Lambda Calculus and Definability

98-317: Hype for Types

Due: 26 February 2019 at 6:30 PM

Introduction

This week we talked about lambda calculus. In particular, we discussed how any program in any known programming language can be expressed in lambda calculus. In this assignment we'll make this observation concrete by implementing translation from a language that we're convinced is Turing complete to lambda calculus. This homework has only one question in it, which is required.

Turning in the Homework Submit `Translate.sml` to Autolab.

Lambda++ Revisited

We're using the same language we used in the previous homework, Lambda++, but with two modifications:

- We've added a fixed-point operator.
- We've removed the type annotations.

The fixed-point operator makes it easy to write recursive functions. Removing the type annotations makes it easier to construct Lambda++ programs. Here is the syntax of this homework's version of Lambda++:

Expression	e	::=	x	variable
			$\texttt{fn}\ x \Rightarrow e$	lambda
			$e\ e$	application
			$\langle e, e \rangle$	pair
			$\#1\ e$	left projection
			$\#2\ e$	right projection
			$\texttt{INL}\ e$	left injection
			$\texttt{INR}\ e$	right injection
			$\texttt{case}\ e\ \texttt{of INL}\ x \Rightarrow e\ \texttt{|}\ \texttt{INR}\ x \Rightarrow e$	case analysis
			$\texttt{fix}\ x\ \texttt{is}\ e$	fixed point

An intuition for fixed points was presented in lecture. Here is the dynamic semantics rule for the fixed-point operator:

$$\frac{}{\texttt{fix}\ x\ \texttt{is}\ e \longmapsto [\texttt{fix}\ x\ \texttt{is}\ e/x]e}$$

We will give Lambda++ an eager dynamics, like we're used to from languages like SML. The complete dynamic semantics for Lambda++ can be found in the appendix to this homework.

We implement Lambda++'s syntax with the following SML datatype:

```
datatype exp =
  Variable of string
| Lambda of string * exp
| Apply of exp * exp
| Tuple of exp * exp
| First of exp
| Second of exp
| Left of exp
| Right of exp
| Case of exp * (string * exp) * (string * exp)
| Fix of string * exp
```

This datatype, along with a parser, pretty-printer, and dynamic step function for Lambda++ have been implemented in the `LambdaPlusPlus` module.

Lambda Calculus

Here is the syntax we'll use for lambda calculus in this homework. It is simply the syntax of Lambda++ but with most of it removed.

Expression	e	::=	x	variable
			`fn` $x \Rightarrow e$	lambda
			$e\ e$	application

We'll use a lazy dynamics with the lambda calculus because an eager dynamics would make pretty much any interesting lambda calculus program infinitely loop.

We implement lambda calculus's syntax with the following SML datatype, which too is simply that of Lambda++ but with most of it removed:

```
datatype exp =
  Variable of string
| Lambda of string * exp
| Apply of exp * exp
```

This datatype, along with a parser, pretty-printer, and dynamic step function for lambda calculus have been implemented in the `LambdaCalculus` module.

Using the REPLs for these Languages

We've implemented REPLs (read-evaluate-print loops, like you're used to from SML/NJ, coin, or python) for Lambda++ and lambda calculus. After running `sml -m sources.cm` to compile the provided code, use one of the following functions to enter the REPL for Lambda++ or lambda calculus, respectively:

```
Top.lpp_repl : bool -> unit
Top.lc_repl : bool -> unit
```

Use the `bool` argument to specify whether you would like to use interactive evaluation mode. In interactive evaluation mode, every intermediate step of every evaluation is displayed, and you use the `<Enter>` key to progress through the steps. When not in interactive mode, evaluation happens immediately and silently. These REPLs do not typecheck the input, and they evaluate it until no further progress can be made.

There is an additional REPL called `Top.translate_repl` which you can use to test the code you'll write for this homework. This REPL will be described in the next section.

Required

"I've done it!" exclaimed Charles, waving a circuit board in Jeanne's face. "I've built a CPU which runs lambda calculus rather than x86!" He hadn't actually, but wanted to trick Jeanne into thinking he knows how to build hardware.

"Wow! Unfortunately, all the code I've been writing is in Lambda++, not lambda calculus." replied Jeanne, not having fallen for Charles's transparent lie but wishing to humor him so as not to crush his spirits. "Lambda++ is such a feature-rich language, and lambda calculus is such a simple language that I'd be surprised if my code can run on your CPU."

"Not to eavesdrop on a private conversation, but did someone say 'I am would be surprised if lambda calculus was/were as powerful as Lambda++'?" shouted all the students of 98-317: Hype for Types, in unison.

"No, that's not exactly what I said; in fact that sounds like incorrect grammar, but close enough," replied Jeanne.

Chris, who arrived late to the conversation, added "did someone say my name?" No one had.

Required Task In `Translate.sml`, prove Jeanne wrong by implementing the function

```
translate : LambdaPlusPlus.exp -> LambdaCalculus.exp
```

which translates a Lambda++ expression into a lambda calculus expression with the same behavior.

You may want the ability to generate arbitrary new variable names to use in your translated code. We've provided a function `fresh : unit -> string` which returns a string consisting of a % symbol followed by a number which is unique every time you call the `fresh` function. The parser for Lambda++ doesn't allow for variable names that start with the % symbol, so variable names generated by the `fresh` function are guaranteed not to conflict with variable names in the source program.

You may find the `Top.translate_repl` REPL useful to test your code. It expects you to give it Lambda++ code as input, which it runs through your `translate` function then evaluates according to lambda calculus's dynamics. It may be useful to use this REPL in interactive mode so that you can see the direct output of your translation before it evaluates.

Appendix: Dynamics

The dynamics presented here are those used in the REPLs.

Eager dynamics for Lambda++:

$$\frac{}{x\ \mathsf{val}} \qquad \frac{}{\texttt{fn}\ x \Rightarrow e\ \mathsf{val}} \qquad \frac{e_1 \longmapsto e_1'}{e_1\ e_2 \longmapsto e_1'\ e_2} \qquad \frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{e_1\ e_2 \longmapsto e_1\ e_2'}$$

$$\frac{e'\ \mathsf{val}}{(\texttt{fn}\ x \Rightarrow e)\ e' \longmapsto [e'/x]e} \qquad \frac{e_1 \longmapsto e_1'}{\langle e_1, e_2\rangle \longmapsto \langle e_1', e_2\rangle} \qquad \frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\langle e_1, e_2\rangle \longmapsto \langle e_1, e_2'\rangle} \qquad \frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\langle e_1, e_2\rangle\ \mathsf{val}}$$

$$\frac{e \longmapsto e'}{\#1\ e \longmapsto \#1\ e'} \qquad \frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\#1\ \langle e_1, e_2\rangle \longmapsto e_1} \qquad \frac{e \longmapsto e'}{\#2\ e \longmapsto \#2\ e'} \qquad \frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\#2\ \langle e_1, e_2\rangle \longmapsto e_2}$$

$$\frac{e \longmapsto e'}{\texttt{INL}\ e \longmapsto \texttt{INL}\ e'} \qquad \frac{e\ \mathsf{val}}{\texttt{INL}\ e\ \mathsf{val}} \qquad \frac{e \longmapsto e'}{\texttt{INR}\ e \longmapsto \texttt{INR}\ e'} \qquad \frac{e\ \mathsf{val}}{\texttt{INR}\ e\ \mathsf{val}}$$

$$\frac{e \longmapsto e'}{\texttt{case}\ e\ \texttt{of INL}\ x_1 \Rightarrow e_1\ \texttt{|}\ \texttt{INR}\ x_2 \Rightarrow e_2 \longmapsto \texttt{case}\ e'\ \texttt{of INL}\ x_1 \Rightarrow e_1\ \texttt{|}\ \texttt{INR}\ x_2 \Rightarrow e_2}$$

$$\frac{e\ \mathsf{val}}{\texttt{case INL}\ e\ \texttt{of INL}\ x_1 \Rightarrow e_1\ \texttt{|}\ \texttt{INR}\ x_2 \Rightarrow e_2 \longmapsto [e/x_1]e_1}$$

$$\frac{e\ \mathsf{val}}{\texttt{case INR}\ e\ \texttt{of INL}\ x_1 \Rightarrow e_1\ \texttt{|}\ \texttt{INR}\ x_2 \Rightarrow e_2 \longmapsto [e/x_2]e_2}$$

$$\frac{}{\texttt{fix}\ x\ \texttt{is}\ e \longmapsto [\texttt{fix}\ x\ \texttt{is}\ e/x]e}$$

Lazy dynamics for lambda calculus:

$$\frac{}{x\ \mathsf{val}} \qquad \frac{}{\texttt{fn}\ x \Rightarrow e\ \mathsf{val}} \qquad \frac{e_1 \longmapsto e_1'}{e_1\ e_2 \longmapsto e_1'\ e_2} \qquad \frac{}{(\texttt{fn}\ x \Rightarrow e)\ e' \longmapsto [e'/x]e}$$