Well-founded Recursion over Contextual Objects

Shanshan (Sherry) Ruan
Computation and Logic Group, School of Computer Science, McGill University
shanshan.ruan@mail.mcgill.ca http://cgi.cs.mcgill.ca/∼sruan2

Introduction

Today, we routinely specify and reason about the runtime behavior of software using formal systems such as type systems 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 and a consistency proof based on logical relations.**

Features of our language

We concentrate our work on **Beluga** [Pientka, 2008], a novel language with supports for specifying formal systems in LF and reasoning with contextual objects, i.e. a LF object paired with a context. Our language is a simply-typed Beluga combining **contextual types** [Nanevski et al., 2008] with a **well-founded primitive recursion principle**.

► Represent abstract syntax of the object language

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

- The object language is a simply-typed fragment $\lambda^{\rightarrow}$
- Model binders in the object language by binders in the meta-language
- Assume the basic computation-level type (int)

Motivating Examples:

- Formalize the object language in LF

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

- Count constructors

cntcons (lam (λx.x)) = 1
cntcons (lam (λx.lam (λy. app x y))) = 3

► Traverse λ-terms and analyze open data

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

► Express the counting function in Beluga-like syntax

```
rec cntcons: (g:ctx)  [g.term] → int =
fn y ⇒ case y of
| [g.#p..] ⇒ 0                                          % base case
| [g.lam λx.M..x] ⇒ 1 + cntcons [g,x:term.M..x]         % step case
| [g.app (M..) (N..)] ⇒ 1 + cntcons [g.M..] + cntcons [g.N..];   % step case
```

► Formal representation of the counting function

- **Challenge**: To generate the appropriate recursive calls
  Solution: f [h,x:term] [h,x.M..x] denotes the valid recursive calls
- **Challenge**: To extend the context when appealing to recursive calls
  Solution: Introduce context variable h: ctx in each branch

```
cntcst = Λg. λy. rec^[g.term]⇐int (y,
    h: ctx, #p: h.term.
        f [h] [h.#p..] ⇒ 0
    h: ctx, M: h,x:term.term. f [h,x:term] [h,x.M..x]: int.
        f [h] [h.lam λx.M..x] ⇒ 1 + f [h,x:term] [h,x.M..x]
    h: ctx, M: h.term, N: h.term. f [h] [h.M..]: int, f [h] [h.N..]: int.
        f [h] [h.app (M..) (N.)] ⇒ 1 + f [h] [h.M..] + f [h] [h.N..]
```

Programs as proofs

► Generalize recursive functions to encode inductive proofs

Proofs depending on assumptions are characterized by contextual objects

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

- Branches must be complete ⇐ Verified by the **splitting & coverage algorithm**
- Programs must terminate ⇐ Verified by the **weak normalization**

These two verifications guarantee the **consistency** of our language.

Splitting & coverage checking algorithm

► Splitting & coverage checking algorithm

split (g ⊢ g.P) = $\overrightarrow{\{ (g,\Delta \vdash M: g.P,\ f\ [h]\ [h.N]) \}}$

Incorporate the algorithm into the type system to perform the coverage checking

► Example: splitting on a contextual object of type *term*

```
split (g ⊢ g.term) = { (g, #p: g.term ⊢ g.#p..: g.term),
                       (g, M: g,x:term.term ⊢ g.lam λx.M..x: g.term, f [g,x:term] [g,x.M..x]),
                       (g, M: g.term, N: g.term ⊢ g.app (M..) (N..): g.term, f [g] [g.M..], f [g] [g.N..]) }
```

► Properties of the algorithm

- We proved the algorithm generates all valid primitive recursive calls.
- We proved the soundness of the algorithm.

Weak Normalization

► Understanding what the weak normalization is about

Weak Normalization Theorem

If a closed term e is well-typed (i.e. $\cdot;\cdot \vdash e : \tau$), then there exists a finite reduction sequence of e (i.e. e is weakly normalizing).

The weak normalization and the determinacy of the evaluation verify the termination.

► Principal idea underlying our proof of the weak normalization

We build the proof by proving the following lemmas together with some auxiliary lemmas, following the classical *logical relations* technique invented by Tait [1967].

If $\cdot;\cdot \vdash e : \tau$, then $e \in \mathcal{R}_\tau$.
(e has type τ infers e is reducible at τ)

If $e \in \mathcal{R}_\tau$, then e is weakly normalizing.
(e is reducible infers e is weakly normalizing)

► Defining Reducibility Candidate $\mathcal{R}_\tau$

We define $\mathcal{R}_\tau$ inductively on type τ, building on the work from Lindley and Stark [2005].

Example: define $\mathcal{R}_{[g.P]}$

$e \in \mathcal{R}_{[g.P]}$ if (1) e is weakly normalizing (2) $\cdot;\cdot \vdash e : [g.P]$
(3) for all $\overrightarrow{b} \in \mathcal{R}_{\Pi h.[h.P]\Leftarrow\tau}$, rec$(e, \overrightarrow{b})$ is weakly normalizing$\}$

Conclusion

Our main contributions towards justifying well-founded induction principle over LF terms:

► Defining a call-by-value small-step semantics

► Designing a sound splitting and coverage checking algorithm

► Proving the weak normalization (the consistency of the simply-typed Beluga)

Future work

► Extending the simply-typed framework to the dependently-typed setting

► Progressing from primitive recursive functions to general recursive functions

► Developing a well-founded recursion principle for context variables

References

S. Lindley and I. Stark. Reducibility and ⊤⊤-lifting for computation types. In *TLCA 2005*, pages 262–277. Springer-Verlag, 2005.

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.

W. Tait. Intensional interpreta

Acknowledgements

I want to express my sincere gratitude to my supervisor, Professor Brigitte Pientka, for her patient supervision, expert guidance, and continued encouragement.