`x = 1`

Resources: The slides of this lecture were derived from [Järvi], with permission of the original author, by copy & paste or by selection, annotation, or rewording. [Järvi] is in turn based on [Pierce] as the underlying textbook.

`let x = 1 in ...`

`x(1).`

`!x(1)`

`x.set(1)`

Programming Language Theory

The Simply Typed Lambda Calculus

Ralf Lämmel

[Järvi] Slides by J. Järvi: "Programming Languages", CPSC 604 @TAMU (2009)
[Pierce] B.C. Pierce: Types and Programming Languages, MIT Press, 2002

Towards typed lambda calculus

- Now suppose you want to distinguish values of different types:
  - Booleans
  - Numbers
  - **Functions on Booleans**
  - **Functions on functions on Booleans**
  - Products, sums of ...
  - ...
- These types need to be ...
  - specified in the program, and
  - checked to be correct.

Setting up the simply typed lambda calculus

- Define syntax for simple types and function types.
- Extend lambda abstractions for explicit types.
- Define typing rules.
- Revise reduction semantics.
- Establish type safety.
- Consider extensions.

Revised lambda abstraction

- Lambda abstractions are annotated with types:

  $\lambda x : T.t$

- Grammar of types:

  $T ::=$ **bool**

  nat

  $T \rightarrow T$

We only consider these simple types here for simplicity.

Examples

- What are the types of these terms?
  - $\lambda x : bool.x$
  - $\lambda f : bool \rightarrow bool.f\ x$

Note that lambda variables are typed explicitly.

- Here are the same terms for the untyped calculus:
  - $\lambda x\ .x$
  - $\lambda f.f\ x$

Meaningless terms

- Some terms diverge.
- Some applications are ill-typed, e.g.:

  $(\lambda f : bool \rightarrow bool.f\ x)\ true$

- Goal: a type system to reject ill-typed terms.

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Typing relation with context

$\Gamma \vdash t : T$ *Term t has type T in the typing context Γ*

- A typing context is a sequence of bindings.
- Each binding is a variable-type pair, e.g.: $x : T$.
- Contexts are composed as in $\Gamma, x : T$.
- All variable names are distinct for a given Γ.
- Γ can be omitted if it is empty.
- Γ can be empty for closed terms.

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Typing rules for simply-typed lambda calculus

- x, y, z, f, g range over variables
- s, t, u range over terms
- S, T, U range over types

T-Variable

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

T-Abstraction

$$\frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T.u : T \rightarrow U}$$

T-Application

$$\frac{\Gamma \vdash t : U \rightarrow T \qquad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T}$$

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Rules for **bool**

T-True

$\vdash$ `true` : `bool`

T-False

$\vdash$ `false` : `bool`

These typing rules illustrate one option to add specific types and their operations to a basic lambda calculus. Basically**, we need to add one rule per operation.**

Typing derivations

Construct derivations as proofs of terms having a certain type.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{f : \texttt{bool} \to \texttt{bool} \in f : \texttt{bool} \to \texttt{bool}}{f : \texttt{bool} \to \texttt{bool} \vdash f : \texttt{bool} \to \texttt{bool}}
      \qquad
      f : \texttt{bool} \to \texttt{bool} \vdash \texttt{false} : \texttt{bool}
    }{f : \texttt{bool} \to \texttt{bool} \vdash f\ \texttt{false} : \texttt{bool}}
  }{\vdash (\lambda f : \texttt{bool} \to \texttt{bool}.f\ \texttt{false}) : (\texttt{bool} \to \texttt{bool}) \to \texttt{bool}}
  \qquad
  \dfrac{
    \dfrac{g : \texttt{bool} \in g : \texttt{bool}}{g : \texttt{bool} \vdash g : \texttt{bool}}
  }{\vdash \lambda g : \texttt{bool}.g : \texttt{bool} \to \texttt{bool}}
}{\vdash (\lambda f : \texttt{bool} \to \texttt{bool}.f\ \texttt{false})\ \lambda g : \texttt{bool}.g : \texttt{bool}}
$$

T-Variable

$$\dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

T-Abstraction

$$\dfrac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T.u : T \to U}$$

T-Application

$$\dfrac{\Gamma \vdash t : U \to T \qquad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T}$$

Evaluation rules

- Syntax (terms, values, types)

$$t ::= x \mid v \mid t\ t$$
$$v ::= \lambda x : T.t \mid \texttt{true} \mid \texttt{false}$$
$$T ::= \texttt{bool} \mid T \rightarrow T$$

- Evaluation rules

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}$$

$$(\lambda x : T.t)\ v \longrightarrow [v/x]t$$

Evaluation rules do not bother with types.

Type safety $=$ progress + preservation

Progress: If t is a closed, well-typed term, then either t is a value, or there exists some u, such that $t \longrightarrow u$.

Preservation: If $\Gamma \vdash t : T$ and $t \longrightarrow u$, then $\Gamma \vdash u : T$

Requires several trivial lemmas (properties) that are omitted here.

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

A few extensions

- *Recursion* (fixed point combinator)
- *Unit type and sequencing* (for effects eventually)
- *Type annotation* (for documentation, abstraction)
- *Pairs* (as a simple form of type construction)
- *Lists* (another example of type construction)
- *Records* (as a first step towards objects)

Recursion

- A fixed point combinator is definable in the untyped calculus.
- It is not definable in the simply typed version.
  - A special combinator is added to the formal system.
  - Alternatively, a more powerful type system is needed.

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Recursion in the presence of types

- Self application is not typeable:

  $\lambda x : ? \, . \, x \; x$

- Y is not typeable either.
- Solution: add a primitive **fix**.

  $t ::= \ldots \mid \mathit{fix}\; t$

Typing rule

$$\frac{\Gamma \vdash t : T \rightarrow T}{\Gamma \vdash \texttt{fix}\; t : T}$$

Evaluation rules

$$\frac{t \rightarrow t'}{\texttt{fix}\; t \rightarrow \texttt{fix}\; t'}$$

$$\texttt{fix}\; (\lambda x : T.t) \rightarrow [(\texttt{fix}\; (\lambda x : T.t))/x]\; t$$

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Illustration of fix

iseven : nat → bool

iseven = fix g

g: (nat→ bool)→ nat→ bool

g = **λ** *e:nat → bool.* **λ** *x:nat.*

if iszero x then true

else if iszero (pred x) then false

else e (pred (pred x))

g is a generator for the *iseven* function. Given a function that equates with *iseven* for numbers up to *n*, g defines an approximation up to *n* + 2. *fix g* extends this to all *n*.

Unit type and sequencing

- New syntax: $t ::= \ldots\ \texttt{unit} \mid t;\ t$
- New value: $v ::= \ldots\ \texttt{unit}$
- New type: $T ::= \ldots\ \texttt{unit}$
- Typing of unit and sequencing:

$$\Gamma \vdash \texttt{unit} : \texttt{unit} \qquad \frac{\Gamma \vdash t : \texttt{unit} \qquad \Gamma \vdash u : U}{\Gamma \vdash t;\ u : U}$$

- Evaluation of sequencing:

$$\frac{t \longrightarrow u}{t;\ s \longrightarrow u;\ s} \qquad \texttt{unit};\ u \longrightarrow u$$

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Type annotation (ascription)

- Syntax:

  t ::= ... | t as T

- Typing rule:

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \texttt{ as } T : T}$$

- Evaluation rules:

$$\frac{t \longrightarrow u}{t \texttt{ as } T \longrightarrow u \texttt{ as } T}$$

$$v \texttt{ as } T \longrightarrow v$$

Pairs

New syntax: $t ::= \ldots \{t, t\} \mid t.1 \mid t.2$
New types: $T ::= \ldots T \times T$

- Typing of pairs:

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash u : U}{\Gamma \vdash \{t, u\} : T \times U} \qquad \frac{\Gamma \vdash t : T \times U}{\Gamma \vdash t.1 : T} \qquad \frac{\Gamma \vdash t : T \times U}{\Gamma \vdash t.2 : U}$$

- Evaluation rules:

$$\{v_1, v_2\}.1 \rightarrow v_1 \qquad \{v_1, v_2\}.2 \rightarrow v_2 \qquad \frac{t \rightarrow t'}{\{t, u\} \rightarrow \{t', u\}}$$

$$\frac{u \rightarrow u'}{\{v, u\} \rightarrow \{v, u'\}} \qquad \frac{t \rightarrow t'}{t.1 \rightarrow t'.1} \qquad \frac{t \rightarrow t'}{t.2 \rightarrow t'.2}$$

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Lists

- New type: ... | *List T*

- New syntax: ... | *nil*[*T*] | *cons*[*T*] *t t* | *isnil*[*T*] *t* | *head*[*T*] *t* | *tail*[*T*] *t*

- New congruence rules, e.g.: $$\frac{t_1 \to t_1'}{\texttt{cons}[T]\ t_1\ t_2 \to \texttt{cons}[T]\ t_1'\ t_2}$$

- New computation rules, e.g.: $\texttt{head}[S]\ (\texttt{cons}[T]\ v_1\ v_2) \to v_1$

- New typing rules, e.g.: $$\frac{\Gamma \vdash t : \texttt{List}\ T}{\Gamma \vdash \texttt{head}[T]\ t : T}$$

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Records

- Pairs generalize to tuples.
- Tuples further generalize to records.
- Records generalize to extensible records.
- Extensible records generalize to objects.

We use the syntax:

```
{age=44, name="Smith"}              // record value
{age=44, name="Smith"}.name         // field access
```

and write the types as:

```
{age=44, name="Smith"} : {age:Int, name:String}
```

Records

- New syntax: $t ::= \ldots \ \{l_i = t_i{}^{i \in 1 \ldots n}\} \mid t.l$
- New values: $v ::= \ldots \ \{l_i = v_i{}^{i \in 1 \ldots n}\}$
- New types: $T ::= \ldots \ \{l_i : T_i{}^{i \in 1 \ldots n}\}$
- Typing of records:

$$\frac{\text{for each } i,\ \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i{}^{i \in 1 \ldots n}\} : \{l_i : T_i{}^{i \in 1 \ldots n}\}} \qquad \frac{\Gamma \vdash t : \{l_i : T_i{}^{i \in 1 \ldots n}\}}{\Gamma \vdash t.l_j : T_j}$$

- Evaluation rules:

$$\{l_i = v_i{}^{i \in 1 \ldots n}\}.l_j \longrightarrow v_j \qquad \frac{t \longrightarrow t'}{t.l \longrightarrow t'.l}$$

$$\frac{t_j \longrightarrow t_j{}'}{\begin{array}{l}\{l_i = v_i{}^{i \in 1 \ldots j-1}, l_j = t_j, l_k = t_k{}^{k \in j+1 \ldots n}\} \\ \quad \longrightarrow \{l_i = v_i{}^{i \in 1 \ldots j-1}, l_j = t_j{}', l_k = t_k{}^{k \in j+1 \ldots n}\}\end{array}}$$

Prolog as a sandbox for semantics of lambda calculi

Typed NB

https://slps.svn.sourceforge.net/svnroot/slps/topics/semantics/nb/

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Types in NB

`T`	::=		types:
		`Bool`	the Boolean type
		`Nat`	the type of numeric values

NB typing rules

T-True

$\texttt{true} : \texttt{Bool}$

T-False

$\texttt{false} : \texttt{Bool}$

T-If

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T}$$

T-Zero

$\texttt{0} : \texttt{Nat}$

T-Succ

$$\frac{t : \texttt{Nat}}{\texttt{succ } t : \texttt{Nat}}$$

T-Pred

$$\frac{t : \texttt{Nat}}{\texttt{pred } t : \texttt{Nat}}$$

T-Iszero

$$\frac{t : \texttt{Nat}}{\texttt{iszero } t : \texttt{Bool}}$$

NB typing rules

Prolog

1:1 mapping

```
welltyped(true,bool).
welltyped(false,bool).
welltyped(zero,nat).
welltyped(succ(T),nat) :- welltyped(T,nat).
welltyped(pred(T),nat) :- welltyped(T,nat).
welltyped(iszero(T),bool) :- welltyped(T,nat).
welltyped(if(T1,T2,T3),T) :-
 welltyped(T1,bool),
 welltyped(T2,T),
 welltyped(T3,T).
```

Conditional evaluation for typed NB

Prolog

```
main(Input)
 :-
    see(Input), read(Term), seen,
    format('Input term: ~w~n',[Term]),
    welltyped(Term,Type),
    format('Type of term: ~w~n',[Type]),
    manysteps(Term,X),
    show(X,Y),
    format('Value of term: ~w~n',[Y]).
```

An applied, typed lambda calculus

https://slps.svn.sourceforge.net/svnroot/slps/topics/semantics/lambda/

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

Revised syntax

- Lambda abstractions are annotated with types:

  $\lambda x : T.t$

- Grammar of types:

  $T ::= \quad$ **bool**

  $\qquad\quad$ **nat**

  $\qquad\quad T \rightarrow T$

Revised syntax

Prolog

Build on top of untyped applied lambda calculus.

The untyped version is no longer to be used.

```
:- ['../applied/term.pro'].
:- ['../applied/value.pro'].

term(lam(X,A,T)) :- variable(X), type(A), term(T).
term(fix(T)) :- term(T).

value(lam(X,A,T)) :- variable(X), type(A), term(T).

type(bool).
type(nat).
type(fun(A1,A2)) :- type(A1), type(A2).
```

Typing rules for simply-typed lambda calculus

- x, y, z, f, g range over variables
- s, t, u range over terms
- S, T, U range over types

T-Variable
$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

T-Abstraction
$$\frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T.u : T \to U}$$

T-Application
$$\frac{\Gamma \vdash t : U \to T \qquad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T}$$

Typing rules for simply-typed lambda calculus

Prolog

```
:- ensure_loaded('../../shared/map.pro').

welltyped(G,var(X),A)
 :-
    member((X,A),G).

welltyped(G,app(T1,T2),B)
 :-
    welltyped(G,T1,fun(A,B)),
    welltyped(G,T2,A).

welltyped(G1,lam(X,A,T),fun(A,B))
 :-
    update(G1,X,A,G2),
    welltyped(G2,T,B).
```

Prolog

Lifted typing rules of NB

```
welltyped(T,A) :- welltyped([],T,A).

welltyped(_,true,bool).
welltyped(_,false,bool).
welltyped(_,zero,nat).
welltyped(G,succ(T),nat) :- welltyped(G,T,nat).
welltyped(G,pred(T),nat) :- welltyped(G,T,nat).
welltyped(G,iszero(T),bool) :- welltyped(G,T,nat).
welltyped(G,if(T1,T2,T3),T) :-
 welltyped(G,T1,bool),
 welltyped(G,T2,T),
 welltyped(G,T3,T).
```

We had to ***rewrite*** the typing rules for NB to *incorporate the typing context*.

Small-step semantics

$$\frac{t_1 \longrightarrow t_1{}'}{t_1\ t_2 \longrightarrow t_1{}'\ t_2} \qquad \frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}$$

$$(\lambda x : T.t)\ v \longrightarrow [v/x]t$$

Types play no role in the semantics.

Update on evaluation rules

Prolog

Build on top of untyped applied lambda calculus.

```
:- ['../applied/eval.pro'].
```

"Type erasure" for lambdas

```
eval(lam(X,_,T),lam(X,T)).
substitute(N,X,lam(Y,_,M),T) :- substitute(N,X,lam(Y,M),T).
freevars(lam(X,_,M),FV) :- freevars(lam(X,M),FV).
```

This slide is derived from Jaakko Järvi's slides for his course "Programming Languages", CPSC 604 @ TAMU.

The fix construct

Since fixed point combinators are not typeable in this calculus, we need fix instead.

Syntax **fix**.

$t ::= \ldots \mid \mathit{fix}\ t$

Typing rule

$$\frac{\Gamma \vdash t : T \rightarrow T}{\Gamma \vdash \texttt{fix}\ t : T}$$

Evaluation rules

$$\frac{t \longrightarrow t'}{\texttt{fix}\ t \longrightarrow \texttt{fix}\ t'}$$

$$\texttt{fix}\ (\lambda x : T.t) \longrightarrow [(\texttt{fix}\ (\lambda x : T.t))/x]\ t$$

The fix construct

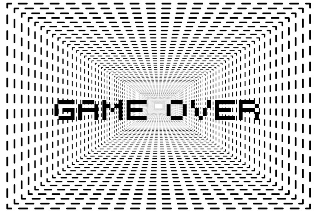
Prolog

Evaluation rules

```
eval(fix(T1),fix(T2)) :- eval(T1,T2).
eval(fix(lam(X,T1)),T2) :- substitute(fix(lam(X,T1)),X,T1,T2).
substitute(N,X,fix(T1),fix(T2)) :- substitute(N,X,T1,T2).
freevars(fix(T),FV) :- freevars(T,FV).
```

Typing rule

```
welltyped(G,fix(T),A)
 :-
   welltyped(G,T,fun(A,A)).
```



- **Summary**: *The typed lambda calculus*
  - *Typing relation carries argument for context.*
  - *Many forms of types can be added modularly.*
  - *Recursion requires built-in Y combinator.*
- **Prepping**: *"Types and Programming Languages"*
  - *Chapters 7 and 11*
- **Outlook**:
  - Lambda calculi with polymorphism
  - Process calculi
  - Object calculi