# Dependently Typed Programming with Domain-Specific Logics


## Dan Licata

**Thesis Committee:**

Robert Harper

Karl Crary

Frank Pfenning

Greg Morrisett, Harvard

# Domain-specific logics

Type systems for reasoning about a specific application domain/programming style:

* Cryptol: cryptographic protocols
* Ynot/HTT: imperative code
* Aura and PCML5: access to controlled resources

# Domain-specific logics

Type systems for reasoning about a specific application domain/programming style:

- **Cryptol: cryptographic protocols**
- Ynot/HTT: imperative code
- Aura and PCML5: access to controlled resources

# Cryptol

**Track length in type**

swab : Word 32 → Word 32

swab [a b c d] = [b a c d]

**Pattern-match as four Word 8's**

# Cryptol

swab : Word 32 → Word 32

swab [a b c d] = [b c d]

# Cryptol

swab : Word 32 → Word 32

swab [a b c d] = [b ~~c~~ d]

**Type error!**

# Domain-specific logics

Type systems for reasoning about a specific application domain/programming style:

- Cryptol: cryptographic protocols
- **Ynot/HTT: imperative code**
- Aura and PCML5: access to controlled resources

# Ynot

- Start with lax modality for mutable state: $\bigcirc A$
- Index with pre/postconditions:

ST P A Q

**Precondition:**
**heap → prop**

**Postconditon:**
**Π a:A, initial: heap, final:heap. prop**

# Ynot

$$\mathsf{read} : \Pi r{:}\mathsf{loc}.\ \mathsf{ST}\ (r \hookrightarrow_A -)\ A\ (\lambda a\, i\, f.\ f = i\ \wedge \\ \forall v{:}A.\ (r \hookrightarrow v)\, i \rightarrow a = \mathsf{Val}\, v)$$

$$\mathsf{write} : \Pi r{:}\mathsf{loc}.\ \Pi v{:}A.\ \mathsf{ST}\ (r \hookrightarrow -)\ \mathsf{unit}\ (\lambda a\, i\, f.\ a = \mathsf{Val}\ \mathsf{tt}\ \wedge \\ f = \mathsf{update\_loc}\, i\, r\, v)$$

# Domain-specific logics

Type systems for reasoning about a specific application domain/programming style:

* Cryptol: cryptographic protocols
* Ynot/HTT: imperative code
* **Aura and PCML5: access control**

# Security-typed PL

Authorization logic [Garg + Pfenning]:

- Resources (F): files, database entries, …
- Principals (K): users, programs, …
- Permissions: K mayread F, …
- Statements by principals: K says A, …
- Proofs

# Security-typed PL

Principals and resources:

```
sort : type.
princ : sort.
res   : sort.

term : sort -> type.
admin : term princ.
dan   : term princ.
bob   : term princ.
```

# Security-typed PL

Permissions:

```
aprop : type.

owns  : term princ -> term res -> aprop.
mayrd : term princ -> term res -> aprop.
maywt : term princ -> term res -> aprop.
```

# Security-typed PL

Propositions:

```
prop : type.

atom    : aprop -> prop.
implies : prop -> prop -> prop.
says    : term princ -> prop -> prop.
all     : (term S -> prop) -> prop.
```

**HOAS**

# Security-typed PL

Judgements: **$\Gamma \Rightarrow$ (A true) and**

**$\Gamma \Rightarrow$ (K affirms A)**

```
conc    : type.

true     : prop -> conc.
affirms : term princ -> prop -> conc.
```

**A true**

**K affirms A**

# Security-typed PL

Judgements: hyp : prop -> type.
|- : conc -> type.

Sequent A1 … An ⇒ C

**A true or K affirms A**

represented by

A1 hyp -> … -> An hyp -> |- C

# Security-typed PL

Proofs:

```
saysr : |- (K says A) true
        <- |- K affirms A.

saysl : ((K says A) hyp -> |- K affirms C)
        <- (A hyp -> |- K affirms C).
```

# Security-typed PL

Policy:

```
ownsplan :
  (atom (dan owns /home/dan/plan)) hyp.

danplan  :
 (dan says (all [p] atom (p mayrd /home/dan/plan))) hyp.
```

# Security-typed PL

Access controlled-primitives:

read : ∀r:term res.

∀D : |- (atom (self mayrd r)) true.

string

**need a proof of authorization to call read!**

# Security-typed PL

Compute with derivations:

- Policy analysis
- Auditing: log cut-full proofs;
  eliminate cuts to see who to blame [Vaughn+08]

# Domain-specific logics

Type systems for reasoning about a specific application domain/programming style:

- Cryptol: cryptographic protocols
- Ynot/HTT: imperative code
- Aura and PCML5: access control

# Domain-specific logics

How are they implemented?

- Cryptol: stand-alone
- Ynot/HTT: extension of Coq
- Aura and PCML5: stand-alone

# Problems

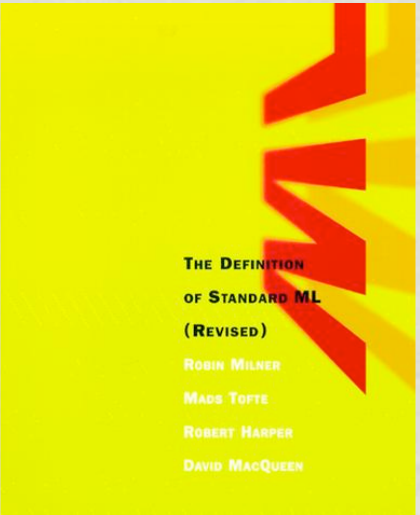
* Engineer compiler, libraries, documentation
* Train/convince programmers
* Hard to use multiple DSLs in one program
* Programmer can’t pick the appropriate abstraction

# This work

A host language that makes it easy to:

- Represent domain-specific logics
- Reason about them (mechanized metatheory)
- Use them to reason about code (certified software)

# Ingredients




+


- **functional programming**
- **effects: state, exceptions, ...**
- **polymorphism and modules**

- **binding and scope**
- **dependent types**
- **total programming**

# Thesis contributions

Previous work [LICS08]:

Integration of binding and computation
using higher-order focusing

# Thesis contributions

Proposed work:

| Theory | Practice |
| --- | --- |
| • Dependency | • Meta-functions |
| • Effects | • Term reconstruction |
| • Modules | |

# Outline

- Previous work
- Proposed work
- Related work

# Outline

- **Previous work**
- Proposed work
- Related work

# Polarity [Girard '93]

Sums $A + B$ are positive:

- Introduced by choosing inl or inr
- Eliminated by pattern-matching

ML functions $A \to B$ are negative:

- Introduced by pattern-matching on A
- Eliminated by choosing an A to apply to

# Focusing [Andreoli '92]

Sums $A + B$ are positive:

- Introduced by choosing inl or inr
- Eliminated by pattern-matching

ML functions $A \to B$ are negative:

- Introduced by pattern-matching on A
- Eliminated by choosing an A to apply to

# Focusing [Andreoli ’92]

Focus = make choices

Sums $A + B$ are positive:

- Introduced by choosing inl or inr
- Eliminated by pattern-matching

ML functions $A \to B$ are negative:

- Introduced by pattern-matching on A
- Eliminated by choosing an A to apply to

# Focusing [Andreoli '92]

Sums $A + B$ are positive:

- Introduced by choosing inl or inr
- Eliminated by pattern-matching

ML functions $A \to B$ are negative:

- Introduced by pattern-matching on A
- Eliminated by choosing an A to apply to

# Focusing [Andreoli ’92]

Sums $A + B$ are positive:

- Introduced by choosing inl or inr
- Eliminated by pattern-matching

**Inversion = respond to all possible choices**

ML functions $A \to B$ are negative:

- Introduced by pattern-matching on A
- Eliminated by choosing an A to apply to

# Binding + computation

1. Computation: negative function space ($A \rightarrow B$)

2. Binding: positive function space ($P \Rightarrow A$)

   - Specified by intro $\lambda u.V$

   - Eliminated by pattern-matching

# Arithmetic expressions

Arithmetic expressions with let-binding:

let x be (const 4) in (plus x x)

e ::= const n
  | let x be e1 in e2
  | plus e1 e2
  | times e1 e2
  | sub e1 e2
  | mod e1 e2
  | div e1 e2

# Arithmetic expressions

Arithmetic expressions with let-binding:

let x be (const 4) in (plus x x)

```
e ::= const n
    | let x be e1 in e2
    | plus e1 e2
    | times e1 e2
    | sub e1 e2
    | mod e1 e2
    | div e1 e2
```

*Suppose we want to treat binops uniformly…*

# Arithmetic expressions

Arithmetic expressions with let-binding

e ::= const n
  | let x be e1 in e2
  | binop e1 φ e2

where φ : (nat → nat → nat) is
the code for the binop.

# Arithmetic expressions

const : nat ⇒ exp

let : exp ⇒ (exp ⇒ exp) ⇒ exp

binop : exp ⇒ (nat → nat → nat) ⇒ exp ⇒ exp

---

let x be (const 4) in (x + x)

represented by

let (const 4) (λx.binop x add x)

where add:(nat → nat → nat) is the code for addition

# Structural properties

Identity, weakening, exchange, contraction, substitution, subordination-based strengthening

- Free in LF
- May fail when rules use computation

# Weakening

Can’t necessarily go from

f : nat → nat

proof by induction

to

extends nat with new datatype constructor

(weaken f) : nat ⇒ (nat → nat)

doesn’t have a case for the new variable!

# Structural properties

Our solution:

- $\lambda$x.V eliminated by pattern-matching: Nothing forces $\Rightarrow$ to be structural
- But structural props may be implemented generically for a wide class of rule systems

# Structural properties

const : nat ⇒ exp

let : exp ⇒ (exp ⇒ exp) ⇒ exp

binop : exp ⇒ (nat → nat → nat) ⇒ exp ⇒ exp

- Can't weaken exp with nat:
  could need new case for → in a binop
- Can weaken exp with exp:
  doesn't appear to left of →

# Higher-order focusing

Zeilberger’s higher-order focusing:

- Specify types by their patterns
- Type-independent focusing framework
  - Focus phase = choose a pattern
  - Inversion phase = pattern-matching

# Higher-order focusing

Zeilberger's higher-order focusing:

Inversion = pattern-matching is *open-ended*

Represented by meta-level functions
from patterns to expressions

**Use datatype-generic programming
to implement structural properties!**

# Higher-order focusing

$$\boxed{\Delta; \Psi \Vdash p :: A^+}$$

**Pattern-bound variables**

**Inference rule context:**

let : exp ⇒ (exp ⇒ exp) ⇒ exp,

...

# Higher-order focusing

$$\frac{\Delta_1 ; \Psi \Vdash p_1 :: A^+ \quad \Delta_2 ; \Psi \Vdash p_2 :: B^+}{\Delta_1, \Delta_2 ; \Psi \Vdash (p_1, p_2) :: A^+ \otimes B^+}$$

$$\frac{\Delta ; \Psi \Vdash p :: A^+}{\Delta ; \Psi \Vdash \mathsf{inl}\, p :: A^+ \oplus B^+} \qquad \frac{\Delta ; \Psi \Vdash p :: B^+}{\Delta ; \Psi \Vdash \mathsf{inr}\, p :: A^+ \oplus B^+}$$

# Higher-order focusing

$$\frac{u : P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+ \in (\Sigma, \Psi) \qquad \Delta_1 ; \Psi \Vdash p_1 :: A_1^+ \quad \ldots \quad \Delta_n ; \Psi \Vdash p_n :: A_n^+}{\Delta_1, \ldots, \Delta_n ; \Psi \Vdash u\, p_1 \ldots p_n :: P}$$

$$\frac{\Delta; \Psi, u{:}R \Vdash p :: B^+}{\Delta; \Psi \Vdash \lambda u.p :: R \Rightarrow B^+}$$

# Higher-order focusing

Inversion = pattern-matching:

$$(\text{case } (e : \,<\Psi>\, A) \text{ of } \varphi) \quad : \quad C$$

$\varphi$ : Function from $(\Delta\, ;\, \Psi \Vdash p :: A)$ to expressions of type $C$ in $\Delta$

**Infinitary: when A is nat, one case for each numeral**

# Outline

- Previous work
- **Proposed work**
- Related work

# Proposed work

<u>Theory</u>

- Dependency
- Effects
- Modules

<u>Practice</u>

- Meta-functions
- Term reconstruction

# Proposed work

Theory

- **Dependency**
- Effects
- Modules

Practice

- Meta-functions
- Term reconstruction

# Dependency

Three levels of ambitiousness

- Dependency on LF
- Dependency on positive data
- Dependency on negative computation, too

# Dependency on LF

First-order quantifiers over LF terms:

$$\frac{\Psi \vdash_{\mathrm{LF}} M : A \qquad \Delta\,;\Psi \Vdash p :: \tau^{+}(M)}{\Delta\,;\Psi \Vdash (M, p) :: \exists_A(\tau^{+})}$$

Pattern-bound variables

LF context

Meta-function mapping LF terms to positive types

# Dependency on LF

Derived elimination form is infinitary, with one case for each LF term M of appropriate type

pres: ∀ E E':exp, T:tp.

∀ D1 : of E T.  ∀ D2 : step E E'.

∃ D' : of E' T. unit

# Dependency on LF

Meta-function $\tau$ used for logical relations:

HT (arr T2 T) E =

∀ E2:exp.  HT T2 E2 → HT T (app E E2)

**Defined by recursion on T**

# Positively dependent

- Integrate $\Rightarrow$ and $\rightarrow$ as in LICS paper
- Allow dependency on patterns for positive types: subsumes LF
- No need to compare negative computations for equality

# Negatively dependent

- After-the-fact verification
- Predicates on higher-order store in HTT
- Judgements about computationally higher-order syntax

# Proposed work

<u>Theory</u>

- Dependency
- **Effects**
- Modules

<u>Practice</u>

- Meta-functions
- Term reconstruction

# Effects

- See proposal document for refs
- Open question:

  Controlling effects and
  programmer-defined indexed modalities
  (ST P A Q)

  **Defined in LF**

# Proposed work

Theory

- Dependency
- Effects
- Modules

**Practice**

- Meta-functions
- Term reconstruction

# Practice

- Finitary syntax for meta-functions:

  1. positive (unification) variables

  2. structural properties

- Term reconstruction: steal from Twelf/Agda

# Outline

- Previous work
- Proposed work
- **Related work**

# Related work

Why is our language is better for programming with DSLs than…

- NuPRL, Coq, Epigram, Agda, Omega, ATS, ...
- Twelf, LF/ML, Delphin, Beluga
- Nominal logic/FreshML

# Conclusion

**Thesis statement:**

*The logical notions of polarity and focusing provide a foundation for dependently typed programming with domain-specific logics, with applications to certified software and mechanized metatheory.*

# Conclusion

**Proof:**

- Theory: polarized type theory with support for binding, dependency, effects, modules
- Practice: meta-functions, reconstruction, implementation, examples

Thanks for listening!