CS 3MI3: Fundamentals of Programming Languages

Due on Friday, November 10th at 11:59pm EST

Dr. Jacques Carette

Idea

The goals of this assignment are:

1. Learn how to interpret small-step semantics
2. Learn how to translate semantics into an implementation
3. Learn how to identify buggy semantics

Logistics

A project template is available on the course GitHub that includes all of the code from this document, along with a testing framework. You can find it at the following url:

`https://github.com/JacquesCarette/COMPSCI3MI3-F2023/tree/main/Assignments/a3`

The Tasks

1 Revenge of the Goblins and Gnomes [40 points]

The goblins and gnomes of the magical island were very entertained by your riddle solving program. Unfortunately, they got bored of that pretty fast, and will not let you go home unless you can write a program that plays the famous Gnomish game called "SKI".

The rules of "SKI" are as follows: you are given an expression in the following grammar:

$$\begin{array}{l} \langle expr \rangle ::= \mathbf{S} \\ \quad \mid \ \mathbf{K} \\ \quad \mid \ \mathbf{I} \\ \quad \mid \ \langle expr \rangle \ \langle expr \rangle \\ \quad \mid \ (\langle expr \rangle) \end{array}$$

Unparenthesized expressions are left associated, so **SKI** should be parsed as (**SK**)**I**.

To play "SKI", you must apply the following sequence of reduction rules[1]:

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'}$$

$$\frac{}{\mathbf{S}xyz \to xz(yz)} \qquad \frac{}{\mathbf{K}xy \to x} \qquad \frac{}{\mathbf{I}x \to x}$$

Implement a Haskell datatype called `SKI` that corresponds to the provided BNF, and then implement a Haskell function with the following signature that performs a single step of these reduction rules.

[1]Nobody said that gnomes were good game designers.

```haskell
ski :: SKI -> Maybe SKI
```

The function `ski` should return a `Just` containing a reduced value, or `Nothing` if no reductions were possible. Write at least 10 tests using HUnit, making sure that you test all reduction rules, as well as edge cases. Code is worth 15 points, tests are worth 15, and documentation is worth 10. You should write your code in a file called “src/A3/SKI.hs”, and your tests in a file called “test/SKITests.hs”; both of these files are already set up for you in the project skeleton.

2 Loopy Lambdas

One of the major reasons programming-language theorists love the λ-calculus is that it is a good basis for building other programming languages. Here, we will consider an extension of the λ-calculus that adds natural numbers, and a simple looping construct.

$$
\begin{aligned}
\langle \mathit{expr} \rangle ::= &\ \langle \mathit{var} \rangle \\
\mid &\ \lambda\ \langle \mathit{var} \rangle.\ \langle \mathit{expr} \rangle \\
\mid &\ \langle \mathit{expr} \rangle\ \langle \mathit{expr} \rangle \\
\mid &\ 0 \\
\mid &\ 1 +\ \langle \mathit{expr} \rangle \\
\mid &\ \text{loop}\ \langle \mathit{expr} \rangle\ \langle \mathit{expr} \rangle\ \langle \mathit{expr} \rangle \\
\mid &\ (\langle \mathit{expr} \rangle)
\end{aligned}
$$

Intuitively, ‘loop i s f’ denotes a loop that counts down from i to 0, applying f at each step until it terminates with s. However, intuition isn’t enough: we need some sort of formal semantics to specify exactly how our programs evaluate!

2.1 One Small Step for Lambdakind [40 points]

Implement the following operational semantics in Haskell; it should have the signature

```haskell
stepLoop :: Expr -> Maybe Expr
```

$$
\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{}{(\lambda x.e_1)\, e_2 \to e_1[e_2/x]}
$$

$$
\frac{e \to e'}{1 + e \to 1 + e'}
$$

$$
\frac{e_1 \to e_1'}{\text{loop}\ e_1\ e_2\ e_3 \to \text{loop}\ e_1'\ e_2\ e_3} \qquad \frac{}{\text{loop}\ 0\ e_2\ e_3 \to e_2} \qquad \frac{}{\text{loop}\ (1 + e_1)\ e_2\ e_3 \to e_3\,(\text{loop}\ e_1\ e_2\ e_3)}
$$

The `Expr` type can be found in the project skeleton, along with code for checking α-equivalence of terms and performing substitutions.

You should also write at least 25 tests, making sure to cover all constructors, as well as edge cases. At least 3 of these tests should construct your student number. You are allowed (and encouraged!) to use the provided helper functions in your testing code.

Furthermore, both you step function and tests should be documented. The documentation for the step function should make clear what rules you are applying, and the documentation for the tests explain *why* you are testing something. Code is worth 15 points, tests are worth 15, and documentation is worth 10. You should write your code in a file called “src/A3/LoopyLambda.hs”, and your tests in a file called “test/LoopyLambdaTests.hs”; both of these files are already set up for you in the project skeleton.

2.2 Another Small Step for Lambdakind? [20 points]

Consider the following additional reduction rule.

$$\frac{e_2 \to e_2'}{\text{loop } e_1 \ e_2 \ e_3 \to \text{loop } e_1 \ e_2' \ e_3}$$

If it is possible to extend our implementation with this reduction rule, write a function

```
stepLoopExtra :: Expr -> Maybe Expr
```

If it is not possible, explain why. Your answer should take the form of a comment, using the following format:

```
{- [Question 2.2]:
   <your answer here>
-}
```

Submission Requirements

- Must be handed in as a `.zip`, `.gz`, or `.7z` file. Other archive formats will **not** be accepted, resulting in a score of 0. The archive should be called `A3_macemailid.zip` (with your email address, I am "carette", substituted in).
- The name of the file **does** matter.
- Code which **does not compile** is worth **0** marks for the code (including testing) portion of the assignment. Code should be compilable on at least Windows, macOS, and Linux.
- Marks will be deducted if you have junk in your archive (such as object files, `.DS_Store` files, pointless subdirectories, etc.). Stack project files and cabal files are exempt from this.
- If you looked things up online (or in a book) to help, document it in your code. If you have asked a friend for help, document that too. "Looked things up online" includes all AI tools. Put this in a README file (as plain text, Markdown, or HTML).

Bonus

Mechanized Metatheory [Difficulty: ★★]

Encode the loopy lambda calculus in either Agda or Idris 2, and encode the reduction rules as an indexed type. Use this indexed type to prove that $(\lambda x.\, x\, x)(\lambda x.\, x\, x)$ has no normal form.

Include the Agda or Idris file inside of the submitted `.zip` file.

Downhill SKI-ing [Difficulty: ★★★]

Gnomes may be bad game designers, but it turns out that they know a thing or two about theoretical computer science. To start, note that it is possible to interpret the SKI language into the λ-calculus like so[2]:

$$\begin{aligned} S &\mapsto \lambda x.\, \lambda y.\, \lambda z.\, x\, z\, (y\, z) \\ K &\mapsto \lambda x.\, \lambda y.\, x \\ I &\mapsto \lambda x.\, x \end{aligned}$$

In fact, it is also possible to interpret lambda calculus into SKI. Your task is to invent an algorithm for compiling expressions in the untyped λ-calculus into SKI expressions, and to implement this algorithm in Haskell.

This should be submitted as `src/A3/LambdaToSKI.hs`.

[2]One should look at the reduction rules for SKI to convince themselves that this interpretation makes sense!

Self-referential SKI [Difficulty: ★★★★★]

Church-encoding allows us to represent datatypes inside of the untyped λ-calculus. Furthermore, the syntax of untyped λ-calculus is itself a datatype. If we put these two facts together, we can encode the syntax of untyped λ-calculus *inside of itself* via Church-encodings. Use this observation to implement the compiler from λ-calculus into SKI *inside of your implementation* of the untyped λ-calculus. For fun, note that you can use your solution from Bonus 2 to make your program "self-hosting": it is a compiler from untyped λ-calculus to SKI, written only using SKI![3]

This should be submitted as `src/A3/SelfReferenceSKI.hs`. Be warned: this is devilishly hard.

[3]Please do not submit the obfuscated code though: we need to be able to read it to grade it!