Mechanizing Language Definitions

Robert Harper
Carnegie Mellon University
September, 2005

Acknowledgements

- Thanks to
  - Michael Ashley-Rollman
  - Susmit Sarkar
  - Karl Crary
  - Frank Pfenning
- Thanks to the ICFP PC for the invitation!

Language Definitions

- What does it mean for a programming language to exist?
- The “standard” answer is exemplified by C.
  - Informal description (a la K&R, say).
  - A “reference” implementation (gcc, say).
  - Social processes such as standardization committees.

Language Definitions

- The PL research community has developed better definitional methods.
  - Classically, various grammatical formalisms, denotational and axiomatic semantics.
  - Most successfully, type systems and operational semantics.
- Nearly all theoretical studies use these methods! (e.g., every other ICFP paper)

Language Definitions

- What good is a language definition?
  - Precise specification for programmers.
  - Ensures compatibility among compilers.
  - Admits rigorous analysis of properties.
- The Definition of Standard ML has proved very successful in these respects!

Language Definitions

- But a language definition is also a burden!
  - Someone has to maintain it.
  - Not easy to make changes.
- Definitions can be mistaken too!
  - Internally incoherent.
  - Difficult or impossible to implement.

Language Definitions

- A definition alone is not enough! Must maintain a body of meta-theory as well.
  - Type safety: coherence of static and dynamic semantics.
  - Decidability of type checking, determinacy of execution, ...
- Developing and maintaining the meta-theory is onerous.

Mechanized Definitions

- Can we alleviate some of the burden through mechanization?
  - Formalize the definition in a logical framework.
  - Automatically or semi-automatically verify key meta-theoretic properties.
- Can we do this at scale?

Formalizing Languages

- This talk is about using Twelf to
  - Formalize language definitions.
  - Reason about their meta-theory.
- Several other groups are using Coq, Isabelle, and other provers for similar purposes!
  - Too early to judge what's “best” (IMO).

What I've Learned

- Twelf is a very convenient and effective tool for mechanized meta-theory.
  - Natural, pattern-matching style of presentation.
  - Easy to state and verify simple, but informative, invariants.
  - A "type system" for language definitions: simple sanity checks are powerful!

What I've Learned

- One cannot (and should not) expect a "waterfall" process.
  - Definitional technique is influenced by the demands of mechanization.
  - Mechanization process uncovers mistakes, ambiguities, infelicities in the language.
  - LF tends to enforce good hygiene (rather than require contortions).

What I've Learned

- LF/Twelf is not the last word!
  - The methodology is robust and likely to remain useful.
  - There is a clear path to improvement (e.g., linearity, structural congruences).
  - It will never be everything to everyone.

LF Methodology

- Establish a compositional bijection between
  - objects of each syntactic category of object language
  - canonical forms of associated types of the LF lambda calculus
- “Compositional” means “commutes with substitution”, giving meaning to variables.

LF Methodology

- For a PL the syntactic categories include
  - abstract syntax, usually including binding and scoping conventions
  - typing derivations
  - evaluation derivations
- The latter two cases give rise to the slogan “judgements as types”.

Example: STLC

```
% abstract syntax
tp : type.
b : tp.
arrow : tp -> tp -> tp.
tm : type.
lam : tp -> (tm -> tm) -> tm.
app : tm -> tm -> tm.
```

Example: STLC

```
% typing (excerpt)

of : tm -> tp -> type.

of_lam :
  ({x : tm}{dx : of x T} of (F x) U) ->
  of (lam T F) (arr T U).

of_app :
  of E1 (arr T U) -> of E2 T ->
  of (app E1 E2) U.
```

Example: STLC

% evaluation (excerpt)

```
step : tm -> tm -> type.

beta :
  step (app (lam T F) E) (F E).

fun :
  step E1 E1' -> step (app E1 E2) (app E1' E2).
```

Adequacy Theorem

Crucial!

Cat'y	Rep'n	Contexts/World
τ type	T : tp	
e term	E : tm	x : tm
e : τ	D : of E T	x : tm, dx : of x U

Meta-Reasoning

- Adequacy ensures that we can reason about the object language by analyzing canonical forms of appropriate LF type.
  - Canonical forms are long $\beta\eta$ normal forms.
  - Simultaneous and iterated structural induction over canonical forms.
- Applies to informal and formal reasoning!

Meta-Reasoning in Twelf

- Twelf supports checking of proofs of Pi_2 ($\forall\exists$) propositions over canonical forms in a specified class of contexts (world).
  - Enough for preservation, progress, ...
- These are totality assertions for a relation between inputs ($\forall$/+) and outputs ($\exists$/-)!
  - Polarity notation is a relic ...

Relational Meta-Theory

- Preservation Theorem as a relation:

```
pres : of E T -> step E E' -> of E' T -> type.
```

- Axiomatize this relation:

```
pres_beta :
  pres (of_app (of_lam D) D')
       beta
       (D _ D').
```

etc.

plug arg typing into function body typing

Relational Meta-Theory

- Ask Twelf to verify the totality of the relation representing the theorem.
  - Specify the worlds to consider.
  - Specify input/output mode of the relation.
  - Specify induction principle to use.
- Checks that all cases are covered, and induction is used appropriately.

Relational Meta-Theory

- For preservation this consists

```
%mode pres +D1 +D2 -D3.
%worlds () (pres _ _ _).
%total D (pres _ D _).
```

∀typings ∀steps

∃typing

closed deriv's

induction on steps

- Twelf performs a mode check, world check, and a coverage and termination check.
  - Coverage similar to ML exhaustiveness.

Relational MetaTheory

- For simple examples like this we can easily prove progress and preservation.
  - Formulate the inductive steps using pattern matching.
  - Check coverage and totality.
- Substitution, weakening, contraction are all provided “free” by the framework.

Life’s Not Always Easy

- It’s not always so straightforward!
  - Some features present challenges.
  - Often the challenges uncover implicit structure or expose design problems.
  - Sometimes you just suffer.
- But you can get surprisingly far with a bit of experience and ingenuity.

Adding A Store

- Suppose we wish to add reference cells to the language a la ML.
- The typing judgement has the form

$$\Gamma \vdash_{\Lambda} e : \tau$$

- Here Λ is a location typing assigning types to memory locations.

Adding A Store

- The "obvious" encoding is to add a location typing to the typing judgement:

```
of : lt -> tm -> tp -> type.
```

- Typing rules change accordingly:

```
of_lam :
  ({ x : tm }{ dx : of L x T }
    of L (F x) U) ->
    of L (lam T F) (arrow T U).
```

Adding A Store

- Adequacy for $\ldots\ x_i : \tau_i\ \ldots \vdash_\Lambda e : \tau$
  - canonical forms of type of L e T
  - in worlds ... xi : tm, dxi : of L x T ...
- But we cannot prove type preservation (for the usual operational semantics)!

Adding A Store

- Typing should be preserved by allocation!
- Must consider a coherent family of type systems, not just one at a time:

$$\frac{\Gamma \vdash_{\Lambda} e : \tau \quad \Lambda \subseteq \Lambda'}{\Gamma \vdash_{\Lambda'} e : \tau}$$

Adding A Store

- But if we simply add this rule ...

```
weaken :
  of L E T -> ext L L' -> of L' E T -> type.
```

- ... the encoding is no longer adequate!
  - Lose compositionality criterion.

Adding A Store

- Why compositionality fails:
  - Suppose x : tm, dx : of L x T.
  - Suppose _ : of L′ (E′ x) T′ in this context.
  - Suppose _ : of L′ E T.
- Cannot conclude _ : of L (E′ E) T′.
  - Even if the typings arose via weakening!

Adding A Store, Revisited

- The "trick" is to remove the location typing from assumptions!
  - Side-steps the mismatch just observed.
  - But is substitution still valid?
- Illustrates a recurring technique of isolating variables for special treatment, without abandoning HOAS!

Adding A Store, Revisited

- Retain location typing on main judgement:
  `of : lt -> tm -> tp -> type.`
- Add a typing judgement for assumptions:
  `assm : tm -> tp -> type.`
- Consider worlds of the form
  `x : tm, dx : assm x T`

Adding A Store, Revisited

- Add an explicit "hypothesis" rule:

  ```
  of_var : assm E T -> of L E T.
  ```

- Revise typing rules accordingly:

  ```
  of_lam :
   ( { x : tm } { dx : assm x T } of L (F x) U )
      -> of L (lam T F) (arrow T U).
  ```

Meta-Theory For Stores

- However, we now must check that substitution preserves typing.

```
subst_pres:
  ({x : tm}{dx : assm x T} of L (F x) U) ->
   of L E T -> of L (F E) U.
%mode subst_pres +D1 +D2 -D3.
```

- The proof is easily verified using Twelf.

Reasoning About Variables

- Quite often one wishes to prove a meta-theorem about the behavior of variables.
  - eg, substitution preserves typing
  - eg, narrowing a variable to a subtype
- Since the context is represented only implicitly in LF, these can be a bit tricky.
  - eg, POPLmark challenge for F<:

Reasoning About Variables

- For example, why does this type ...

```
({x : tm}{dx : assm x T} of (F x) U) ->
 of E T -> of (F E) U -> type.
```

- ... codify this substitution principle?

if G,x:T,G' |- F : U and G |- E : T,
then G,G' |- [E/x]F : U

Reasoning About Variables

- The key is permutation, which permits us to regard G,x:T,G′ as G,G′,x:T in STLC.
- If permutation is available, it is easy to prove properties of variables.
  - Any given variable may be thought of as occurring “last”.
- But what if we don’t have permutation?

Reasoning About Variables

- From the POPLmark challenge:
  if G, X<:Q, G' |- A <: B, and G |- P <: Q,
  then G, X<:P, G' |- A <: B.

- Stated relationally,

```
narrow :
  ( {X:tp} {dX : assm X Q} sub A B ) ->
  sub P Q ->
  ( {X:tp} {dX : assm X P} sub A B ) ->
  type.
```

Reasoning About Variables

- But this statement cannot be proved!
  - Descending into a binder introduces an additional assumption, say Y<:X.
  - Cannot permute Y<:X before X<:Q!
- So we must consider a general G′, which cannot be done uniformly in LF.
  - The context G′ is not a "single thing".

Reasoning About Variables

- Adequacy for F<: is for worlds of the form

  ```
  X : tp, dX : assm X T
  ```

- For example,

  ```
  tlam_of :
    ({X : tp}{dX : assm X T}
      of (F X) (U X)) ->
        of (tlam T F) (all T U).
  ```

Reasoning About Variables

- We cannot, in general, permute such pairs past one another due to dependencies.
- But, a limited form of permutation is OK:

```
{ X : tp } { Y : tp }
{ dY : assm Y X } { dX : assm X P }
```

- The strategy is to permit “mixed” permutations so that an assm can be last!

Reasoning About Variables

- Revised relational statement of narrowing permits X to be separated from dX:

```
{X:tm} ({dX : assm X Q} sub A B) ->
       sub P Q ->
       ({dX : assm X P} sub A B) ->
       type.
```

- But now assm X Q no longer ensures that X is a variable!

Reasoning About Variables

- We "tag" each variable and "link" it to an `assm` assumption:

```
var : tm -> type.
assm_var : assm X T -> var X -> type.
%mode assm_var +D1 -D2.
```

- Consider context blocks of these forms:
  - `X : tp, vX : var X`
  - `dX : assm X T, dvX : assm_var dX vX`

Solving POPLmark

- This was the hardest problem in the POPLmark challenge!
  - The rest was handled easily using standard methods with no serious complications.
  - This solution is a simplification of another that was much harder.
- We solved the challenge in about a week!

Scaling Up

- We use Twelf daily at CMU for mechanizing meta-theory.
  - Checking proofs in research papers on languages and logics.
  - Building a certification infrastructure for ConCert.
- But does it scale to “real” languages?

Scaling Up To SML

- A full-scale language such as SML presents many complications.
  - Scope resolution.
  - Type inference.
  - Pattern compilation.
- (Not to mention multi-parameter constructor classes with functional dependencies!)

Scaling Up To SML

- The formalization in The Definition presents some further obstacles:
  - Doesn't support a direct statement of type safety (need "wrong").
  - Implicit evaluation rules.
  - Complications involving signature matching.
  - Ad hoc semantic objects (cf Russo).

Scaling Up To SML

- These complexities have significantly impeded mechanization.
  - To my knowledge, there is no complete proof of soundness of Standard ML!
- The demands of formalization suggest re-factoring The Definition.
  - Useful for other reasons (e.g., in TILT).

A Type-Theoretic Definition of SML

- Formalize the elaboration of SML into HSIL.
  - Type inference, equality compilation, pattern compilation, coercive matching.
  - Takes care of the “conveniences” of ML.
- Formalize HSIL as a conventional type system with SOS rules for evaluation.
  - Easily representable in LF.

A Type-Theoretic Definition of SML

- Elaboration judgements (schematic):

$$\mathcal{E} \vdash exp \Rightarrow \mathrm{e} : \tau$$

- Elaboration context includes typing context.
- IL terms and types serve as static semantic objects.

A Type-Theoretic Definition of SML

- Internal language is derived from HL94 calculus.
  - Extensions to support SML constructs such as references, exceptions.
- Operational semantics employs explicit stack and store.
  - Manage ref's, exceptions, name gen.

Mechanizing the Meta-Theory of SML

- The meta-theory then breaks into two major components:
  - Type safety for HSIL.
  - Elaborated programs are well-typed.
- The hypothesis is that this should make the job more tractable.

Scaling Up To SML

- We are in the process of verifying the meta-theory of this formulation using Twelf.
  - Progress, regularity for the HSIL done.
  - Preservation for the HSIL in progress.
  - Elaboration remains “to do”.

Scaling Up To SML

- We've overcome a few obstacles.
  - explicit store and label management
  - handling variables as sketched earlier
- And uncovered a few bugs in HSIL.
  - Missing rules, missing premises.
  - An unsound typing rule.

Scaling Up To SML

- One sticking point is type equality!
  - Defined type constructors.
  - Type sharing specifications.
- Progress theorem requires inversion.
  - eg, if $A \rightarrow B = A' \rightarrow B'$, then $A=A'$ and $B=B'$.
  - Non-trivial for a "declarative" formulation.

Scaling Up To SML

- Our solution is to use an "algorithmic" formulation of equality.
  - What an implementation would do.
  - Inversion principles are immediate.
- But we leave open whether the algorithmic formulation is equivalent to the declarative.
  - May require techniques beyond Twelf.

Conclusions

- Lots of meta-theory for language definitions can be readily mechanized today.
  - We do this routinely for small-scale languages and logics.
- It's not yet clear whether we can scale up to languages such as SML.
  - No "show stoppers" so far, but we've had to make some compromises.

Conclusions

- A language definition must be formulated with the demands of mechanization in mind.
  - Often good hygiene anyway.
- What we need now are more experiments!
  - Different languages.
  - Different frameworks and tools.

Questions?

- Twelf info:
  www.cs.cmu.edu/~twelf
- POPLMark Challenge:
  www.cis.upenn.edu/proj/plclub/mmm