \mid \mu'} \quad (\text{E-ASSIGN1})$$

$$\frac{t_2 \mid \mu \longrightarrow t'_2 \mid \mu'}{v_1 := t_2 \mid \mu \longrightarrow v_1 := t'_2 \mid \mu'} \quad (\text{E-ASSIGN2})$$

$$l := v_2 \mid \mu \longrightarrow \text{unit} \mid [l \mapsto v_2]\mu \quad (\text{E-ASSIGN})$$

$$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{\text{ref } t_1 \mid \mu \longrightarrow \text{ref } t'_1 \mid \mu'} \quad (\text{E-REF})$$

$$\frac{l \notin \text{dom}(\mu)}{\text{ref } v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \quad (\text{E-REFV})$$

Evaluation Rules (ctd)

$$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{!t_1 \mid \mu \longrightarrow !t'_1 \mid \mu'} \quad (\text{E-DEREF})$$

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \quad (\text{E-DEREFLOC})$$

$$\frac{t_1 \mid \mu \longrightarrow t'_1 \mid \mu'}{t_1 \ t_2 \mid \mu \longrightarrow t'_1 \ t_2 \mid \mu'} \quad (\text{E-APP1})$$

$$\frac{t_2 \mid \mu \longrightarrow t'_2 \mid \mu'}{v_1 \ t_2 \mid \mu \longrightarrow v_1 \ t'_2 \mid \mu'} \quad (\text{E-APP2})$$

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

Typing

- Since we extended the syntax to model intermediary states, we also need to extend the notion of well-typed state
- We introduce the notion of “store typing” (Σ) to remember what types are stored at what locations
- The proof of Preservation will show how to maintain store typings during evaluation
 - They are empty when the program starts(!)

Typing Rules

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \text{Ref } T_1} \quad (\text{T-LOC})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash \text{ref } t_1 : \text{Ref } T_1} \quad (\text{T-REF})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} \quad (\text{T-DEREF})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}} \quad (\text{T-ASSIGN})$$

Preservation

A store μ is said to be *well typed* with respect to a typing context Γ and a store typing Σ , written $\Gamma \mid \Sigma \vdash \mu$, if $\text{dom}(\mu) = \text{dom}(\Sigma)$ and $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in \text{dom}(\mu)$.

Theorem: If

$$\Gamma \mid \Sigma \vdash \mathfrak{t} : \mathsf{T}$$

$$\Gamma \mid \Sigma \vdash \mu$$

$$\mathfrak{t} \mid \mu \longrightarrow \mathfrak{t}' \mid \mu'$$

then, for **some** $\Sigma' \supseteq \Sigma$,

$$\Gamma \mid \Sigma' \vdash \mathfrak{t}' : \mathsf{T}$$

$$\Gamma \mid \Sigma' \vdash \mu'.$$

Progress

Theorem: Suppose t is a closed, well-typed term (that is, $\emptyset \mid \Sigma \vdash t : T$ for some T and Σ). Then either t is a value or else, for any store μ such that $\emptyset \mid \Sigma \vdash \mu$, there is some term t' and store μ' with $t \mid \mu \longrightarrow t' \mid \mu'$.

Proofs of Progress and Preservation are relatively straightforward

Exceptions

Non-catchable exceptions

$t ::= \dots$ *terms*
 error *run-time error*

$\text{error } t_2 \longrightarrow \text{error}$ (E-APPERR1)

$v_1 \text{ error} \longrightarrow \text{error}$ (E-APPERR2)

$\Gamma \vdash \text{error} : T$ (T-ERROR)

NOTE: Types are no longer unique!

THEOREM [PROGRESS]: *Suppose t is a closed, well-typed normal form. Then either t is a value or $t = \text{error}$.*

Handling Exceptions

$t ::= \dots$
 $\text{try } t \text{ with } t$

Evaluation

terms

trap errors

$\text{try } v_1 \text{ with } t_2 \longrightarrow v_1$ (E-TRYV)

$\text{try error with } t_2 \longrightarrow t_2$ (E-TRYERROR)

$$\frac{t_1 \longrightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t'_1 \text{ with } t_2}$$
 (E-TRY)

Typing

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T}$$
 (T-TRY)

Exceptions carrying values

$t ::= \dots$	<i>terms</i>
<code>raise t</code>	<i>raise exception</i>

$(\text{raise } v_{11}) t_2 \longrightarrow \text{raise } v_{11}$ (E-APPRaise1)

$v_1 (\text{raise } v_{21}) \longrightarrow \text{raise } v_{21}$ (E-APPRaise2)

$$\frac{t_1 \longrightarrow t'_1}{\text{raise } t_1 \longrightarrow \text{raise } t'_1}$$
 (E-RAISE)

$\text{raise } (\text{raise } v_{11}) \longrightarrow \text{raise } v_{11}$ (E-RAISERAISE)

$\text{try } v_1 \text{ with } t_2 \longrightarrow v_1$ (E-TRYV)

$\text{try } \text{raise } v_{11} \text{ with } t_2 \longrightarrow t_2 v_{11}$ (E-TRYRAISE)

$$\frac{t_1 \longrightarrow t'_1}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t'_1 \text{ with } t_2}$$
 (E-TRY)

Typing Rules

$$\frac{\Gamma \vdash t_1 : T_{\text{exn}}}{\Gamma \vdash \text{raise } t_1 : T} \quad (\text{T-EXN})$$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T_{\text{exn}} \rightarrow T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T} \quad (\text{T-TRY})$$

- We can choose several interesting T_{exn}
 - Natural numbers (error numbers)
 - Strings
 - Variants or extensible variants

Wrap up

- Our techniques can deal with:
 - Various base types
 - The Unit type and sequential composition
 - Type ascriptions
 - Let bindings
 - Pairs, tuples, records, variants
 - Recursion
 - References and assignment
 - Exceptions