Validating Constructive Meta-Theory with Rogue$^{\Sigma\Pi}$

Aaron Stump
Assistant Professor
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA
http://cl.cse.wustl.edu

Overview

deductive systems ↓ constructive meta-theory

LF ↓ *RSP*

type-preserving compilers
proof-producing decision procedures
meta-theory of higher-order logic

Deductive Systems

- Derive judgments like:
  - “P is a valid formula of classical f.o.l.”
  - “M has type A under typing assumptions Γ”
- Begin by identifying deductive systems with finite axiomatizations in minimal first-order logic.
- 3 kinds of judgments:
  - *atomic*: atomic formula.
  - *hypothetical*: implication
  - *parametric*: universal quantification

Example

dn : p. Valid(Implies(Not(Not(p)),p)

k : p. q. Valid(Implies(p,Implies(q,p)))

s : p. q. r.
Valid(Implies(Implies(p,Implies(q,r)),
Implies(Implies(p,q), Implies(p,r))))

mp : p. q.
Valid(Implies(p,q)) Valid(p) Valid(q)

Term Calculus for Proofs

A term calculus is used for proofs in the meta-logic:

- Proofs of universal and hypothetical judgments are represented as lambda terms.
- Proofs using (meta-logic) modus ponens and instantiation are represented as applications.

Proof by k of Implies(p,Implies(Implies(q,q),p)) is:

k @ p @ Implies(q,q)

Refining the Meta-Language

The meta-logic remains first-order, but:

- Unify meta-logical and . Write "u : p q" (or "p q" if u not free in q). This is useful for restricting the types of parameters.
- Unify proof terms and first-order terms. So, Implies(p,q) becomes Implies @ p @ q. This requires Implies to be viewed as a parametric first-order term. Abbreviate p @ q to p(q).
- Use hypothetical first-order terms to represent binding constructs (*higher-order abstract syntax*).

Edinburgh Logical Framework (LF)

This is our refined meta-logic, due to Harper, Honsell, and Plotkin [HHP93]. It is essentially .

o : *

Implies : o o o

False : o

Valid : o *

Dn : (p : o Valid(Implies @ Not(Not(p)) @ p))

MP : (p : o q : o

Valid(Implies @ p @ q) Valid(p)

Valid(q))

...

Theory and Meta-Theory

Deductive systems	⟶	LF signatures
Judgments	⟶	LF types
Terms, derivations	⟶	LF terms
Meta-theoretic proofs	⟶	???

Example

Deduction Theorem: If hypothetical judgment Valid(p)   Valid(q) is provable, so is atomic judgment Valid(Implies @ p @ q).

Proof: By induction on structure of the derivation d of the hypothetical judgment, with case analysis:

Case d is _x:Valid(p).d, where d is an instance of an axiom (proving formula q):

Valid(Implies @ p @ q) is proved by:

MP @ (K @ q @ p) @ d

Example

Case d is _x:Valid(p).x:

Valid(Implies @ p @ p) is proved like this:

(MP @ (MP @ (S @ p @ (Implies @ p @ p) @ p)
@ (K @ p @ (Implies @ p @ p)))
@ (K @ p @ p)))

Case d is _x:Valid(p).MP @ r @ q @ d1 @ d2:

Valid(Implies @ p @ q) is proved by:

MP @ (MP @ (S @ p @ r @ q) @ d1') @ d2'

where d1' and d2' exist by I.H.

Meta-Theoretic Proofs as Programs

d : Valid(p)   Valid(q)

Proof of
Deduction Theorem

d' : Valid(Implies @ p @ q)

Meta-Theoretic Proofs as Programs

induction $\longrightarrow$ recursion

case analysis $\longrightarrow$ pattern matching

Implementing Meta-Theory

Tactics in ML	Theorem datatype guarantees proofs are built only using the logic's proof rules. But proofs might not check.
LP in Twelf	LF types are viewed as higher-order Horn clauses. Type-checking guarantees all proofs built will check.
Delphin	LF terms are manipulated by pure functional programs. Type-checking guarantees proofs check. Coverage checking is supported.

Rogue$^{\Sigma\Pi}$ (RSP)

- Combines LF and the Rho Calculus (Rogue).
- Separates representation and computation.
- Features new approaches to dependently typed pattern abstractions and dependent pairs.
- Type checking guarantees proofs will check.
- Enables imperative programming using expression attributes.
- Prototype type checker and compiler to untyped Rogue are implemented.
- Several projects underway based on RSP.

Pattern Abstractions

In P^2TS, pattern abstractions look like:

$$P: \;\; . \; M$$

The typing rule is:

$$\frac{, \quad \vdash \quad\; : B \qquad\qquad \vdash \quad P: \;\; . \; B : s}{\vdash \;\; P: \;\; . \; M : \quad P: \;\; . \; B}$$

Comment: it seems rules with different patterns cannot be uniformly combined with “,” (or “|”).

Pattern Abstractions in RSP

RSP's pattern abstractions are of the form:

$$x{=}P{:}\ \ .\ M$$

The typing rule is:

$$\frac{,\quad \vdash P : A \qquad ,\quad ,\ x{=}P \vdash \quad : \qquad \vdash \quad {}^{c}x{:}A.\ B : s}{\vdash \quad x{=}P{:}\ \ .\ M : \quad {}^{c}x{:}A.\ B}$$

So types do not depend on the form of the pattern.
Conversion uses equation x=P.

Recursive Functions in RSP

- Implemented via recursive equations.
- These can be implemented just using *expression attributes*.
  - a.b attribute read
  - Set(a.b, c) attribute write
- We set a.b to be some abstraction mentioning a.b.
- RSP's type system keeps attribute expressions out of types. Otherwise, type preservation would fail: consider reflexivity of conversion on (c @ Set(a.b,a.b+1)).

Representational Abstractions

- HOAS represents binding constructs from the object language as meta-language functions.
- This is fine in LF, since LF functions are computationally very weak.
- Arbitrary recursive functions are too expressive.
- RSP supports *representational abstractions* x:A   B, in addition to pattern abstractions.

Evaluation Order for RSP

Leftmost innermost order is used for evaluating RSP expressions, with two exceptions:

- no evaluation is performed in the body of a pattern abstraction (standard for programming languages).
- evaluation **is** performed in the bodies of representational abstractions. This appears to be needed to enable programming with higher-order abstract syntax.

Constructs of RSP

M @ N	application
x \ P \ D M	pattern abstraction (computational)
x : A M	pure abstraction (representational)
$x : A \;{}_{c}\, B$	computational function space
$x : A \;{}_{p}\, B$	representational function space
Null(A)	for match failure, uninitialized attribute
M | N	deterministic choice (computational)
*	the basic kind
a.b	attribute read (computational)
Set(a.b, c)	attribute write (computational)
(x : A, B)	dependent sum type
(x = M,N)	dependent pair
M.i	projections (for i in {1,2})
M:A	ascription

Proof of Deduction Theorem in RSP

base : *
rvc : base
dedthm : (base $_c$
A : O B : O $_c$ (Valid(A) Valid(B)) $_c$
Valid(Implies @ A @ B))

dedthm_h : (base $_c$ (u:O Valid(u))
A : O B : O $_c$ Valid(B) $_c$
Valid(Implies @ A @ B))

Set(rvc.dedthm,
A:O B:O \ null D:(Valid(A) Valid(B)) \ null
(bridge : (u:O Valid(u))
rvc.dedthm_h @ bridge @ A @ B @ (D @ bridge(A))) @
Null(u:O Valid(u)))

Proof of Deduction Theorem in RSP

```
Set(rvc.dedthm_h,
  bridge : (u:O     Valid(u))     A:O
    (B \ A \ null     F \ bridge @ B \ null
      MP @ (MP @ (S @ A @ (Implies @ B @ B) @ B)
                      @ (K @ A @ (Implies @ B @ B))
           @ (K @ A @ B)  |

     B:O \ null
        (F \ MP_ @ P @ B @ d1 @ d2
           \ (P : O, d1 : Valid(Implies @ P @ B), d2 : Valid(P))
          MP @ (MP @ (S @ A @ P @ B)
                          @  (rvc.dedthm_h @ bridge @ A
                                                  @ (Implies @ P @ B) @ d1)))
               @ (rvc.dedthm_h @ bridge @ A @ P @ d2) |

        D : Valid(B) \ null     MP @ (K @ B @ A) @ D))
```

Applications

proof-producing decision procedures
type-preserving compilers
meta-theory of higher-order logic

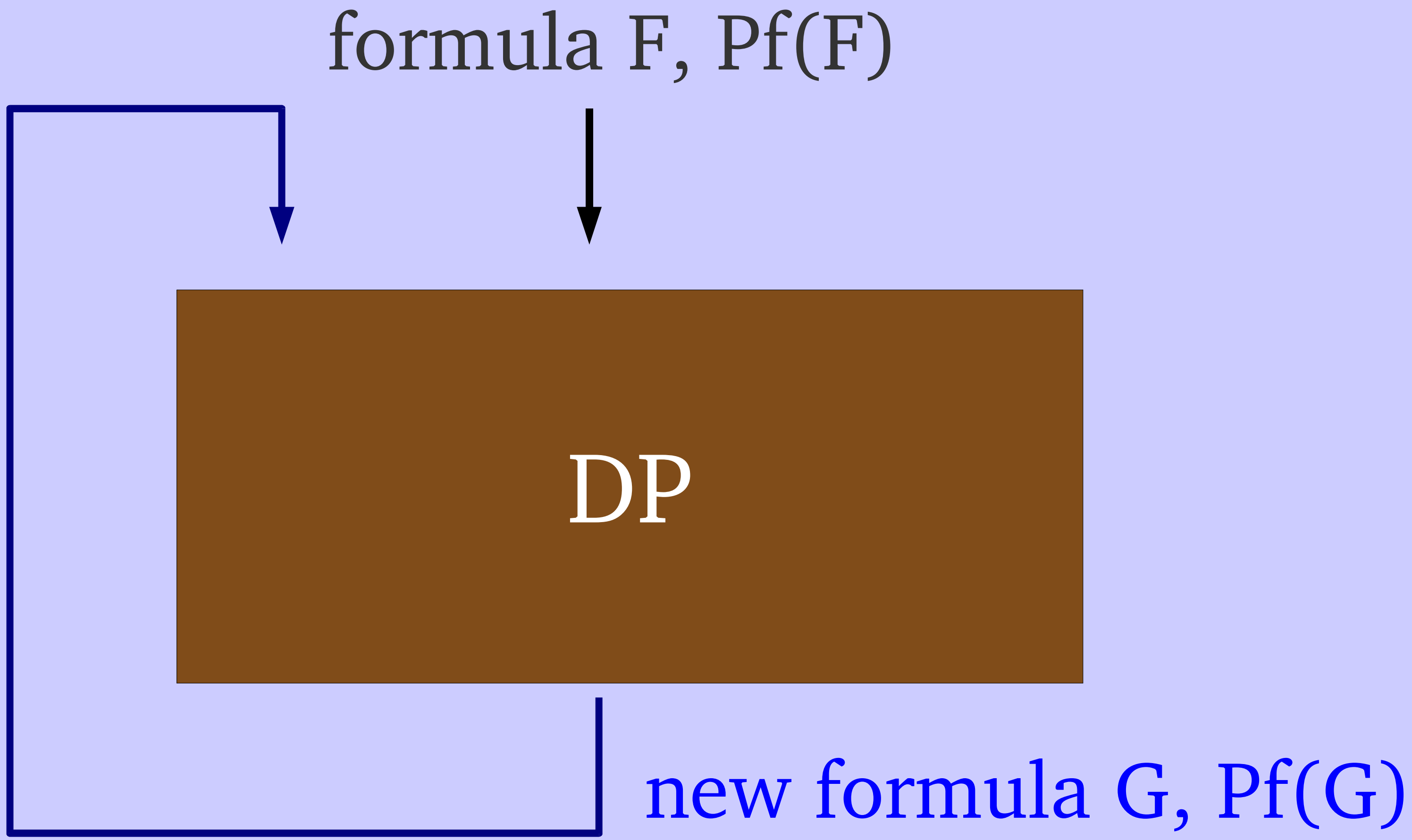
Proof-Producing Decision Procedures

- Decision procedures (DPs) for first-order theories are increasingly important in automated reasoning and verification.
- To incorporate their results, applications like proof-carrying code require explicit proofs to be produced.
- Proofs can catch soundness bugs (rather rare).
- Many bugs caught in proof production code!
- For long runs, proofs are huge and slow to check.

Proof-Producing DPs in RSP

- Type preservation for RSP ensures that LF proof objects produced by the DP would always check.
- Nulls can creep into proofs due to run-time errors.
- In the absence of Nulls, any RSP proof object represents a well-formed proof.
- Hence, proofs produced by succesful runs of the DP do not need to be checked or even produced.
- Under some restrictions, we can slice out all the proof producing code except for a little residue to propagate Nulls.

Proof-Producing Saturating DPs



Pairs are essential to this approach.

Dependent Pairs in LF

Adding dependent pairs to LF breaks unicity of types, and thus bottom-up type checking. One repair is to require ascriptions at every pair [Sarnat 2003, Yale TR].

Suppose U(x,y) is of type Pf(Equals @ x @ y), and consider:

(y, U(x,y)) :   z:I. Pf(Equals @ x @ z)

vs.

(y, U(x,y)) :   z:I. Pf(Equals @ x @ y)

Dependent Pairs in RSP

Sometimes casts can be avoided, if we take pairs to be of the form:

$$(x{=}M,N)$$

The typing rule is:

$$\frac{\vdash \quad : A \qquad , x{=}M \vdash \quad : \qquad \vdash \quad x{:}A.\ B : *}{\vdash (x{=}M,N) : \quad x{:}A.\ B}$$

For bottom-up checking, we use conversion just when checking ascriptions and applications.

Example

Suppose U(x,y) is of type Pf(Equals @ x @ y), and consider:

	Computed type:
(z=y, U(x,z))	z:I. Pf(Equals @ x @ z)

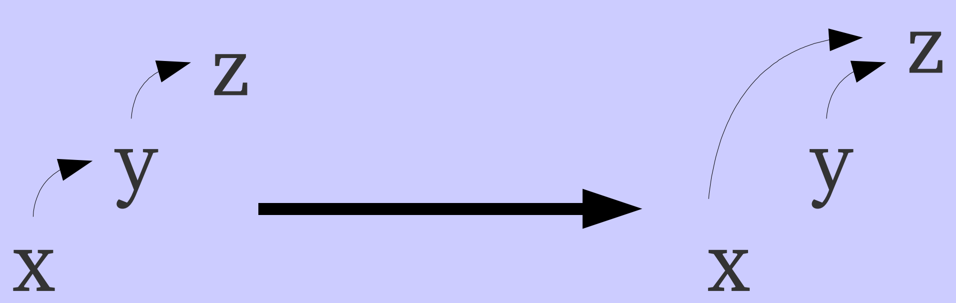
vs.

(z=y, U(x,y))	z:I. Pf(Equals @ x @ y)

In practice, it seems ascriptions are still frequently needed.

Union-Find

- Equational reasoning often relies on union-find.
- Equivalence classes are maintained as disjoint trees.
- The root of the tree is the canonical representative for the equivalence class.
- Each member of the class has a pointer ("findp") towards the root.
- Path compression bashes pointers to the root.



Proof-Producing Union-Find in RSP

- Use an attribute for findp.
- For individual x, x.findp stores a pair:
  - the first element is the individual y that x's find pointer points to.
  - the second element is a proof that x equals y.
- Path compression connects proofs using transitivity of equality.

findp : i $\quad {}_{c}$(y:i, Pf(Equals @ x @ y))

RSP Code for Find

rank : (I $_c$ Int)

findp : (x : I $_c$ (y:I, Pf(Equals @ x @ y)))

find : (base $_c$ x : I $_c$ (y:I, Pf(Equals @ x @ y)))

union : (base $_c$ x : I $_c$ y:I $_c$ Pf(Equals @ x @ y) $_c$ Int)

```
Set(uf.find, x : I \ null ->
   Let(fx, x.findp,
   Ite(fx,
       Let(ffx, uf.find @ fx.1,
           Set(x.findp, (y \ ffx.1,
                         Eqtrans @ x @ fx.1 @ y @ fx.2 @ ffx.2))),
       Drop1(Set(x.rank, 0),
             (y \ x, Eqrefl @ x : Pf(Equals @ x @ y)))))))
```

Type-Preserving Compilers

- Proposed by Morrisett and others both for improved compiler quality and to certify resulting code to code consumers.
- "From System F to Typed Assembly Language" shows how to compile a polymorphic pure functional language to assembly.
- The compiler is proved – on paper – to preserve types correctly.
- Implementing in RSP allows us to prove type preservation of an actual implementation.

Alvin Compiler

^{F} type t, Pf(P:t)

P, ^{F} type t, Pf(P:t)

proof-producing
^{F} type checker

CPS conversion

P ^{F}

P' , type t', Pf(P':t')

...

Meta-Theory of Classical H.O.L.

Peter Andrews's logic Q_0 is a classical higher-order logic based on simply typed lambda abstractions and equality.

It has inference rules like Rule R, “if X=Y is a theorem and C is a theorem, then so is D, where D is C with a single (non-binding) occurrence of X replaced by Y”.

This rule allows variable capture.

Q_0 in RSP

- The natural shallow embedding is not faithful.
- A deep embedding quickly becomes extremely tedious to use:
  - substitution (albeit not capture-avoiding substitution) must be defined.
  - for replacement in proofs from hypotheses, eigenvariable restrictions must be enforced by hand.
  - logical rules like replacement now require proofs of syntactic judgments.
- We are implementing (validated) tactics to help alleviate this burden.

Current Prototype System

RSP (600 lines Rogue)

Rogue (50 lines MicroRogue)
+ standard library (70 lines Rogue)

MicroRogue (2000 lines C++)