

# Programming with Logical Frameworks

OGST – October 12th, 2004

Adam Poswolsky

Yale University

Advisor: Carsten Schürmann



# Inspiration

Type systems are useful!

- We can represent functions of type  $\text{int} \rightarrow \text{int}$ , why not functions of type  $(\text{int} > 0) \rightarrow (\text{int} > 1)$
- These “extended” (or dependent) types suggest something is proved about the program.

For example, a *red-black tree* has the following properties:

- Root is colored black.
- Distance from root to any leaf has fixed number of black nodes.
- No two consecutive red nodes. (Note that when we insert a node we make it red and fix the properties)

# How do we represent this property?

- Assume we already have some type for trees and  $t$  : tree.
- Assume we have a red-black coloring.  $ct$  : (color  $t$ ) (this is an example of a dependent type).
- We encode “RBT  $ct$ ” to mean that  $ct$  is a valid red-black tree.
- Insertion would have the following type:
  - $(pf : \text{RBT } ct) \rightarrow \text{int} \rightarrow (pf : \text{RBT } ct')$

# What are Logical Frameworks?

This is our choice for data-structures! Used to reason about languages/systems.

- Representation of problem domains.
- Encodings of proofs, certificates, and other complex data structure (via Judgments-as-types paradigm).
- Expressive.

Logically motivated.

# Why LF? – Expressive Power!

- Higher-order Encodings
- Proof Carrying Code. [Necula, Lee]
- Constructive type theories such as Coq and Nuprl provide a mechanism of inductive datatypes incompatible with the existence of higher-order encodings. Therefore, both lack some representational power we desire to leverage.
- Other logical frameworks (LLF, OLF, CLF)

# Why Higher-Order encodings

- How do you represent functions? Manually encoding De-Bruin indices can be messy (as we will see!)
- Programming with “trusted holes”, i.e. trust standard library functions.

# What are trusted holes?

- Let's say you are implementing a function:  
 $\text{mult} : (x:\text{int}) \rightarrow (y:\text{int}) \rightarrow [z:\text{int}] \text{ pf } (\text{times } x \ y \ z)$
- You use a built in function called “std.add”
- Change what we are proving to simply:  
 $\text{mult} : (x:\text{int}) \rightarrow (y:\text{int}) \rightarrow$   
 $(\text{propertyOfplus} \rightarrow ([z:\text{int}] \text{ pf } (\text{times } x \ y \ z)))$

# Crash Course in LF

Programmers: Think combinators!

Logicians: Think Hilbert calculus!

• Formulas:  $A ::= P \mid A \supset B$

• Judgment:  $\vdash A$



$$\frac{}{\vdash A \supset B \supset A} \text{K} \quad \frac{\vdash A \supset B \quad \vdash A}{\vdash B} \text{MP}$$
$$\frac{}{\vdash (A \supset B \supset C) \supset (A \supset B) \supset (A \supset C)} \text{S}$$

# Representation in a Logical Framework

- Judgments-as-types.

$$\ulcorner \vdash A \urcorner : \text{type} = \text{comb } \ulcorner A \urcorner$$

- Derivations-as-objects.

$$\begin{array}{c} \ulcorner \qquad \qquad \qquad \qquad \urcorner \\ \mathcal{H}_1 \qquad \mathcal{H}_2 \\ \vdash A \supset B \quad \vdash A \\ \hline \vdash B \end{array} \text{MP} \quad : \quad \text{comb } \ulcorner B \urcorner \\ = \quad \text{MP } \ulcorner A \urcorner \ulcorner B \urcorner \ulcorner \mathcal{H}_1 \urcorner \ulcorner \mathcal{H}_2 \urcorner$$

# Higher-order Encodings

$o$  : *type*

$\Rightarrow$  :  $o \rightarrow o \rightarrow o$

*comb* :  $o \rightarrow \textit{type}$

$K$  :  $\Pi A : o. \Pi B : o.$

*comb* ( $A \Rightarrow B \Rightarrow A$ )

$S$  :  $\Pi A : o. \Pi B : o. \Pi C : o.$

*comb* ( $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow$   
 $(A \Rightarrow C)$ )

$MP$  :  $\Pi A : o. \Pi B : o.$

*comb* ( $A \Rightarrow B$ )  $\rightarrow$  (*comb*  $A \rightarrow$  *comb*  $B$ )

# More Encodings

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathit{app} e_1 e_2 : \tau_1}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathit{lam} x : \tau_1 . e : \tau_1 \rightarrow \tau_2}$$

$\mathit{exp} : o \rightarrow \mathit{type}$

$\mathit{lam} : \Pi A : o . \Pi B : o .$

$(\mathit{exp} A \rightarrow \mathit{exp} B) \rightarrow \mathit{exp} (A \Rightarrow B)$

$\mathit{app} : \Pi A : o . \Pi B : o .$

$\mathit{exp} (A \Rightarrow B) \rightarrow (\mathit{exp} A \rightarrow \mathit{exp} B)$

# Functional Programming

- Dependent type systems.

DML, Cayenne, Epigram, ...

- Module systems.

SML, Haskell, ...

- Effect Systems.

- Monads

[Moggi, Wadler]

- Exceptions, State

SML

- Arrows

[Hughes]

- Compilation techniques.

[Appel, Harper, Shao, ...]

- Hostlanguage: Domain specific languages.

# Observations

- There is no canonical way of programming.
- Adequacy desired
  - Bijection: Objects and their encoding.
  - Example: Circuits, Typing derivations.
- Consider objects modulo!
  - $\alpha$ : renaming of variables.
  - $\beta$ : substitution application.
  - $\eta$ : canonical representations.
  - ...: many others.

[Fresh ML]

# Summary of Delphin

Delphin =  
Logical Frameworks +  
Functional Programming

Joint work with Carsten Schürmann and Jeffrey Sarnat.

Features:

- *Object* level and *meta* level.
- Computation under  $\lambda$ -binders.

# Simply typed Object Level

- Simply typed logical framework.

Types:  $A ::= a \mid A_1 \rightarrow A_2$

Objects:  $M ::= x \mid c \mid \lambda x : B. M \mid M_1 M_2$

Signatures:  $\Sigma ::= \cdot \mid \Sigma, a : \mathit{type} \mid \Sigma, c : A$

# Our Example (without dependent types)

- Recall our examples
- Remove all formula information

$comb : type$   
 $K : comb$   
 $S : comb$   
 $MP : comb \rightarrow comb \rightarrow comb$

$exp : type$   
 $lam : (exp \rightarrow exp) \rightarrow exp$   
 $app : exp \rightarrow (exp \rightarrow exp)$

# Meta-level

Meta level

Expressions  $e$ , types  $\tau$

---

Object level

Objects  $M$ , types  $A$

# Meta-level: Injection

Injection of object-level into the meta-level.

Types:  $\tau ::= \langle A \rangle$

Expressions:  $e ::= \langle M \rangle$

Example:

$ID$  :  $\langle comb \rangle$   
=  $\langle MP (MP S K) K \rangle$

$ID'$  :  $\langle exp \rangle$   
=  $\langle lam (\lambda x : exp. x) \rangle$

# Meta-level: Functions

Example: Ackermann function

$$\mathit{acker} \ z \ y \quad = \quad \mathit{s} \ y$$

$$\mathit{acker} \ (\mathit{s} \ x) \ z \quad = \quad \mathit{acker} \ x \ (\mathit{s} \ z)$$

$$\mathit{acker} \ (\mathit{s} \ x) \ (\mathit{s} \ y) \quad = \quad \mathit{acker} \ x \ (\mathit{acker} \ (\mathit{s} \ x) \ y)$$

$$\mathit{acker} \quad : \quad \langle \mathit{nat} \rangle \Rightarrow \langle \mathit{nat} \rangle \Rightarrow \langle \mathit{nat} \rangle$$

$$= \quad \langle \mathit{z} \rangle \mapsto \epsilon y : \mathit{nat}. \langle y \rangle \mapsto \langle \mathit{s} \ y \rangle$$

$$\mid \quad \epsilon x : \mathit{nat}. \langle \mathit{s} \ x \rangle \mapsto \langle \mathit{z} \rangle \mapsto \mathit{acker} \cdot \langle x \rangle \cdot \langle \mathit{s} \ z \rangle$$

$$\mid \quad \epsilon x : \mathit{nat}. \langle \mathit{s} \ x \rangle \mapsto \epsilon y : \mathit{nat}. \langle \mathit{s} \ y \rangle$$

$$\mapsto \mathit{acker} \cdot \langle x \rangle \cdot (\mathit{acker} \cdot \langle \mathit{s} \ x \rangle \cdot \langle y \rangle)$$

# Meta-level: Functions

Types:  $\tau, \sigma ::= \langle A \rangle \mid \tau \Rightarrow \sigma$

Expressions:  $e, f ::= \dots \mid u \mid e_1 \mapsto e_2$   
 $\mid \epsilon x : A. e \mid \epsilon u \in \tau. e$   
 $\mid e_1 \cdot e_2 \mid (e_1 \mid e_2) \mid \mathbf{rec} \ u \in \tau. e$

- $\mapsto$ : Pattern-matching.
- $\epsilon$ : Non-deterministic choice.
- $\cdot$ : Application.
- $\mid$ : Alternative.

# Meta-level: Functions

- Separation between  $\epsilon$  and  $\mapsto$  crucial.
  - (see “evaluation under  $\lambda$ ”)
- Strictness system:
  - $\epsilon$  bound variables guaranteed instantiated.
  - Flexible design.
  - *Algorithms for equality and unification in the presence of notational definitions.* [with Frank Pfenning]
  - Work in progress. [Sarnat]

# Meta-level: Evaluation under $\lambda$

Example: Count parameters in a  $\lambda$ -term.

$$\begin{aligned} \mathit{cntvar} (\mathit{app} \ e_1 \ e_2) &= \mathit{plus} (\mathit{cntvar} \ e_1, \mathit{cntvar} \ e_2) \\ \mathit{cntvar} (\mathit{lam} \ e) &= \nu p. \mathit{cntvar} (e \ p) \\ &\text{where } \mathit{cntvar} (p) = (s \ z) \end{aligned}$$

- $\nu$ : Recurse under  $\lambda$ -binder.
- Where clause covers parameter case ( $\nabla$ ).



# Meta-level: Evaluation under $\lambda$

- Possible world semantics.
  - Traversing under a  $\lambda$  = Transition into a new world
- Challenges.
  - Scope extrusion of free parameters.
  - A.K.A: Sound return from the next world.
- Solution.
  - $\nu$ . transitions into the next world.
  - $\nabla$ . accesses hypothesis in this world.

# Meta-level: Evaluation under $\lambda$

Types:  $\tau, \sigma ::= \langle A \rangle \mid \tau \Rightarrow \sigma \mid \Box \tau$

Expressions:  $e, f ::= \dots \mid \nu x : A. e \mid \mathbf{pop} e \mid \nabla x : A. e$

- **pop** is like a modal **box**.
  - Result must be valid in previous world.
- $\nabla$ : “for some parameter”
  - Existential quantifier like  $\epsilon$ .
  - Access parameters in a world.

# Say No to DeBruijn indices

```
<v      : rational -> db>  
<l      : db -> db>  
<a      : db -> db -> db>
```

```
db : <exp> -> <rational> -> <db>  
= {{x}} <x M> |--> <N> |--> <v (N + (~ M))>  
| <app E1 E2> |--> <N> |--> <a> @ (db <E1> <N>)  
                                @ (db <E2> <N>)  
| <lam E> |--> <N> |--> {x:rational -> exp}  
    case (db <E (x (N + 1))> <N + 1>) of  
        (<F> |--> pop <l F>) ;
```

# Type system

- Stacks keep world history.

Object contexts:  $\Gamma ::= \cdot \mid \Gamma, x \in A$

Meta contexts:  $\Phi ::= \cdot \mid \Phi, u \in \tau$

Frame stacks:  $\Omega ::= \cdot \mid \Omega, (\Gamma; \Phi)$

- Typing Judgement:  $\Omega \vdash e \in \tau$ .

# Meta-theoretical results

- Operational semantics  $\hookrightarrow$  omitted.
- Theorem [Subject reduction and scope extrusion]: If  $\Omega \vdash e \in \tau$  and  $\Omega \vdash e \hookrightarrow v$  then  $\Omega \vdash v \in \tau$ .
- Further criteria to investigate:
  - Strictness,
  - Termination,
  - Coverage.

# Implementation

- Working name: Elphin
- Website: [www.cs.yale.edu/~delphin](http://www.cs.yale.edu/~delphin).
- Epsilon reconstruction algorithm.
  - Key insight is that it is type-correct to put all Epsilons at the beginning of the program, and then we can try to work them further in.
- Type reconstruction algorithm.

# Delphin

- Adds Dependent Types to Elphin
- LF [Harper, Honsell, Plotkin]
- Edinburgh Logical Framework.

Kinds:  $K ::= \textit{type} \mid \Pi x : A_1. A_2$

Types:  $A ::= a \mid A_1 \rightarrow A_2 \mid A M \mid \Pi x : A_1. A_2$

Objects:  $M ::= x \mid c \mid \lambda x : B. M \mid M_1 M_2$

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

- Work in Progress but almost complete!

# Future Work

- Polymorphism on Meta-level.
- Effect systems (monads, references)
- Debugger.
  - Datatypes are much more powerful than found in traditional systems. The debugger will need to take the user through the encoded inference rules.