Typed λ-Calculus and Logic

April 17, 2002

CS 131: Programming Languages

Pure Simply-Typed λ-Calculus

- Syntax

```
M,N ::=  x              variables
      |  λx:t.M         functions
      |  M N            applications

t,u ::=  α1 | α2 | ...  type variables
      |  t -> u         function types
```

One-step β-Reduction

- The relation $\to_\beta$ is defined by:

$$\frac{}{(\lambda x{:}t.M)N \to_\beta M[x \to N]}$$

$$\frac{M \to_\beta M'}{M\ N \to_\beta M'\ N} \qquad \frac{N \to_\beta N'}{M\ N \to_\beta M\ N'}$$

$$\frac{M \to_\beta M'}{\lambda x{:}t.M \to_\beta \lambda x{:}t.M'}$$

Extension: Adding Pairs

- Syntax as in NQSML

```
M,N ::=  ...
      |  <M,N>              pairs
      |  fst M | snd M      projections
t,u ::=  ...
      |  t * u              pair types
```

$$\frac{}{\texttt{fst}\ \langle M,N\rangle \to M} \qquad \frac{}{\texttt{snd}\ \langle M,N\rangle \to N}$$

Summary: Static Semantics

$$\frac{}{\ldots, \mathtt{x{:}t}, \ldots \vdash \mathtt{x : t}}$$

$$\frac{\Gamma, \mathtt{x{:}t} \vdash \mathtt{M : u}}{\Gamma \vdash \mathtt{(\lambda x{:}t.M) : t\text{->}u}} \qquad \frac{\Gamma \vdash \mathtt{M : t\text{->}u} \quad \Gamma \vdash \mathtt{N : t}}{\Gamma \vdash \mathtt{M\ N : u}}$$

$$\frac{\Gamma \vdash \mathtt{M : t} \quad \Gamma \vdash \mathtt{N : u}}{\Gamma \vdash \mathtt{<M,N> : t*u}}$$

$$\frac{\Gamma \vdash \mathtt{M : t*u}}{\Gamma \vdash \mathtt{fst\ M : t}} \qquad \frac{\Gamma \vdash \mathtt{M : t*u}}{\Gamma \vdash \mathtt{snd\ M : u}}$$

Erasing All but the Types

$$\frac{}{\ldots, \mathtt{t}, \ldots \vdash \mathtt{t}}$$

$$\frac{\Gamma, \mathtt{t} \vdash \mathtt{u}}{\Gamma \vdash \mathtt{t\text{->}u}} \qquad \frac{\Gamma \vdash \mathtt{t\text{->}u} \quad \Gamma \vdash \mathtt{t}}{\Gamma \vdash \mathtt{u}}$$

$$\frac{\Gamma \vdash \mathtt{t} \quad \Gamma \vdash \mathtt{u}}{\Gamma \vdash \mathtt{t*u}}$$

$$\frac{\Gamma \vdash \mathtt{t*u}}{\Gamma \vdash \mathtt{t}} \qquad \frac{\Gamma \vdash \mathtt{t*u}}{\Gamma \vdash \mathtt{u}}$$

Rules for Propositional Logic

$$\frac{}{\ldots, p, \ldots \vdash p}$$

$$\frac{\Gamma, p \vdash q}{\Gamma \vdash p \Rightarrow q} \qquad \frac{\Gamma \vdash p \Rightarrow q \quad \Gamma \vdash p}{\Gamma \vdash q}$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \wedge q}$$

$$\frac{\Gamma \vdash p \wedge q}{\Gamma \vdash p} \qquad \frac{\Gamma \vdash p \wedge q}{\Gamma \vdash q}$$

Curry-Howard Isomorphism

- a.k.a. "Proofs as programs", "Propositions as types"
- Every type corresponds to a logical proposition
  - Type variables correspond to propositional variables
  - Function types correspond to implications
  - Pair types correspond to conjunctions
- A proposition is provable if and only if there is a term of the corresponding type
  - Such types are said to be *inhabited*.
  - Typed λ-terms are encodings of proofs.

Examples

1. Show that

   $$\vdash p \Rightarrow (p \wedge p)$$

   by finding a term of type

   $$\alpha \texttt{ -> } (\alpha * \alpha)$$

2. Show that

   $$\vdash (p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \wedge q) \Rightarrow r))$$

   by finding a term of type

   $$(\alpha \texttt{->} (\beta \texttt{->} \gamma)) \texttt{ -> } ((\alpha * \beta) \texttt{->} \gamma)$$

Extensions

- The `true` proposition corresponds to any non-empty type
  - E.g., `unit`.
- The `false` proposition corresponds to an empty type.
  - Usually called `void`.
  - Encode $\neg p$ as $(p \Rightarrow$ `false`$)$.
- Second-order predicate calculus: polymorphic types
  - Second-order = quantifying over *propositions*.
  - E.g., $\forall\alpha.\ \alpha \rightarrow (\alpha * \alpha)$ vs. $\forall p.\ p \rightarrow (p \wedge p)$
- Disjunctions: sum types
- Propositional logic: dependent types
- Modal logic: run-time code generation
- Linear logic: linear types

Intuitionism

- Generally, λ-calculi correspond to *constructive* or *intuitionistic* logics.
  - E.g., no terms of type

$\forall\alpha.\ \alpha$ + (α->void)	*($\forall p.\ p$ or $\neg p$)*
$\forall\alpha.$ ((α->void)->void) -> α	*($\forall p.\ \neg\neg p \Rightarrow p$)*
$\forall\alpha.\forall\beta.$(($\alpha$->$\beta$)->$\alpha$) -> α	*(Pierce's Law)*

Proof Normalization

- Reductions as proof "simplifications".

$$\frac{\dfrac{\Gamma \vdash \mathtt{M} : \mathtt{t} \qquad \Gamma \vdash \mathtt{N} : \mathtt{u}}{\Gamma \vdash \langle \mathtt{M},\mathtt{N} \rangle : \mathtt{t*u}}}{\Gamma \vdash \mathtt{fst}\ \langle \mathtt{M},\mathtt{N} \rangle : \mathtt{t}} \qquad\qquad \Gamma \vdash \mathtt{M} : \mathtt{t}$$

$$\frac{\dfrac{\Gamma \vdash \mathtt{t} \qquad \Gamma \vdash \mathtt{u}}{\Gamma \vdash \mathtt{t} \wedge \mathtt{u}}}{\Gamma \vdash \mathtt{t}} \qquad\qquad \Gamma \vdash \mathtt{t}$$

Hilbert System for Implication

Axiom Schema

$$\frac{}{p \Rightarrow (q \Rightarrow p)}$$

$$\frac{}{(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))}$$

Modus Ponens

$$\frac{p \Rightarrow q \quad p}{q}$$

Summary

λ-Calculus		Logic
Type	⟷	Proposition
Term (program)	⟷	Proof
Reduction	⟷	Proof Normalization

Logical Frameworks

- A *deductive system* is a specification containing axioms and rules of inference.
  - Examples:
    - Logics
    - Mathematical theories
    - Syntax definitions
    - Typing rules
    - Dynamic semantics
- A *logical framework* is a language for formally specifying deductive systems.

Automath

- 1960's-70's: de Bruijn creates Automath system(s) for formalizing mathematics
  - Goals:
    - Mechanical checking of mathematical proofs
    - Simple, very general framework
      - Not committed to any specific mathematical theory
      - Proof-checking easy because all steps are small.
  - Ph.D. student translated and mechanically checked an entire text on foundations of real analysis.
    - Development of the properties of real and complex numbers starting from the Peano axioms.

LF

- The particular logical framework I will concentrate on today
  - Based upon (dependently-typed) λ-calculus
    - So we can use β-conversion whenever we want
  - Implemented by a system called Elf

Representing Object Languages

- The axioms and rules of inference are about things in some domain
  - e.g., logical propositions, program expressions, types, etc.
  - The domain of whatever problem we care about is called the *object language*.
- To formalize a deductive system we need to first describe the object language.

Example: Predicate Logic

```
o : type.  % type of propositions
i : type.  % type of individuals

true  : o.
false : o.
and   : o -> o -> o.
or    : o -> o -> o.
imp   : o -> o -> o.
not   : o -> o.

zero  : i.
succ  : i -> i.
even  : i -> o.
```

$\ulcorner (\neg \text{true}) \Rightarrow (\text{even}(\text{successor}(0))) \urcorner$ =
`imp (not true) (even (succ zero)) : o`

What About Variables?

- What about free or bound variables in the object language?
  - e.g., quantifiers, function parameters, etc.

- Two main methods
  - Use constants for variables (e.g., strings)
  - Use logical-framework variables and functions
    - Known as HOAS (Higher-Order Abstract Syntax)

What About Variables?

- Representing variables as constants:

  ⌜ $\forall x.((\text{even } x) \vee (\text{even}(\text{successor}(x))))$ ⌝ =

  ```
  forall "x" (or (even "x") (even (succ "x")))
  ```

- Advantages:
  - Simple
  - Works better with certain inductive proofs
- Disadvantages:
  - Must also formalize checking for unbound variables, renaming of bound variables, substitution, etc.

What About Variables?

- Representing variables as variables:

  ```
  forall : (i -> o) -> o
  exists : (i -> o) -> o
  ```

  ⌜ $\forall x.((\text{even } x) \vee (\text{even}(\text{successor}(x))))$ ⌝ =

  `forall` (λx`:i.(or (even` x`) (even (succ` x`)))  :  o`

- Advantages
  - Typechecking ensures all variables are bound
  - Can get substitution from function application

Representing Proofs

- Now that we can represent propositions, we would like to be able to formalize the proof rules.

- We can do this by setting up
  - a type of *proofs*
  - a representation of proof trees.

Natural Deduction (First Try)

$$\frac{}{\texttt{true}} \qquad \frac{\texttt{p} \quad \texttt{q}}{\texttt{p}\wedge\texttt{q}} \qquad \frac{\texttt{p}\wedge\texttt{q}}{\texttt{p}} \qquad \frac{\texttt{p}\wedge\texttt{q}}{\texttt{q}}$$

```
nd : type

truei : nd
andi  : nd -> nd -> nd
andel : nd -> nd
ander : nd -> nd
```

$$\frac{\dfrac{\dfrac{}{\texttt{true}} \qquad \dfrac{}{\texttt{true}}}{\texttt{true}\wedge\texttt{true}}}{\texttt{true}}$$

`andel(andi(truei,truei)) : nd`

But `andel(truei) : nd` as well !?

Natural Deduction

- Key idea: Instead of defining a type of proofs, define a *family* of types:

```
nd : o -> type.
```

- For any proposition p, the type nd p will be the type of proofs of p.
  - e.g., `nd(even(zero))` and `nd(even(succ(zero)))`

Natural Deduction

$$\frac{}{\texttt{true}} \qquad \frac{\texttt{p} \quad \texttt{q}}{\texttt{p}\wedge\texttt{q}} \qquad \frac{\texttt{p}\wedge\texttt{q}}{\texttt{p}} \qquad \frac{\texttt{p}\wedge\texttt{q}}{\texttt{q}}$$

```
nd : o -> type.

truei : nd(true).
andi  : nd(A) -> nd(B) -> nd(and A B).
andel : nd(and A B) -> nd(A).
ander : nd(and A B) -> nd(B).
```

`andel(andi(truei,truei)) : nd(true)`

`andel(true)` is no-longer well-typed!

Natural Deduction: Quantifiers

$$\frac{\forall x.\mathrm{p}}{\mathrm{p}[x\to \mathrm{c}]}$$

$\ulcorner\forall x.\mathrm{p}\urcorner$ = `forall`$(\lambda x$`:i.`$\ulcorner \mathrm{p}\urcorner)$

$\ulcorner \mathrm{p}[x\to\mathrm{c}]\urcorner = \ulcorner \mathrm{p}\urcorner[x\to\ulcorner \mathrm{c}\urcorner]$
$\qquad \leftrightarrow_\beta^* \ (\lambda x.\ulcorner \mathrm{p}\urcorner)\ulcorner \mathrm{c}\urcorner$

```
foralle : nd(forall F) -> C:i -> nd(F C)
```

$$\dfrac{\dfrac{D}{\forall \mathrm{x.even(x)}}}{\mathrm{even(1)}}$$

$\ulcorner D\urcorner$: `nd(forall(`λx`:i.even(x)))`

`foralle` $\ulcorner D\urcorner$ `(succ zero) :`
`nd (even (succ zero))`

Applying LF to PL Theory

- Object Language

```
exp : type

Zero : exp
Succ : exp -> exp
Pair : exp -> exp -> exp
Fst  : exp -> exp
Fn   : (exp -> exp) -> exp
App  : exp -> exp -> exp
```

$\ulcorner$ `fn x =>(x,0)` $\urcorner$ =
`Fn(`λx`:exp.Pair(`x`,Zero))` : `exp`

Applying LF to PL Theory

- Value-ness

```
value : exp -> type.

val_z    : value Zero.
val_s    : value A -> value (Succ A).
val_pair : value A -> value B -> value (Pair A B).
```

$$\frac{}{\texttt{0 value}} \qquad \frac{\texttt{v value}}{\texttt{Succ(v) value}} \qquad \frac{\texttt{v1 value} \quad \texttt{v2 value}}{\texttt{(v1,v2) value}}$$

Applying LF to PL Theory

- Evaluation Rules

```
eval : exp -> exp -> type.

eval_z : eval Zero Zero.
eval_s : eval A B -> eval (Succ A) (Succ B).
eval_pair : eval A A' -> eval B B' ->
                 eval (Pair A B) (Pair A' B').
```

$$\frac{}{\texttt{0} \Downarrow \texttt{0}} \qquad \frac{\texttt{e} \Downarrow \texttt{v}}{\texttt{Succ(e)} \Downarrow \texttt{Succ(v)}} \qquad \frac{\texttt{e1} \Downarrow \texttt{v1} \quad \texttt{e2} \Downarrow \texttt{v2}}{\texttt{(e1,e2)} \Downarrow \texttt{(v1,v2)}}$$

Summary

- Logical frameworks like LF permit:
  - Specification of object languages
  - Specification of axioms and inference rules
  - Specification of meta-theorems (not shown)
  - Automatic checking of proofs.
- Key observations:
  - Proofs become terms in LF, a very small language.
  - Proof validity-checking performed by *typechecking* the corresponding LF.
  - LF typechecking is decidable.
  - An LF typechecker is very simple to implement.

Code Safety

- Given a piece of code that somebody else wrote, how do you know whether it's safe to run?
- Here safety can have a variety of meanings, depending on context, such as:
  - Won't crash
  - Won't cause other programs to crash
  - Won't consume too many resources
  - Won't access private data
  - Won't publicize private data
  - Won't erase your hard drive
- Given an arbitrary binary executable, determining safety is invariably an undecidable problem.

Example Application

- Networking
  - Operating system receives many network packets
  - Expensive to notify a processes for each packet
    - Most processes are not interested in all packets
- Packet filters:
  - Programmer-supplied code that scans a packet and decides whether it's of interest or not.
  - Typically runs in the kernel to avoid context-switches
    - Usually allowed to read from packet, and read/write small area of scratch memory.
    - Must not do arbitrary reads/writes to kernel memory.
    - Must execute quickly (in particular, no infinite loops).
  - Given a packet filter, how can the kernel decide if it is trustworthy?

Cryptography

- Don't run a program unless it is digitally signed by somebody you trust.
  - E.g., Microsoft.
- May protect against malicious code, but not against accidental errors.
  - Guarantees you know who to blame (or who to sue) if the program goes wrong.

Sandboxing

- Run-time checks
  - OS Protection: Give program its own "virtual machine"
    - Virtual memory, etc., restricts program accesses
  - Software Fault Isolation: Rewrite binary executables
    - Inserting check any time something bad might happen.
- Example: memory accesses
  - Packet filter is supposed to only read and write a certain set of memory addresses (packet + small scratch memory)
  - SFI: before every load or store instruction, check the address being accessed and abort if it's out of range
- Run-time overhead can be large.

Safe Languages

- The type systems of languages like SML, Modula 3, and Java guarantee certain safety properties for well-typed programs.
  - So, only run programs given in SML source, or Java bytecode, etc.
- Interpreters are slow
  - BSD packet filters use an in-kernel interpreter for a very simple, safe specialized language.
- Compilers are large, complicated programs.
  - Do you trust your SML or Java compiler to be bug-free?

Proof-Carrying Code

- Binary executables are packaged with proof of safety.
  - To see whether the code is trustworthy, just check the proof
    - No invalid steps
    - The proof must be a safety proof for this particular binary
  - Proof-checking is much easier than finding a proof from scratch!
- Advantages:
  - No run-time overhead
  - Very small "trusted computing base"
    - Just the code related to proof-checking
  - Immune to both accidental and malicious errors.

Proof-Carrying Code

- Where do proofs come from?
  - Manual
    - Whoever generates the binary also does some theorem proving
  - Certifying compilers
    - Compilers which automatically generate safety certificate.
    - Easiest if the source language already guarantees safety.
    - Note 1: If the compiler cannot prove safety (i.e., that a particular array access is always in bounds), it always has the alternative of inserting a run-time check that makes its job easier.
    - Note 2: there is no requirement that the compiler be bug-free!

Example: PCC Packet Filters

- Necula and Lee [OSDI 1996]
  - Hand-coded sample packet filters (for speed).
  - Safety proofs generated automatically with a simple theorem prover.
  - Safety proofs encoded as an LF term (!)
    - Compact and easy to validate
    - Kernel just needs an LF typechecker (around 5 pages of C code)
  - Results: by definition, lowest per-packet overhead
    - 5-8x faster than BPF interpreter, 5x faster than Modula-3, 15-25% faster than SFI.
    - Proof-checking overhead at beginning very quickly amortized.

PCC Packet Filters

- What exactly does safety mean here?
  - Every read is from the packet or scratch memory
  - Every write is to scratch memory
  - All branches are forward
  - Certain reserved registers are not modified.
- What can we assume when code starts running?

$$
\begin{array}{l}
\texttt{r1 mod } 2^{32} = \texttt{r1} \\
\wedge\ \texttt{r2 mod } 2^{32} = \texttt{r2} \wedge \texttt{r2} \geq 64 \wedge \\
\wedge\ \texttt{r3 mod } 2^{32} = \texttt{r3} \wedge \\
\wedge\ \forall \texttt{i}.(\texttt{i} \geq 0 \wedge \texttt{i} < \texttt{r2} \wedge (\texttt{i\&7}) = 0) \Rightarrow \texttt{readable(r1+i)} \\
\wedge\ \forall \texttt{j}.(\texttt{j} \geq 0 \wedge \texttt{j} < 16 \wedge (\texttt{i\&7}) = 0) \Rightarrow (\texttt{readable(r3+j)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \texttt{writable(r3+j)}) \\
\wedge\ \forall \texttt{i,j}.(\texttt{i} \geq 0 \wedge \texttt{i} < \texttt{r2} \wedge \texttt{j} \geq 0 \wedge \texttt{j} < 16) \Rightarrow (\texttt{r1+i}) \neq (\texttt{r3+j})
\end{array}
$$