Structural Recursion over Contextual Objects

Sherry Shanshan Ruan
McGill University
Montreal, Quebec, Canada
http://www.cs.mcgill.ca/~sruan2

Brigitte Pientka
McGill University
Montreal, Quebec, Canada
http://www.cs.mcgill.ca/~bpientka

Andreas Abel
Chalmers University of Technology
Gothenburg, Sweden
http://www2.tcs.ifi.lmu.de/~abel

Introduction

Today, we routinely specify and reason about the behavior of software using formal systems such as type systems or logics to establish safety properties. The Logical framework LF is a powerful system for encoding formal systems, but justifying well-founded recursion principles over LF specifications remains a long-standing problem in the area of mechanizing proofs. **Our current contribution is the design of a well-founded induction principle over LF terms, a measure for contextual objects, and a consistency proof based on logical relations.**

Logical Framework (LF)

The logical framework [Harper et al., 1993] supports concise and elegant specifications of formal systems and proofs based on higher-order abstract syntax (HOAS) where we model binders in the object language using binders in LF.

▶ Grammar for an object language

$$term\ M, N := x \mid lam x.M \mid M\ N$$

▶ Formalize the object language in LF

```
datatype term: type =
| lam: (term → term) → term
| app: term → term → term ;
```

- Model variables in the object language by variables in LF
- Model binders in the object language by binders in LF

▶ Correspondence between the object language and the meta-language (LF)

Object language	⇝	Meta-language (LF)
lam x.x	⇝	lam (λx.x)
lam x.lam y.xy	⇝	lam (λx.lam (λy.app x y))

▶ Traverse λ-terms and analyze open terms

lam (λx.lam (λy. *I am a hole. My type is* [x, y ⊢ term].))

- Characterize a hole as a contextual object [g ⊢ M] (M can depend on g)
- The local context g may grow when traversing terms

▶ Contextual LF

A **contextual object** [Nanevski et al., 2008] is an LF object paired with a context where the object may depend on variables in the context.

Beluga [Pientka, 2008, Cave and Pientka, 2012] is a novel language with supports for specifying formal systems in LF and reasoning with contextual objects. Our language is a dependently-typed Beluga combining **contextual types** with a **well-founded primitive recursion principle**.

Example: type uniqueness

▶ Represent types and terms in LF

$$tp\ T, S := bool \mid arr\ T\ S \qquad tm\ M, N := x \mid lam x{:}T.M \mid M\ N$$

```
datatype tp : type =
| bool : tp
| arr : tp → tp → tp ;
```

```
datatype tm : type =
| lam : tp → (tm → tm) → tm
| app : tm → tm → tm ;
```

▶ Represent typing rules in LF

$$\frac{\text{oft M (arr T S)} \qquad \text{oft N T}}{\text{oft (app M N) S}}\ \text{t\_app} \qquad \frac{\dfrac{}{\text{oft x T}}\,u \\ \vdots \\ \text{oft M S}}{\text{oft (lam x:T.M) (arr T S)}}\ \text{t\_lam}^{x,u}$$

Typing Judgment : Γ ⊢ oft M T
Contexts Γ := · | Γ, x, oft x T

Meaning: "Term M has type T in context Γ"

$$\frac{\Gamma \vdash \text{oft M (arr T S)} \qquad \Gamma \vdash \text{oft N T}}{\Gamma \vdash \text{oft (app M N) S}}\ \text{t\_app} \qquad \frac{\Gamma, x, u : \text{oft x T} \vdash \text{oft M S}}{\Gamma \vdash \text{oft (lam x:T.M) (arr T S)}}\ \text{t\_lam}^{x,u}$$

```
datatype oft : tm → tp → type =
| t_app : oft M (arr T S) → oft N T → oft (app M N) S
| t_lam : (Π x:tm. oft x T → oft (M x) S) → oft (lam T M) (arr T S) ;
```

▶ Represent theorems as types in LF

Theorem (type uniqueness): If Γ ⊢ oft E T and Γ ⊢ oft E S, then eq T S

```
(Γ:ctx) [ Γ ⊢ oft (E..) T ]* → [ Γ ⊢ oft (E..) S ] → [ ⊢ eq T S ]
```

where .. denotes the identity substitution.

Programs as proofs

▶ Represent proofs as functions in LF

```
rec unique : (Γ:ctx) [ Γ ⊢ oft (E..) T ]* → [ Γ ⊢ oft (E..) S ] → [ ⊢ eq T S ] =
fn d ⇒ fn f ⇒ case d of
| [ Γ ⊢ t_app (D1..) (D2..) ] ⇒                                  % Application case
    let [ Γ ⊢ t_app (F1..) (F2..) ] = f in
    let [ ⊢ e_ref ] = unique [ Γ ⊢ D1.. ] [ Γ ⊢ F1.. ] in
    [ ⊢ e_ref ]
| [ Γ ⊢ t_lam (λx.λu. D.. x u) ] ⇒                               % Abstraction case
    let [ Γ ⊢ t_lam (λx.λu. F.. x u) ] = f in
    let [ ⊢ e_ref ] = unique [ Γ, b:block x:tm, u:oft x _ ⊢ D .. b.1 b.2 ] [ Γ, b ⊢ F .. b.1 b.2 ] in
    [ ⊢ e_ref ]
| [ Γ ⊢ #q.2.. ] ⇒                                               % Variable case
    let [ Γ ⊢ #r.2 .. ] = f in                                   % d : oft #q.1 T
    [ ⊢ e_ref ]                                                  % f : oft #r.1 S
;
```

▶ A recursive function constitutes an inductive proof

Inductive proofs	⇝	Recursive programs
Apply induction hypotheses	⇝	Make recursive calls
Cases in a proof	⇝	Branches in a function

▶ To justify well-founded recursive functions implement well-founded inductive proofs

→ Programs must be total (Coverage and termination)

→ Recursive data-type definitions must be strictly positive
OR Recursive data-type definitions must be stratified

Contributions

Our core language naturally supports higher-order functions and provides a proof language for first-order logic with contextual LF as a domain.

▶ Theory:

→ Describe a measure for contextual objects (Take account of context weakening and reordering)

→ Design an algorithm for the generation of valid recursive calls

→ Prove weak normalization using logical relations

Weak Normalization Theorem

If a closed term *e* is well-typed, then there exists a finite reduction sequence of *e* (i.e., *e* is weakly normalizing).

▶ Implementation:

→ Simple generation of recursive calls on contextual objects (Support many proofs including type preservation, value soundness, type uniqueness)

→ Positivity checking for recursive data-types

→ Stratification check for recursive data-types

Future work

▶ Theory:

→ Support lexicographical orderings and mutual recursion

→ Generate recursive calls on recursively defined objects

→ Design a recursion principle for more sophisticated recursive calls

▶ Implementation:

→ Generate more sophisticated recursive calls

References

A. Cave and B. Pientka. Programming with binders and indexed data-types. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1): 143–184, January 1993.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL'08*. ACM Press, 2008.