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

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]
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: OA

* Index with pre/postconditions:

- Precondition: heap $\rightarrow$ prop
- Postcondition: $\Pi a:A, \text{initial: heap, final:heap. prop}$
Ynot

\[
\begin{align*}
\text{read} : \Pi r: \text{loc. } \Pi A : \text{ST } (r \leftrightarrow_A \_A \_A) A (\lambda a \_i f. f = i \land \\
\forall v : A. (r \leftrightarrow v) i \rightarrow a = \text{Val } v)
\end{align*}
\]

\[
\begin{align*}
\text{write} : \Pi r: \text{loc. } \Pi A : \text{ST } (r \leftrightarrow \_) \text{unit } (\lambda a \_i f. a = \text{Val } \text{tt } \land \\
f = \text{update_loc } i \_r v)
\end{align*}
\]
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 may read 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.
Security-typed PL

Judgements: \[ \Gamma \Rightarrow (A \text{ true}) \] and \[ \Gamma \Rightarrow (K \text{ affirms } A) \]

conc : type.

true : prop \rightarrow\ conc.

affirms : term princ \rightarrow prop \rightarrow conc.

A true

K affirms A
Security-typed PL

Judgements: \[ \text{hyp} : \text{prop} \to \text{type}. \]
\[ \text{|-} : \text{conc} \to \text{type}. \]

Sequent \[ A_1 \ldots A_n \Rightarrow C \]
represented by
\[ A_1 \text{ hyp} \to \ldots \to A_n \text{ hyp} \to \text{|- C} \]

A true or K affirms A
Security-typed PL

Proofs:

\[ \text{saysr} : \vdash (K \text{ says } A) \text{ true} \]
\[ \quad \leftarrow \vdash K \text{ affirms } A. \]

\[ \text{saysl} : ((K \text{ says } A) \text{ hyp } \rightarrow \vdash K \text{ affirms } C) \]
\[ \quad \leftarrow (A \text{ hyp } \rightarrow \vdash K \text{ affirms } C). \]
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:

\[
\text{read} : \forall r : \text{term res.}
\]

\[
\forall D : |- (\text{atom (self mayrd r)}) \text{ true.}
\]

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**
- Dependency
- Effects
- Modules

**Practice**
- Meta-functions
- Term reconstruction
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 $A \rightarrow B$ 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 $A \rightarrow B$ 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

Focus = make choices
Focusing [Andreoli ’92]

Sums $A + B$ are positive:
- Introduced by choosing $\text{inl}$ or $\text{inr}$
- Eliminated by pattern-matching

ML functions $A \rightarrow B$ 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 $A \rightarrow B$ are negative:
- Introduced by pattern-matching on $A$
- Eliminated by choosing an $A$ to apply to

Inversion = respond to all possible choices
Binding + computation

1. Computation: \textcolor{red}{negative} function space \((A \rightarrow B)\)

2. Binding: \textcolor{green}{positive} function space \((P \Rightarrow A)\)
   - Specified by intro \(\lambda u. V\)
   - Eliminated by pattern-matching
Arithmetic expressions

Arithmetic expressions with let-binding:

```plaintext
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)

\[ e ::= \text{const } n \]
\[ | \text{let } x \text{ be } e_1 \text{ in } e_2 \]
\[ | \text{plus } e_1 e_2 \]
\[ | \text{times } e_1 e_2 \]
\[ | \text{sub } e_1 e_2 \]
\[ | \text{mod } e_1 e_2 \]
\[ | \text{div } e_1 e_2 \]

Suppose we want to treat binops uniformly…
Arithmetic expressions

Arithmetic expressions with let-binding

\[ e ::= \text{const } n \]
\[ \mid \text{let } x \text{ be } e_1 \text{ in } e_2 \]
\[ \mid \text{binop } e_1 \varphi e_2 \]

where \( \varphi : (\text{nat} \to \text{nat} \to \text{nat}) \) is the code for the binop.
Arithmetic expressions

\[
\begin{align*}
\text{const} & : \text{nat} \Rightarrow \text{exp} \\
\text{let} & : \text{exp} \Rightarrow (\text{exp} \Rightarrow \text{exp}) \Rightarrow \text{exp} \\
\text{binop} & : \text{exp} \Rightarrow (\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \Rightarrow \text{exp} \Rightarrow \text{exp}
\end{align*}
\]

\[
\text{let } x \text{ be } (\text{const } 4) \text{ in } (x + x)
\]

represented by

\[
\text{let } (\text{const } 4) (\lambda x. \text{binop } x \text{ add } x)
\]

where \text{add}:(\text{nat} \rightarrow \text{nat} \rightarrow \text{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

\( f : \text{nat} \rightarrow \text{nat} \)

proof by induction

to

\( (\text{weaken } f) : \text{nat} \Rightarrow (\text{nat} \rightarrow \text{nat}) \)

extends nat with new
datatype constructor

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**

\[
\begin{align*}
\text{const} & : \text{nat} \Rightarrow \text{exp} \\
\text{let} & : \text{exp} \Rightarrow (\text{exp} \Rightarrow \text{exp}) \Rightarrow \text{exp} \\
\text{binop} & : \text{exp} \Rightarrow (\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}) \Rightarrow \text{exp} \Rightarrow \text{exp}
\end{align*}
\]

- Can’t weaken exp with nat: could need new case for \( \Rightarrow \) in a binop
- Can weaken exp with exp: doesn’t appear to left of \( \Rightarrow \)
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

Pattern-bound variables

Inference rule context:

let : exp ⇒ (exp ⇒ exp) ⇒ exp,

...
Higher-order focusing

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

\[
\frac{\Delta ; \Psi \vdash p :: A^+}{\Delta ; \Psi \vdash \text{inl} p :: A^+ \oplus B^+}
\]

\[
\frac{\Delta ; \Psi \vdash p :: B^+}{\Delta ; \Psi \vdash \text{inr} p :: A^+ \oplus B^+}
\]
Higher-order focusing

\[ u : P \iff A_1^+ \cdots \iff A_n^+ \in (\Sigma, \Psi) \]

\[ \Delta_1 ; \Psi \vdash p_1 :: A_1^+ \quad \cdots \quad \Delta_n ; \Psi \vdash p_n :: A_n^+ \]

\[ \Delta_1, \ldots, \Delta_n ; \Psi \vdash up_1 \ldots p_n :: P \]

\[ \Delta ; \Psi, u : R \vdash p :: B^+ \]

\[ \Delta ; \Psi \vdash \lambda u. p :: R \Rightarrow B^+ \]
Higher-order focusing

Inversion = pattern-matching:

\[(\text{case } (e : < \Psi > A) \text{ of } \varphi) : \ C\]

\[\varphi : \text{Function from } (\Delta ; \Psi \vdash p :: A) \text{ to expressions of type } C \text{ in } \Delta\]

Infinitary: when \(A\) is nat, one case for each numeral
Outline

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

**Theory**
- Dependency
- Effects
- Modules

**Practice**
- 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:

\[
\Psi \vdash_{\text{LF}} M : A \quad \Delta; \Psi \vdash p :: \tau^+ (M) \\
\Delta; \Psi \vdash (M, p) :: \exists_A (\tau^+) 
\]
Dependency on LF

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

\[
\text{pres: } \forall E E': \text{exp}, T: \text{tp}.
\]
\[
\forall D1 : \text{of } E T. \quad \forall D2 : \text{step } E E'.
\]
\[
\exists D' : \text{of } E' T. \text{ unit}
\]
Dependency on LF

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

$$\text{HT} \ (\text{arr} \ T_2 \ T) \ E = \ \forall \ E_2:\text{exp.} \ \text{HT} \ T_2 \ E_2 \rightarrow \ \text{HT} \ T \ (\text{app} \ E \ E_2)$$

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

<table>
<thead>
<tr>
<th>Theory</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Dependency</td>
<td>• Meta-functions</td>
</tr>
<tr>
<td>• Effects</td>
<td>• Term reconstruction</td>
</tr>
<tr>
<td>• Modules</td>
<td></td>
</tr>
</tbody>
</table>
Effects

- See proposal document for refs

Open question:

Controlling effects and programmer-defined indexed modalities

\((ST \ P \ A \ Q)\)

Defined in LF
Proposed work

<table>
<thead>
<tr>
<th>Theory</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Dependency</td>
<td>• Meta-functions</td>
</tr>
<tr>
<td>• Effects</td>
<td>• Term reconstruction</td>
</tr>
<tr>
<td>• Modules</td>
<td></td>
</tr>
</tbody>
</table>
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!