An Implementation of Logic Programming Based on the Edinburgh Logical Framework

Mary Southern and Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Midwest Verification Day 2016

Motivation

We are interested in specifications of computational systems for several purposes

- They provide a precise description of the system
- They can be executed and used as a prototype or implementation of the system
- They can be used to support reasoning about the system

In this work we are specifically interested in specifications based on the dependently typed λ-calculus

An Example System

Typing Rules for the Simply Typed λ-Calculus

$$\frac{}{\Gamma_1, x : \tau, \Gamma_2 \vdash x : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 .\ t) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1\ t_2) : \tau_2}$$

Given these rules we can pose questions about whether particular typing judgments hold.

- Does the term $\lambda x : \tau .\ x$ have type $\tau \to \tau$?
- Does the term $\lambda x : \tau .\ y$ have a type?
- Are there any terms of type $\tau \to \tau$?

Specifying the System Using Dependent Types

The system can be formalized in three steps:

1. Describe an encoding of the objects relevant to the system
   - Use expressions of type *ty* to represent (object-language) types
   - Use expressions of type *tm* to represent (object-language) terms

2. Use *dependent types* to capture relationships between these objects.

   ofType : *tm* -> *ty* -> *type*.

3. Identify constants to encode each rule of the system.

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1\ t_2) : \tau_2}$$

app : *ofType* t_1 $(\tau_1 \to \tau_2)$ -> *ofType* t_2 τ_1 -> *ofType* $(t_1\ t_2)$ τ_2.

Using the Specification for Logic Programming

Questions about whether particular typing judgments hold becomes a question about the *inhabitation* of particular dependent types.

- Does the term $\lambda x : \tau.\ x$ have type $\tau \to \tau$?
  Is the type *ofType* $(\lambda x : \tau.\ x)$ $(\tau \to \tau)$ inhabited?
- Does the term $\lambda x : \tau.\ y$ have a type?
  Is there any T such that *ofType* $(\lambda x : \tau.\ y)$ T is inhabited?
- Are there any terms of type $\tau \to \tau$?
  Is there any X such that *ofType* X $(\tau \to \tau)$ is inhabited?

This work aims to provide a *mechanical means* for answering such questions

Animating the Specifications

To support logic programming based on the specifications, we provide a means for answering inhabitation questions

1. We describe a translation of the dependently typed language into an *executable* predicate logic:
   1. Type and term level constants are translated into simply typed constants

ofType : *tm* -> *ty* -> *type*. *ofType* : *LFterm* -> *LFterm* -> *LFtype*.

   2. Next the dependencies are recaptured using formulas

$\forall X$ *hastype X tm* -> $\forall Y$ *hastype Y ty* -> *istype* (*ofType X Y*).

2. We use the *Teyjus* system to solve the logical queries
3. We translate the results in step 2 to yield solutions in the dependently typed setting

The research has to address theoretical questions concerning steps 1 and 3 to make the overall process work

Wrapping Up

We are interested in the efficient animation of dependently typed specifications

Our translation based approach to this problem requires the consideration of two conceptual questions

1. How do we enforce typing constraints in the translation for variables which may be instantiated during search?
2. Can we describe an inverse for the translation of dependently typed terms to simply typed terms?

We are developing a tool based on these ideas which uses the Teyjus system to solve queries