LF: A Logical Framework

Michael Hicks
(thanks to Dave Walker for some slides)
Logical Frameworks

• a logical framework is a meta-language for representing deductive systems
• sample deductive systems
  – logics of all kinds
    • first-order logic, higher-order logic, temporal logic,...
  – programming languages
    • ML, lambda calculus, pi calculus
  – specification languages
    • set theory, type theory, multi-set rewriting, ...
  – compilers
    • translations between languages or logics
# Logical Frameworks

<table>
<thead>
<tr>
<th>meta-logical frameworks</th>
<th>represent properties of logics</th>
<th>Twelf, Nuprl, HOL, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical frameworks</td>
<td>represent logics</td>
<td>Automath, Nuprl, HOL, LF, ...</td>
</tr>
<tr>
<td>logics</td>
<td>represent proofs</td>
<td>FOL, linear logic, temporal logic</td>
</tr>
<tr>
<td>proofs</td>
<td></td>
<td>(A and B) =&gt; C</td>
</tr>
</tbody>
</table>
Logical Frameworks

- **meta-logical frameworks** represent properties of programming langs
  - Twelf, Nuprl, HOL, ...

- **logical frameworks** represent programming langs
  - Automath, Nuprl, HOL, LF, ...

- **programming languages** represent programs
  - C, ML, Java lambda calculus

- **programs**
  - if (n > 17) { ...}
Tasks for Logical Frameworks

• Represent **syntax**
  – of a logic
  – of a programming language

• Represent valid **rules of inference**
  – inference rules (modus ponens, law of the excluded middle) of a logic
  – typing rules, operational rules of a language

• Represent valid **deductions**
  – valid proofs
  – well-typed programs, valid executions
Logical framework design

• Logical framework design resembles programming language design
  – Simple logical frameworks are like simple programming languages
    • few features = simple, trustworthy implementation but limited expressiveness
    • expressive enough in principle but may not scale up (we ignore this problem).
Don’t want to just find new ways of representing trees/text. Want the formal system to help us out:
– Prove the representation is adequate.

This means that there is a correspondence between well-formed representations and the entities they represent.
– Example: given an adequate representation of the expressions and typing derivations (proofs) of a programming language, there is a one-to-one correspondence between typing derivations and their well-formed LF representations. Put another way: type checking in LF amounts to proof checking in the represented language, and vice versa.
LF

• The simply-typed lambda calculus, with a few extra features
  – Dependent types and kinds
    • Needed to capture dependencies arising in derivations
  – Convertibility for canonical forms

• Features
  – Represents pure PL, logics
  – Higher-order abstract syntax
LF syntax (simple)

Types \[ A, B ::= \text{a} \mid A \rightarrow B \]

Objects \[ M, N ::= c \mid x \mid \lambda x:A.M \mid M \; N \]

Signatures \[ \Sigma ::= . \mid \Sigma, a:\text{type} \mid \Sigma, c:A \]

Contexts \[ \Gamma ::= . \mid \Gamma, x:A \]

Signatures \( \Sigma \) define the constants of the system being modeled. We have \( c \) is an object constant, where \( a \) is a type context. If LF were a programming language, these would be things like \( 1 \) for the former, and \( \text{int} \) for the latter.
LF judgments (simple)

\[ +\sum A : \text{type} \quad \text{A is a valid type} \]

\[ \Gamma +\sum M : A \quad \text{M is a valid object of type A in context} \]

\[ +\sum \sum \text{Sig} \quad \sum \text{is a valid signature} \]

\[ +\sum \Gamma \text{Ctx} \quad \Gamma \text{is a valid context} \]
LF convertibility

• We don’t define reduction rules for LF, but rather rules of convertibility (a.k.a. definitional equality)
  – Can apply traditional reduction rules “both ways”
• $\alpha$-conversion
  – Two objects are considered identical if they only differ in the names of bound variables
• $\beta$-conversion
  – $(\lambda x: A. M) N = [N/x]M$
• $\eta$-conversion
  – $(\lambda x: A. M x) = M$
Representing PL Syntax

• Define a type constant for each BNF category
• Define a term constant for each constructor
• Use higher-order abstract syntax for representing binding in the modeled language as binding in the meta-language (which is LF).
  – Advantage: get alpha-conversion and substitution in modeled language for free from meta language
• Define a representation function
  – \([\cdot]\): terms \(\rightarrow\) LF terms
• Examples in the notes
LF Syntax (adding dependency)

| Kinds       | $K ::= \text{type} \mid A \to K$ |
| Types       | $A,B ::= a M \ldots N \mid \Pi x:A.B$ |
| Objects     | $M,N ::= c \mid x \mid \lambda x:A.M \mid M N$ |
| Signatures  | $\Sigma ::= . \mid \sum \!, a:K \mid \sum \!, c:A$ |
| Contexts    | $\Gamma ::= . \mid \Gamma \!, x:A$ |

- Types $A$ can now be *parameterized* by objects $M$. The type $\Pi x:A.B$ is a function type where $x$ is bound in $B$; this is called a *type family*. Normal function types $A \to B$ can be written as $\Pi x:A.B$ where $x$ does not appear free in $B$.
- Type families $a$ are applied to objects $M$ to form *parameterized types* $a M$. 
Example

\[ \Sigma = \text{list}: \text{int} \rightarrow \text{type}, \]
\[ \text{nil}: \text{list} \ 0, \]
\[ \text{cons}: \Pi x: \text{int}. \text{list} \ x \rightarrow \text{int} \rightarrow \text{list} \ (x+1) \]

\[ \Gamma + \Sigma \ \text{cons} \ 1 \ (\text{cons} \ 0 \ \text{nil} \ 1) \ 2 : \text{list} \ 2 \]

- Signature consists of list type parameterized by the list’s length, and standard operations.
- Typechecking is complicated by object parameters! For example, typechecker must ensure that \( \text{list} \ (0+1) = \text{list} \ 1 \), and that \( \text{list}(0+1+1) = \text{list} \ 2 \). Notion of convertibility is lifted to the type level.
Representing PL Deductions

• Define a *type family* for each deduction, parameterized by objects it refers to.

• Define a term constant for each inference rule; non-axioms will be functions

• Define a representation function
  – []: deductions → LF terms

• Examples in notes.
Kinds $K ::= \text{type} \mid \Pi_{x:A} K$

Types $A,B ::= a \mid M \mid \Pi_{x:A} B$

Objects $M,N ::= c \mid x \mid \lambda x:A.M \mid M \ N$

Signatures $\sum ::= \cdot \mid \sum, a:K \mid \sum, c:A$

Contexts $\Gamma ::= \cdot \mid \Gamma, x:A$

- Kinds $K$ can now be \textit{parameterized} by objects $M$. The kind $\Pi_{x:A} K$ is a function kind where $x$ is bound in $K$; this is called a \textit{type family}. Normal kind types $A \rightarrow K$ can be written as $\Pi_{x:A} K$ where $x$ does not appear free in $K$. 
Representing PL Metatheory

• Define a *level 2 type family* (*kind family*) for each metatheoretic induction proof. This family is parameterized by deductions that it relates.
  – For example, type soundness relates the typing judgment to the evaluation judgment

• Define a term constant for each inference rule; non-axioms will be functions

• Define a representation function
  – \([\cdot]\): proofs \(\rightarrow\) LF terms

• Examples in notes.
Proof Adequacy

• Adequacy is not quite what you want here, since meta-theoretic proofs are informal objects. That is, the connection between the LF term and the proof it represents is based on the rules chosen by the user, and this is implicit.

• Contrast the notion of valid (canonical) objects for syntactic elements and normal deductions, which states for valid objects, some proof actually exists.

• Need a notion of coverage, completeness, etc. Ongoing work.