

Mechanizing the Metatheory of a Language With Linear Resources and Context Effects

Daniel K. Lee
Carnegie Mellon University

Derek Dreyer, Andreas Rossberg
Max Planck Institute for Software Systems

September 20, 2008

RTG Language Overview

- ▶ RTG for *recursive type generativity*.
- ▶ Invented for recursive module calculi [Dreyer, ICFP '05].
- ▶ We formalize a variant used as an IL for mixin modules [Dreyer, Rossberg, ICFP '08].

Key Features of RTG

- ▶ RTG is a module calculus with terms, type constructors, and modules.
- ▶ Key feature of the language is ability to forward declare types and define them later.
 - ▶ The variant we formalize allows for circular definitions.
- ▶ Useful for separately defining two modules whose type components might refer to each other, among other things.

Mechanizing the Metatheory of RTG

- ▶ Used *Twelf* in a formalization and type safety proof for *RTG*.
- ▶ Type definitions in RTG required advanced Twelf encoding techniques.
 - ▶ Utilized a variation on technique for encoding linearity.
 - ▶ Lead to an improvement to *explicit contexts* technique.

RTG Syntax

types $A ::= \dots \mid \alpha$

modules $M ::= \dots \mid \text{new } \alpha.M \mid \text{def } \alpha := A \text{ in } M$

- ▶ $\text{new } \alpha.M$ introduces a binding.
- ▶ $\text{def } \alpha := A \text{ in } M$ defines a variable already in scope.

RTG Type Contexts

typing contexts $\Gamma ::= \cdot \mid \Gamma, \alpha \text{ type}$

definition contexts $\Delta ::= \cdot \mid \Delta, \alpha :=? \mid \Delta, \alpha := A$

- ▶ $\alpha :=?$ is a consumable *definability resource* for α , denoting α is waiting for a definition.
- ▶ $\alpha := A$ is an unrestricted *definition resource* for α , denoting α is equivalent to A .

RTG Typing Rules

$$\frac{\Gamma, \alpha \text{ type}; \Delta, \alpha :=? \vdash M : S \quad \Gamma \vdash S \text{ sig} \quad \alpha \text{ defined once in } M}{\Gamma; \Delta \vdash \text{new } \alpha.M : S}$$

- ▶ Syntactic restriction that type variables created by `new` are only defined once.

RTG Typing Rules

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma; \Delta, \alpha := A \vdash M : S}{\Gamma; \Delta, \alpha := ? \vdash \text{def } \alpha := A \text{ in } M : S}$$

- ▶ Inserting a definition into $\Delta, \alpha := ?$ to yield $\Delta, \alpha := A$ is a context effect.

Roadmap

- ▶ Linearity of definability is enforced with a judgment on syntax.
- ▶ Encoding definition context requires a new form of explicit contexts.
- ▶ Circular definitions complicate metatheory of type equivalence.

Definability as a Linear Resource

- ▶ Type safety requires at most one definition for a variable.
- ▶ Requires variables not be defined more than once.
- ▶ Definability is treated as a linear resource.

Typical Encoding of Linearity

```
md : type.
```

```
...
```

```
linear : (md -> md) -> type.
```

- ▶ `linear ([a] M a)` witnesses `M` is a single hole context.

Typical Encoding of Linearity

```
md/pair : md -> md -> md.    % <M1, M2>
```

```
...
```

```
linear/var      : linear ([m] m).
```

```
linear/pair1    : linear ([m] md/pair (M1 m) M2)  
                  <- linear ([m] M1 m).
```

```
linear/pair2    : linear ([m] md/pair M1 (M2 m))  
                  <- linear ([m] M2 m).
```

- ▶ Enforce linearity by restricting subterm in which variable can appear.

Linear Definability

`defonce` : (tp \rightarrow md) \rightarrow type.

`defzero` : (tp \rightarrow md) \rightarrow type.

- ▶ Linearity is too strict, want define once, not appears once.
- ▶ Use a `defonce` judgment to witness linearity of definitions.
- ▶ Requires a `defzero` judgment to witness absence of definitions.

Linear Definability

```
defonce/pair1    : defonce ([a] md/pair (M1 a) (M2 a))  
                  <- defonce ([a] M1 a)  
                  <- defzero ([a] M2 a).  
defonce/pair2    : defonce ([a] md/pair (M1 a) (M2 a))  
                  <- defzero ([a] M1 a)  
                  <- defonce ([a] M2 a).
```

- ▶ `defonce` defined similarly to `linear`, but using `defzero` instead of syntactic irrelevance.

Roadmap

- ▶ Linearity of definability is enforced with a judgment on syntax.
- ▶ Encoding definition context requires a new form of explicit contexts.
- ▶ Circular definitions complicate metatheory of type equivalence.

Definition Contexts

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma; \Delta, \alpha := A \vdash M : S}{\Gamma; \Delta, \alpha := ? \vdash \text{def } \alpha := A \text{ in } M : S}$$

- ▶ Definability is a *consumable resource*.
- ▶ A definition is an unrestricted resource.
- ▶ Safety requires adding a definition for a variable “consumes” its definability.

Contexts in LF

- ▶ Standard practice is to encode assumptions about variables implicitly using the LF context.
- ▶ Twelf world checking treats all such assumptions as unrestricted, so it cannot enforce
 - ▶ linearity of definability.
 - ▶ uniqueness of definitions.
- ▶ Encoding RTG requires a first order representation of definition context.

Explicit Contexts

- ▶ Advanced Twelf technique to prove theorems not directly provable with HOAS encodings. [Crary, WMM 06, LFMTTP 08]
- ▶ Typically used a proof device, rather than formalization technique [Lee et al., POPL 07].
- ▶ We use explicit contexts as part of our formalism, to encode definition contexts.

Typical Explicit Contexts

```
cxt : type.
```

```
nil  : cxt.
```

```
cons : cxt -> tm -> tp -> cxt.
```

```
% Used with worlds to distinguish variables
```

```
% Also assigns "ordering tokens" to them.
```

```
isvar : tm -> nat -> type.
```

```
% Uses ordering tokens to check cxt is ordered.
```

```
ordered : cxt -> type.
```

```
% Looks up from context.
```

```
lookup : cxt -> tm -> tp -> type.
```

Typical Explicit Contexts Encoding of Typing for STLC

```
ofe/lam : ofe G (lam ([x] E x)) (arr T1 T2)
         <- ({x} isvar x I
             -> ofe (cons G x T1) (E x) T2).
```

```
ofe/var : ofe G E T
         <- ordered G      % isvar E I is implicit
         <- lookup G E T. % via def of ordered
```

Proving Metatheory with Explicit Contexts

- ▶ Proofs with typical explicit contexts may require “freshening” of ordering tokens and re-writing derivations.
 - ▶ Requires induction metrics.

The Reordering Lemma

```
reorder
: ({x} isvar x I -> ofe (G x) (M x) A)
  -> ({x} isvar x I' -> ordered (G x))
%%
  -> ({x} isvar x I' -> ofe (G x) (M x) A)
  -> type.
%mode reorder +D1 +D2 -D3.
```

- ▶ Called “bump” in Crary '08.
- ▶ “ α -vary” ordering token.
- ▶ Necessary to prove weakening.
- ▶ Tedious to prove.

An Improved Explicit Contexts Technique

- ▶ Ordering tokens and context well-formedness assumptions in derivations are a pain.
- ▶ Just remove them!
- ▶ When necessary, pass in and maintain context well-formedness assumptions to metatheorems.
 - ▶ Only need to maintain ordering tokens in metatheorems where context well-formedness matters.
 - ▶ A common paper technique.

Simplified Explicit Contexts

```
% like isvar, but omits ordering token  
var : tm -> type.
```

```
ofe/lam : ofe G (lam ([x] E x)) (arr T1 T2)  
          <- ({x} var x  
              -> ofe (cons G x T1) (E x) T2).
```

```
ofe/var : ofe G E T  
          <- var E  
          <- lookup G E T.
```

Eliminating the Reordering Lemma

```
reorder2
  : ({x} var x -> ofe (G x) (M x) A)
%%
  -> ({x} var x -> ofe (G x) (M x) A)
  -> type.
%mode reorder2 +D1 -D2.
```

- ▶ Eliminating ordering tokens from derivations makes reordering lemma trivial.
- ▶ Simplifies a big annoyance when using explicit contexts.

Encoding Contexts for RTG: A Hybrid Approach

- ▶ We encoded RTG with a hybrid of HOAS and explicit contexts.
- ▶ Use LF context for typing assumptions.
- ▶ Use simplified explicit contexts for definition context.
 - ▶ Assigning a definition to a variable encoded as an operation on explicit representation of definition contexts.

Simplified Explicit Contexts Necessary for Hybrid Approach

- ▶ Proving `reorder` lemma would require explicit contexts for both definition and typing contexts. (painful!)
- ▶ Simplified explicit contexts make hybrid encoding possible.
- ▶ Demands of formalizing RTG led us to discover simplified explicit contexts.

Roadmap

- ▶ Linearity of definability is enforced with a judgment on syntax.
- ▶ Encoding definition context requires a new form of explicit contexts.
- ▶ Circular definitions complicate metatheory of type equivalence.

Proving Type Safety

- ▶ Preservation and progress must account for definitions, but are mostly standard.
- ▶ Hole in development related to injectivity of type equality.

Type Constructor Equality

$$\overline{\Gamma; \Delta, \alpha := A \vdash \alpha \equiv A}$$

- ▶ RTG's has higher order types with definitions [Stone's ATTAPL Chapter]
- ▶ β -, η - equality with higher-kinded type constructors.
- ▶ δ - equality, due to definitions.
- ▶ Circular definitions mean no normal forms.

Injectivity and Inequality

- ▶ Canonical forms lemma require inequality properties:
 - ▶ If $A_1 \rightarrow A_2 \equiv \text{unit}$, then false.
- ▶ Preservation requires injectivity:
 - ▶ If $A_1 \rightarrow A_2 \equiv B_1 \rightarrow B_2$, then $A_1 \equiv B_1$ and $A_2 \equiv B_2$.

Proving injectivity and inequality

- ▶ Injectivity usually proven with normalization-based techniques.
 - ▶ Twelf proofs of injectivity rely on techniques based on notions of *hereditary substitution* that preserve normal forms.
- ▶ Circular definitions rule out normal forms.
- ▶ Injectivity still provable with a logical relations argument.
- ▶ Encoding injectivity proof using Twelf an open question.
 - ▶ Currently assumed using `%trustme`.

Future Work

- ▶ Extend RTG formalization with more features.
- ▶ Formalize an elaboration into RTG.
- ▶ Use other tools to formalize RTG for comparison
 - ▶ A Coq development is in progress.

Conclusion

- ▶ Formalized and proved type safety for a language with a number of non-standard features.
 - ▶ Linear Resources
 - ▶ Context Effects
- ▶ Other interesting issues not discussed in talk:
 - ▶ Actual formalism accounts for more features.
 - ▶ Dealing with adding type variables to environment during evaluation.

Fin

Questions?

A Hybrid Approach

```
cxt : type.
```

```
nil  : cxt.
```

```
% D, a := ?
```

```
nodef : cxt -> tp -> cxt.
```

```
% D, a := A
```

```
def   : cxt -> tp -> tp -> cxt.
```

- ▶ Use HOAS for typing assumptions.
- ▶ Use explicit contexts for definition context.

Linear Definability

```
...
md/defstp : tp -> tp -> md -> md. % a := A in M
...
defzero/defstp : defzero ([a] md/defstp A1 (A2 a) (M a))
                 <- defzero ([a] M a).
...
defonce/defstp1 : defonce ([a] md/defstp a (A a) (M a))
                  <- defzero ([a] M a).
defonce/defstp2 : defonce ([a] md/defstp A1 (A2 a) (M a))
                  <- defonce ([a] M a).
```

- ▶ Enforces a variable is used linearly in definition positions.
- ▶ Allows unrestricted use in other places type constructors can appear.

A Hybrid Approach

$$\frac{\Gamma, \alpha \text{ type}; \Delta, \alpha := A \vdash M : S \quad \Gamma \vdash S \text{ sig}}{\Gamma; \Delta \vdash \text{new } \alpha.M : S}$$

```
of/newtp : of D (md/newtp ([a] M a)) S
  <- ({a} tp-wf a
      -> isdefvar a
      -> of (nodef D a) (M a) S)
  <- defonce ([a] M a).
```

A Hybrid Approach

$$\frac{\Gamma \vdash A_2 \text{ type} \quad \Delta, \alpha_1 := A_2; \Gamma \vdash M : S}{\Delta, \alpha_1 := ?; \Gamma \vdash \text{def } \alpha_1 := A_2 \text{ in } M : S}$$

```
of/deftp : of D (md/deftp A1 A2 M) S
  <- isdefvar A1
  <- tp-wf A2
  <- define D A1 A2 D'
  <- of D' M S.
```