Typed λ-Calculus and Logic

April 17, 2002
CS 131: Programming Languages

Pure Simply-Typed λ-Calculus

• Syntax

\[
M, N ::= x \quad \text{variables} \\
   \mid \lambda x : t . M \quad \text{functions} \\
   \mid M \ N \quad \text{applications}
\]

\[
t, u ::= \alpha_1 \mid \alpha_2 \mid \ldots \quad \text{type variables} \\
   \mid t \rightarrow u \quad \text{function types}
\]

One-step β-Reduction

• The relation \( \rightarrow_\beta \) is defined by:

\[
(\lambda x : t . M) N \rightarrow_\beta M[x \rightarrow N]
\]

\[
M \rightarrow_\beta M' \quad N \rightarrow_\beta N' \quad \frac{M N \rightarrow_\beta M' N'}{M N \rightarrow_\beta M' N'}
\]

\[
\frac{M \rightarrow_\beta M'}{\lambda x : t . M \rightarrow_\beta \lambda x : t . M'}
\]

Extension: Adding Pairs

• Syntax as in NQSML

\[
M, N ::= <M, N> \quad \text{pairs} \\
   \mid \text{fst } M \mid \text{snd } M \quad \text{projections} \\
   \mid t \star u \quad \text{pair types}
\]

\[
\text{fst } <M, N> \rightarrow M \quad \text{snd } <M, N> \rightarrow N
\]
**Summary: Static Semantics**

\[
\begin{align*}
\Gamma, x : t \vdash M : u & \quad \Gamma \vdash (\lambda x : t. M) : t \to u \quad \Gamma \vdash MN : u \\
\Gamma \vdash M : t & \quad \Gamma \vdash N : u \\
\Gamma \vdash \langle M, N \rangle : t \times u \\
\Gamma \vdash \text{fst } M : t & \quad \Gamma \vdash \text{snd } M : u
\end{align*}
\]

**Rules for Propositional Logic**

\[
\begin{align*}
\Gamma, p : \mathbb{P} \vdash q & \\
\Gamma, p \vdash q & \quad \Gamma \vdash p \Rightarrow q \quad \Gamma \vdash p \\
\Gamma \vdash p \Rightarrow q & \\
\Gamma \vdash p & \quad \Gamma \vdash q \\
\Gamma \vdash p \land q & \\
\Gamma \vdash p \land q & \quad \Gamma \vdash p & \\
\Gamma \vdash q & \\
\Gamma \vdash \neg p & \quad \Gamma \vdash \neg p & \quad \Gamma \vdash q
\end{align*}
\]

**Erasing All but the Types**

\[
\begin{align*}
\Gamma, t \vdash u & \quad \Gamma \vdash t \Rightarrow u \quad \Gamma \vdash t \\
\Gamma \vdash t \Rightarrow u & \\
\Gamma \vdash t & \quad \Gamma \vdash t \times u & \quad \Gamma \vdash t
\end{align*}
\]

**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 \(\lambda\)-terms are encodings of proofs.
Examples

1. Show that

\[ \vdash p \Rightarrow (p \land p) \]

by finding a term of type

\[ \alpha \rightarrow (\alpha \ast \alpha) \]

2. Show that

\[ \vdash (p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \land q) \Rightarrow r) \]

by finding a term of type

\[ (\alpha \rightarrow (\beta \ast \gamma)) \rightarrow (\alpha \ast \beta) \rightarrow \gamma \]

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 as \((p \Rightarrow \text{false})\).
• Second-order predicate calculus: polymorphic types
  - Second-order = quantifying over propositions.
  - E.g., \(\forall \alpha. \alpha \rightarrow (\alpha \ast \alpha) \) vs. \(\forall p. p \rightarrow (p \land p)\).
• Disjunctions: sum types
• Propositional logic: dependent types
• Modal logic: run-time code generation
• Linear logic: linear types

Intuitionism

• Generally, \(\lambda\)-calculi correspond to constructive or intuitionistic logics.
  - E.g., no terms of type

\[
\begin{align*}
\forall \alpha. \alpha + (\alpha \rightarrow \text{void}) &: \ \text{(vp p or \neg p)} \\
\forall \alpha. ((\alpha \rightarrow \text{void}) \rightarrow \text{void}) &\rightarrow \alpha : \ (vp \rightarrow \neg p \Rightarrow p) \\
\forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \alpha &: \ (\text{Pierce's Law})
\end{align*}
\]

Proof Normalization

• Reductions as proof "simplifications".

\[
\begin{align*}
\Gamma \vdash M : t & \quad \Gamma \vdash N : u \\
\Gamma \vdash <M, N> : t \ast u \\
\Gamma \vdash \text{fst} <M, N> : t & \quad \Gamma \vdash \text{fst} <M, N> : t \\
\Gamma \vdash N & \quad \Gamma \vdash u \\
\Gamma \vdash t & \quad \Gamma \vdash t \land u \\
\Gamma \vdash t & \quad \Gamma \vdash t
\end{align*}
\]
Hilbert System for Implication

**Axiom Schema**

\[
\begin{align*}
p \Rightarrow (q \Rightarrow p) & \\
(p \Rightarrow (q \Rightarrow r)) & \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))
\end{align*}
\]

**Modus Ponens**

\[
\begin{align*}
p \Rightarrow q & \\
p & \\
q
\end{align*}
\]

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.

Summary

<table>
<thead>
<tr>
<th>λ-Calculus</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Proposition</td>
</tr>
<tr>
<td>Term (program)</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction</td>
<td>Proof Normalization</td>
</tr>
</tbody>
</table>

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.
The particular logical framework I will concentrate on today
- Based upon (dependently-typed) \( \lambda \)-calculus
  - So we can use \( \beta \)-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

```
% type of propositions
% 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.

' (¬true) ⇒ (even(successor(0)))' =
imp (not true) (even (succ zero)) : o
```
What About Variables?

- Representing variables as constants:

\[
\forall x. ((\text{even } x) \lor (\text{even } (\text{successor } x)))
\]

\[
\text{forall } "x" (\text{or } (\text{even } "x") (\text{even } (\text{succ } "x")))
\]

- 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:

\[
\forall x. ((\text{even } x) \lor (\text{even } (\text{successor } x)))
\]

\[
\text{forall } (\lambda x : i. (\text{or } (\text{even } x) (\text{even } (\text{succ } x))))
\]

- 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)

- \( \text{true} \): type
- \( \text{truei} : \text{nd} \)
- \( \text{andi} : \text{nd} \rightarrow \text{nd} \rightarrow \text{nd} \)
- \( \text{andel} : \text{nd} \rightarrow \text{nd} \)
- \( \text{nder} : \text{nd} \rightarrow \text{nd} \)

\[
\text{truei} : \text{nd}
\]

\[
\text{andi}(\text{truei}, \text{truei}) : \text{nd}
\]

But \( \text{andel}(\text{truei}) : \text{nd} \) as well!?
Natural Deduction

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

\[ \text{nd : o -> type.} \]

• For any proposition \( p \), the type \( \text{nd } p \) will be the type of proofs of \( p \).
- e.g., \( \text{nd(even(zero))} \) and \( \text{nd(even(succ(zero)))} \)

Natural Deduction: Quantifiers

\[
\begin{align*}
\forall x.p & \quad \text{`\forall x.p` = forall(\lambda x:i.p')} \\
\text{p}[x \rightarrow c] & \quad \text{`p[x \rightarrow c]`} \\
\end{align*}
\]

\[ \text{foralle : nd(forall f) -> C:i \rightarrow nd(f C)} \]

\[ \begin{align*}
\forall x.\text{even(x)} & \quad D' : \text{nd(forall(\lambda x:i.\text{even(x))}} \\
\text{even(1)} & \quad \text{foralle `'D'(succ zero) : nd (even (succ zero))} \\
\end{align*} \]

Natural Deduction

\[
\begin{array}{c|c|c|c|c}
\text{true} & \text{p \& q} & \text{p} & \text{q} \\
\hline
\text{true} & \text{p \& q} & \text{p} & \text{q} \\
\end{array}
\]

\[ \text{nd : o -> type.} \]
\[ \text{truei : nd(true).} \]
\[ \text{andi : nd(A) \rightarrow nd(B) \rightarrow nd(A \& B).} \]
\[ \text{andel : nd(A \& B) \rightarrow nd(A).} \]
\[ \text{ander : nd(A \& B) \rightarrow nd(B).} \]

\[ \text{andel(andi(truei,truei)) : nd(true)} \]
\[ \text{andel(true) is no-longer well-typed!} \]

Applying LF to PL Theory

• Object Language

\[
\begin{align*}
\text{exp : type} & \\
\text{Zero : exp} & \\
\text{Succ : exp \rightarrow exp} & \\
\text{Pair : exp \rightarrow exp \rightarrow exp} & \\
\text{Fst : exp \rightarrow exp} & \\
\text{Fn : (exp \rightarrow exp) \rightarrow exp} & \\
\text{App : exp \rightarrow exp \rightarrow exp} \\
\end{align*}
\]

\[ \text{`fn x \Rightarrow (x,0)`} = \]
\[ \text{F}n(\lambda x:\text{exp.}\text{Pair}(x,\text{Zero})) : \text{exp} \]
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).

v value v1 value v2 value
0 value Succ(v) value (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').

e value e v value v1 v2 value
e1 v1 e2 v2 value
0 e 0 Succ(e) Succ(v) (e1,e2) (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.

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.

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!

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?

\[
\begin{align*}
\text{r1} & \mod 2^32 = \text{r1} \\
\land \; \text{r2} & \mod 2^32 = \text{r2} \land \text{r2} \leq 64 \\
\land \; \text{r3} & \mod 2^32 = \text{r3} \\
\land \; V1. (i \leq 0 \lor i < r2 \land (i \mod 7) = 0) & \implies \text{readable(r1+i)} \\
\land \; V5. (j \leq 0 \lor j < 16 \land (i \mod 7) = 0) & \implies \{\text{readable(r3+j)} \land \text{writable(r3+j)}\} \\
\land \; V1, j. (i \leq 0 \lor i < r2 \land j \leq 0 \lor j < 16) & \implies (r1+i) \neq (r3+j)
\end{align*}
\]