

Mechanizing the Metatheory of Programming Languages

Robert Harper

Computer Science Department
Carnegie Mellon University

October 2008

Acknowledgements

Thanks to my collaborators and colleagues, especially

- Frank Pfenning.
- Karl Crary and Daniel K. Lee.

Thanks to INRIA for the invitation!

Program Verification

Program verification is increasingly important.

- Preserving security and reliability of software systems.
- Quality of life experience for users.
- Controlling the cost of development and maintenance.

Many approaches are being tried, with great success.

- Post-hoc program analysis, e.g. model checking.
- Program extraction, e.g. Coq and NuPRL.
- Equivalence checking, e.g. bisimulation for processes.
- State transition logics, e.g. separation logic.

Program Verification

Verification is **difficult** (but steadily getting easier).

- What to verify?
- Complexity of program analysis.
- Education of developers.

The easiest route to verification is via **type systems**.

- A little verification goes a long way.
- Integration of verification with development.
- Linguistic record of properties.

Language Verification

One theorem *per language* implies one theorem *per program*.

- Large economy of scale.
- Trade-off precision against practicality.

Sounds good in *theory*, but does it work in *practice*?

- Can we design useful and practical type systems?
- Can we verify their properties?

Language Verification

Expressive type systems abound!

- Every other POPL paper is about type systems.
- Convergence of types and verification, e.g. Coq, Agda.
- Security types, session types, Hoare triple types, dependent types,

Each type system must be backed by a body of [metatheory](#).

- Expose often subtle flaws in the design, e.g. unsoundness.
- Clarify assumptions of the model.
- Enable certification of source and even object code.

Example: ML5

Mobility and **locality** are inherent in distributed computing.

- Resources such as sensors or databases reside **somewhere**.
- Mobile code must be capable of execution **anywhere**.

Spatial modalities capture mobility and locality constraints.

- $M : A @ w$ means M is usable at location w .
- $M : \Box A$ means M is **mobile** (usable at **any** location).
- $M : \Diamond A$ means M is **located** (usable at **some** location).

Example: ML5

Theorem

A well-typed ML5 program cannot access a resource other than at the location at which it resides.

Why is this interesting?

- Captures the crucial invariants of spatial awareness.
- Avoids a class of run-time errors by static checking.

Why is this hard?

- Resources can be moved around passively.
- Must establish strong invariants on computations.

Example: PCA

Administrative domains are a fact of life in a distributed system.

- Intellectual property restrictions.
- Privacy restrictions.

Logic-based access control:

- Access control policy is a theory in **authorization logic**.
- Accessor a must **prove** to owner o

$\text{policy} \vdash o \text{ says } a \text{ mayacc } r.$

- Certificates are **assumptions** discharged by run-time checks.

Example: PCA

Problem: access control policies can be hard to understand!

- Unintended authorizations through convoluted chains of reasoning.
- Combinations of policies have unintended consequences.

Theorem (Cut Elimination)

*Every derivation of an entailment in PCA has a **direct** proof.*

Useful for analysis of policies.

- Examine all direct proofs for a given policy.
- Explore scenarios: find a (direct) proof of an entailment.

Example: PCA

PCML5: *security types* for ML5.

- $M : A @ w ! p$: computation of type A usable at w by p .
- $\forall r::PF(w \text{ says } p \text{ mayacc } r).(file[r] \rightarrow \text{string}) @ w ! p$.

Theorem (Auditability)

All accesses to controlled resources by a well-typed PCML5 program are authorized by a valid proof in authorization logic from specified policy.

Language Verification

Verifying such properties is tedious and error-prone!

- Many cases to consider, even for toy languages.
- Much of the proof is routine, but for a few key steps.
- Most theorems are not very deep.

The solution is [mechanization of metatheory](#).

- Formal, analyzable definitions of languages.
- Machine-checked proofs of meta-theoretic properties.

Language Verification

We use Twelf routinely in our day-to-day work:

- Safety of ML5 type system for distributed computing.
- Cut elimination for Authorization Logic.
- Verification of compiler transformations.
- POPLmark challenge problem.

We have also used it for large-scale verifications:

- Type safety for Standard ML.
- TALT certification infrastructure for mobile code.

Mechanizing Metatheory

Representation in the LF Logical Framework.

- Higher-Order Abstract Syntax
- Judgements-as-Types Principle

Verification using the Twelf prover.

- Relational Metatheory
- Totality and Coverage Checking

Representation

The LF *language*:

- Very *weak* dependently typed λ -calculus.
- *Decidable* type checking.

The LF *methodology*:

- HOAS, Judgements-as-Types.
- *Adequacy*: canonical forms of certain types in certain contexts.

LF Representation Language

Main idea: *families* of types and objects given by *generators*.

LF Representation Language

Main idea: *families* of types and objects given by *generators*.

Sets

LF

LF Representation Language

Main idea: **families** of types and objects given by **generators**.

Index

Sets

/ set

LF

I : type.

LF Representation Language

Main idea: **families** of types and objects given by **generators**.

	Sets	LF
Index	I set	$I : \text{type}.$
Family	$\{X_i\}_{i \in I}$	$X : I \rightarrow \text{type}.$

LF Representation Language

Main idea: **families** of types and objects given by **generators**.

	Sets	LF
Index	I set	$I : \text{type.}$
Family	$\{X_i\}_{i \in I}$	$X : I \rightarrow \text{type.}$
Object	$\{x_i \in X_i\}_{i \in I}$	$x : \{i:I\} (X i)$

LF Representation Language

Main idea: **families** of types and objects given by **generators**.

	Sets	LF
Index	I set	$I : \text{type.}$
Family	$\{X_i\}_{i \in I}$	$X : I \rightarrow \text{type.}$
Object	$\{x_i \in X_i\}_{i \in I}$	$x : \{i:I\} (X i)$

Derived families of objects may be built up from generators.

LF Representation Language

Main idea: **families** of types and objects given by **generators**.

	Sets	LF
Index	I set	$I : \text{type.}$
Family	$\{X_i\}_{i \in I}$	$X : I \rightarrow \text{type.}$
Object	$\{x_i \in X_i\}_{i \in I}$	$x : \{i:I\} (X\ i)$

Derived families of objects may be built up from generators.

- $[i:I] (M\ i)$ is a family of type $\{i:I\} (X\ i) \dots$

LF Representation Language

Main idea: **families** of types and objects given by **generators**.

	Sets	LF
Index	I set	$I : \text{type.}$
Family	$\{X_i\}_{i \in I}$	$X : I \rightarrow \text{type.}$
Object	$\{x_i \in X_i\}_{i \in I}$	$x : \{i:I\} (X\ i)$

Derived families of objects may be built up from generators.

- $[i:I] (M\ i)$ is a family of type $\{i:I\} (X\ i) \dots$
- \dots provided $i:I$ implies $M\ i : X\ i$

Representing Syntax

Each **syntactic category** is represented by a **type**.

```
tp : type.           % types
exp : type.          % expressions
```

Each **syntactic construct** is represented by a **generator**.

Representing Syntax

Each **syntactic category** is represented by a **type**.

```
tp : type.           % types
exp : type.          % expressions
```

Each **syntactic construct** is represented by a **generator**.

```
bool : tp.
tt : exp.
ff : exp.
if : exp -> exp -> exp.
```

Representing Syntax

Each **syntactic category** is represented by a **type**.

```
tp : type.           % types
exp : type.          % expressions
```

Each **syntactic construct** is represented by a **generator**.

```
bool : tp.           nat : tp.
tt  : exp.           z  : exp.
ff  : exp.           s  : exp -> exp.
if  : exp -> exp -> exp. plus : exp -> exp -> exp.
```

Higher-Order Syntax

Binding and **scope** are managed by LF.

`let : exp -> (exp -> exp) -> exp.`

An object of type `exp -> exp` represents an expression with a distinguished free variable.

Thus “let x be $1+1$ in $x+x$ ” is represented by the object

`let (plus (s z) (s z)) ([x] (plus x x))`

of type `exp`.

Adequacy for Syntax

There is a **compositional bijection** between

- Syntactic classes, and
- **Canonical objects** of specified LF **types** in specified **worlds**
- (Compositional = commutes with substitution.)

For types and expressions, we have

Class	Type	World
Types	tp	(empty)
Expr's	exp	$x : exp$

The represented language “exists” as canonical objects of these types in such contexts.

Representing Judgements

A **judgement** is identified with the **type** of its **derivations**.

`of : exp -> tp -> type.`

Each **inference rule** is specified by a **generator**.

$$\frac{}{z : nat}$$

`of/z : of z nat.`

$$\frac{E : nat}{s(E) : nat}$$

`of/s :`
`of (s E) nat <-`
`of E nat.`

Representing Judgements

Higher-order syntax requires higher-order judgements:

$$\frac{E_1 : T_1 \quad x : T_1 \vdash E_2 : T_2}{\text{let } x \text{ be } E_1 \text{ in } E_2 : T_2}$$

of/let :

of (let E1 ([x] E2 x)) <-
of E1 T1 <-
{x} of x T1 -> of (E2 x) T2).

The LF declarations are just an “ascii-fication” of the rules!

Representing Judgements

Structural operational semantics:

```
value : exp -> type           % values
step  : exp -> exp -> type.    % transition
```

Rules are specified by generators:

```
st/plus1 :
  step (plus E1 E2) (plus E1' E2) <-
  step E1 E1'.
st/plus2 :
  step (plus E1 E2) (plus E1 E2') <-
  value E1 <- step E2 E2'.
```

Representing Judgements

Adequacy for judgements:

Class	Type	World
Typing	of E T	$x:\text{exp}, dx:\text{of } x T$
Transition	step E E'	(none)

There is a compositional bijection between

- Typings $E : T$ and objects of type (of E T).
- Transitions $E \mapsto E'$ and objects of type (step E E').

Relational Meta-Theory

Theorem (Preservation)

If $E : T$ and $E \mapsto E'$, then $E' : T$.

The proof is by **induction** on $E \mapsto E'$:

- One case for each transition rule.
- Recursively transforms derivations.

Implicitly, the proof defines a **total relation** from

- Derivations $\nabla_1 :: E \mapsto E'$, and ...
- Derivations $\nabla_2 :: E : T$ to ...
- Derivations $pres(\nabla_1, \nabla_2) :: E' : T$.

Relational Meta-Theory

By adequacy for judgement, derivations are (represented by) LF objects!

Consequently, we may formalize the preservation proof as follows:

- Define a judgement `pres` relating derivations.
- Specify the `inputs` and `outputs` of `pres`.
- Verify `totality`: $\forall \text{inputs} \exists \text{outputs}$ satisfying the relation.

The “core competence” of Twelf is proving $\forall \exists$ theorems over LF canonical forms!

Relational Meta-Theory

The judgement `pres` is defined like all other judgements!

- By a collection of generators representing rules.
- Subjects are themselves derivations (of typing and transition judgements).

The generators constitute the “meat” of the proof!

- Each generator specifies an `inductive step`.
- Totality check ensures that all cases are covered, and that the induction is well-founded.

Preservation Proof

Relational formulation of preservation:

```
pres : step E E' -> of E T -> of E' T' -> type.  
%mode +Ds +Dt -Dt'.
```

A case of the proof:

```
- : pres  
  (st/plus/1 Ds1)      % plus E1 E2 |-> plus E1' E2  
  (of/plus Dt2 Dt1)   % plus E1 E2 : nat  
  (of/plus Dt2 Dt1') % plus E1' E2 : nat  
<- pres Ds1 Dt1 Dt1' % IH Dt1' : of E1' nat
```

Preservation Proof

The other cases are formalized very similarly.

Specify the **worlds** to consider:

```
%block pb : some {T:tp} block {x:expr} {dx:of x T}  
%worlds (pb) (pres Ds Dt Dt')
```

Check **totality** by induction on transition derivation:

```
%total Ds (pres Ds _ _)
```

Relational Meta-Theory

Some observations about the methodology:

- It's just like functional programming, but in a relational style.
- You finish with a proof, not a proof script.
- Much of what you write is “forced” by the type system of LF.

But does it scale?

- Applicable to other proofs?
- Suitable for larger languages?

Case Study: SML

Scaling up to Standard ML:

- Full-scale language with state, exception, recursive types.
- Rich module and abstraction mechanism.
- Type inference, pattern matching, limited form of type classes.

Goals:

- A maintainable, evolvable definition of the language.
- Mechanically verifiable meta-theory.

Case Study: SML

Challenges:

- Scale: The Definition of Standard ML is a 100-page book!
- Experience: Previous efforts have failed.
- Feasibility: Can it be done in a reasonable amount of time?

Case Study: SML

Challenges:

- Scale: The Definition of Standard ML is a 100-page book!
- Experience: Previous efforts have failed.
- Feasibility: Can it be done in a reasonable amount of time?

Yes We Can!

Mechanizing SML

Strategy:

- ① Define and verify safety of a **kernel type theory**, KTT.
- ② Define an **elaboration** from SML to KTT.
- ③ Prove the **static correctness** of elaboration.
- ④ **Validate** the mechanization empirically.

Advantages:

- Builds on advances in type systems.
- Separates **essence** from **accident** in SML.
- Integrates with implementation (towards compiler certification).

Kernel Type Theory

Central issue: type equality [Lillibridge, Leroy, Stone, Dreyer].

- Scoped type definitions: `type t = int*int.`
- Scoped type sharing: `sharing type t=u.`

High-level architecture of KTT:

Kinds $K ::= \text{Type} \mid \text{Eqv}(c) \mid \Pi u::K_1.K_2 \mid \Sigma u::K_1.K_2$

Types $T ::= t \mid T_1 \times T_2 \mid T_1 \rightarrow T_2 \mid \dots$

Signatures $S ::= [u::K, T] \mid \Pi s::S_1.S_2 \mid \Sigma u::S_1.S_2$

Kernel Type Theory

Type safety depends on **inversion** of equality.

- $\text{int} \not\equiv \text{bool}$ (ow, $\text{true} + \text{false}$ is well-typed)
- $T \times U \equiv T' \times U'$ implies $T \equiv T'$ and $U \equiv U'$
- ... and many other similar properties.

Singleton kinds introduce **scoped** equations.

- If $c :: \text{Eqv}(d)$, then $c \equiv d :: \text{Type}$.
- `type t=int*int` becomes $t : \text{Eqv}(\text{int} \times \text{int})$.
- `sharing type t=u` becomes $u :: \text{Type}, v :: \text{Eqv}(u)$.

Consequently, inversion principles are **very** non-obvious!

Kernel Type Theory

Central issue is **transitivity** of equivalence.

If $T \times U \equiv T' \times U'$ follows from

- $T \times U \equiv R$ and
- $R \equiv T' \times U'$

there is no obvious way to proceed when R is not $R_1 \times R_2$.

A typical complication in proof theory and type theory!

Kernel Type Theory

Formulate *two forms* of type equivalence.

- **Declarative**: *what* types are equal. Useful for showing two types are equal.
- **Algorithmic**: *how* to determine if types are equal. Useful for analyzing equal types.

About half of the work is in showing these equivalent!

- Stone and Harper: a **logical relations** argument.
- Crary: a proof based on **hereditary substitution**.

Algorithmic Equality

Harper-Stone: $\Gamma \vdash A \Leftrightarrow B :: K$.

- Constructors A and B are **extensionally equivalent** at kind K .
- Decompose according to kind K , then compare results structurally.
- Completeness via a logical relations argument.

Crary-Watkins: define $[A/u]_K B$ for canonical forms.

- Critical case: $u :: K$ is in head position.
- Performs β -normalization during substitution.
- Termination by a direct complexity measure assignment.

Kernel Type Theory

Formalization and verification of KTT complete [POPL 2006].

- About 55K lines of Twelf code.
- About 1.5 man-years of work by a first-year graduate student.

Half the work is devoted to analysis of equivalence!

- About 25K lines of Twelf.
- Applicable to many other languages!

Elaboration

Next step is **elaboration** of SML to KTT.

- Identifier resolution.
- Type inference / reconstruction.
- Datatype representation and pattern matching.
- Type class mechanism.

Elaboration judgements: $\mathcal{E}; \Gamma \vdash \text{exp} \Rightarrow E : T$.

- exp is SML abstract syntax.
- \mathcal{E} represents bookkeeping data structures.

Elaboration

Elaborator makes heavy use of **non-determinism**.

- Ideal for specification and verification.
- Complicates implementation (more later).

Example: inferring missing type information.

$$\frac{\mathcal{E}; \Gamma \vdash \text{exp}_1 \Rightarrow E_1 : T_2 \rightarrow T \quad \mathcal{E}; \Gamma \vdash \text{exp}_2 \Rightarrow E_2 : T_2}{\mathcal{E}; \Gamma \vdash \text{exp}_1 \text{exp}_2 \Rightarrow E_1 E_2 : T}$$

Static Correctness

Theorem (Static Correctness)

if $\mathcal{E}; \Gamma \vdash \text{exp} \Rightarrow E : T$, then $\Gamma \vdash E : T$ in KTT.

The proof is given relationally as a transformation

- **Input:** derivations of $\mathcal{E}; \Gamma \vdash \text{exp} \Rightarrow E : T$;
- **Output:** derivations of $\Gamma \vdash E : T$

Twelf checks mechanically checks totality of transformation!

Corollary: SML is type safe!

Static Correctness

Elaboration and static correctness complete (Spring 2008).

- About 11K lines of Twelf code, one man-year of effort.
- Main complication: pattern compilation.

The [same](#) elaboration is used in the TILT compiler.

- Constitutes a verification of first phase of TILT.
- Uncovered a few (easily repairable) bugs.

Validation

But have we done anything at all!?

- For all we know the elaborator rejects **all** programs.
- How do we know that it elaborates Standard ML?

Solution: empirical validation against a test suite.

- Relies on an **implementation** of elaboration.
- Could/should use TILT, but instead we “got (too?) clever.”

Validation

Twelf provides a **logic programming** interpretation of types.

- Given exp , find E and T (if any) such that

$$\mathcal{E}; \Gamma \vdash \text{exp} \Rightarrow E : T$$

- Totally unusable in practice because of indeterminacy!

We are developing an **effective elaborator**:

- **Deterministic**, hence directly executable by Twelf.
- **Certifying**: produces an (declarative) elaboration on each run.

Deterministic Elaboration

Main idea: implement Damas-Milner in Twelf.

- Re-use Twelf's (pattern) unification.
- Identify generalizable type variables.
- Explicit generalization and instantiation.

Two key techniques:

- Identifying existential variables in a logic program.
- Reducing unification to LF pattern fragment.

Deterministic Elaboration

Testing if a term is **not** an existential variable:

```
isnt_evar_bool : tp -> bool -> type.  
- : isnt_evar_bool junk false.  
- : isnt_evar_bool _ true.  
%deterministic isnt_evar_bool  
isnt_evar : tp -> type.  
- : isnt_evar M <- isnt_evar_bool M X <- eq X true.
```

Ideas:

- junk can only unify with an existential variable.
- %det rules out backtracking.

Deterministic Elaboration

Testing if a term **is** an existential variable:

```
is_evar_bool : tp -> bool -> type.  
- : is_evar_bool M false <- isnt_evar M.  
- : is_evar_bool _ true.  
%deterministic is_evar_bool.  
is_evar : tp -> type.  
- : is_evar M <- is_evar_bool M X <- eq X true.
```

Main idea: negation-by-failure.

Deterministic Elaboration

Status: expected completion December, 2008?

- Unless there are further complications.
- Logic programming approach may have limitations.

Conclusion

LF and Twelf are remarkably effective for mechanizing metatheory.

- Great for proof-of-concept verifications.
- Scales to realistic languages.

Please attend our tutorial at POPL in Charleston!

See twelf.org for more!