Well-founded Recursion over Contextual Objects

Sherry Shanshan Ruan

School of Computer Science, McGill University

shanshan.ruan@mail.mcgill.ca

Overview

Background

- Today, we reason about the runtime behavior of software using formal systems to establish safety properties
- Logical framework LF is a system for specifying formal system via axioms and inference rules
- In practice, we need to model proof objects which depend on a context of assumptions
- Contextual object $[\Psi.M]$ pairs an open term M together with the context Ψ in which it is meaningful
- Proofs depending on assumptions are characterized by contextual objects

Key challenge

- Inductive proofs are represented with recursive programs
- Justifying that **well-founded recursive programs implement well-founded inductive proofs** is an important milestone towards establishing a framework for programming proofs
- It is a long-standing open problem in the area of mechanizing the reasoning about formal systems [Schurmann, Pfenning'97]
- It has applications in certified programming, proof-carrying architectures, and mechanizing meta-theory of programming languages

A motivating example in Beluga-like syntax

- Beluga is a novel programming environment for reasoning about formal systems specified in the logical framework LF
- Beluga allows embedding contextual LF objects in programs and directly supports common and tricky routines dealing with variables
- We concentrate on simply-typed Beluga at present, but the framework scales to dependently-typed settings
- We present an example which recursively counts the total number of constructors in a term

A motivating example in Beluga-like syntax

Example (counting the occurrences of constructors)

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

schema ctx = term;

rec cnt: (g:ctx) [g.term] -> int =
fn m => case m of
  | [g.#p..] -> 0
  | [g.lam x.M..x] -> 1 + cnt [g,x:term.M..x]
  | [g.app (M..) (N..)] -> 1 + cnt [g.M..] + cnt [g.N..];
```

Formal representation of the counting function

Translating the counting function into our lambda-calculus

- Challenge: Verifying the completeness of case analyses
- Solution: Designing a splitting algorithm to generate all patterns
- Challenge: Extending context when appealing to recursive calls
- Solution: Abstracting over context variables in branches

Example

$\mathsf{cnt} = \Lambda\psi.\ \lambda y.\ \mathsf{rec}^{[\psi.\mathsf{term}]\Leftarrow\mathsf{int}}\ (y,$

ϕ: ctx, $\#p$: ϕ.term.

f $[\phi]$ $[\phi.\#p..]$ $\Rightarrow$ 0

ϕ: ctx, u: ϕ,x:term.term. f $[\phi,x{:}\mathsf{term}]$ $[\phi,x.u..x]$: int.

f $[\phi]$ $[\phi.\mathsf{lam}\ \lambda x.u..x]$ $\Rightarrow$ 1 + f $[\phi,x{:}\mathsf{term}]$ $[\phi,x.u..x]$

ϕ: ctx, u: ϕ.term, v: ϕ.term. f $[\phi]$ $[\phi.u..]$: int, f $[\phi]$ $[\phi.v..]$: int.

f $[\phi]$ $[\phi.\mathsf{app}\ (u..)\ (v..)]$ $\Rightarrow$ 1 + f $[\phi]$ $[\phi.u..]$ + f $[\phi]$ $[\phi.v..]$

Major work

- Correspondence between recursive programs and inductive proofs

Curry-Howard correspondence

Inductive proofs $\leftrightsquigarrow$ Recursive programs

Applying induction hypotheses $\leftrightsquigarrow$ Making recursive calls

Cases in induction $\leftrightsquigarrow$ Branches in function

- To justify well-founded functions implement well-founded inductions

Two verifications

Branches must be complete

$\Leftarrow$ Verified by **coverage checking algorithm**

Programs must terminate

$\Leftarrow$ Verified by **weak normalization theorem**

Coverage checking algorithm

Example: splitting on [g.tm]

split $(g \vdash g.tm) = \{$ $(g,\ \#p{:}g.tm \vdash g.\#p..{:}\ g.tm)$,
$(g,\ M{:}g,x{:}tm.tm \vdash g.lam\ \lambda x.M..x{:}\ g.tm,\ f\ [g,x{:}tm]\ [g,x.M..x])$,
$(g,\ M{:}g.tm,\ N{:}g.tm \vdash g.app\ (M..)\ (N..){:}\ g.tm,\ f\ [g]\ [g.M..],\ f\ [g]\ [g.N..])$ $\}$

- The algorithm splits on a contextual object which can depend on context variables
- The algorithm generates all valid primitive recursive calls for a given contextual type
- We proved the soundness of the algorithm
- We incorporated the algorithm into the type system to perform the coverage checking

Weak normalization proof

Theorem (Weak normalization)

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

- Weak normalization property guarantees that a well-typed program terminates
- We adopted a classical method *logical relations* invented by Tait

If a term M has some type τ, then it is reducible at τ

If a term M is reducible at type τ, then it is weakly normalizing

- Our design of reducibility candidates is inspired by the recent work from Lindley and Stark

Summary of my contribution

My main contributions towards justifying well-founded induction principle over LF specifications are:

- Defined a call-by-value small-step semantics
- Designed a splitting and coverage algorithm and proved its soundness
- Proved weak normalization theorem
  (the consistency of the simply-typed Beluga)

Future work

- Extending the simply-typed framework to dependent types
- Including more general recursive functions
- Covering recursion principle over contexts

The End