Notes on formalizing the STLC

D. Welch

Abstract. Just some notes trying out "LaTeX literate Agda mode" as I work through this next chapter on the lambda calculus (eventually denotational semantics for the untyped version of the calculus). Some of the list chapter exercises are unfinished (including monoids).

1 Intro to the Lambda Calculus

The lambda calculus, first published by Alonzo Church in 1932 is a tiny language with only three syntactic constructs: variables, abstraction, and (function) application.

The *simply typed lambda calculus* (STLC) is a variant of the lambda calculus published in 1940 that adds static typing to the o.g. 1932 untyped lambda calculus. It has the three types of constructions previously mentioned plus additional syntax to support types and type annotations. We use a 'Programmable Computable Function' (PCF) style syntax and add operations for naturals and recursive function definitions.

Specifically we formalize the base constructs that make up the simply-typed lambda calculus: its syntax, small-step semantics, typing, and context formation rules. After this a number of properties of the language such as progress and preservation are stated and proven. The notes may extend the language with additional features such as records.

Note: these notes do not present a recommended approach to formalization as they eschew a locally nameless representation of terms (via e.g. DeBruijn indices). A later section of these notes might look into this.

1.1 Imports

First, we'll need some imports:

```
module stlc where
open import Data.Bool using (Bool; true; false; T; not)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.List using (List; _::_; [])
open import Data.Nat using (ℕ; zero; suc)
open import Data.Product using (∃-syntax; _×_)
open import Data.String using (String; _≟_)
open import Data.Unit using (tt)
open import Relation.Nullary using (Dec; yes; no; ¬_)
open import Relation.Nullary.Decidable using (False; toWitnessFalse)
open import Relation.Nullary.Negation using (¬?)
open import Relation.Binary.PropositionalEquality using (_≡_; _≢_; refl)
```

2 Term Syntax

Terms have seven constructs. Three are for the core lambda calculus:

- variables $`x$
- abstractions $\lambda x\ :\ T$
- applications $L \cdot M$

Three are for the naturals, ℕ:

- zero $`zero$
- successor $`suc\ M$
- case expressions $case\ L\ [zero \Rightarrow M \mid suc\ x \Rightarrow N]$

There is also one for recursion:

- fixpoint $\mu\ x \Rightarrow M$

Abstraction is also called lambda abstraction, and is the construct from which the calculus takes its name.

With the exception of variables and fixpoints, each term form either constructs a value of a given type (abstractions yield functions, zero and successor yield natural numbers) or deconstructs it (applications use functions, case expressions use naturals). This will arise again when we fix rules for assigning types to terms – where ctors correspond to intro. rules and deconstructors, eliminators.

Here is the syntax of terms in Backus-Naur Form (BNF):

$$L,\ M,\ N ::= `x \mid `\lambda x \Rightarrow N \mid L \cdot M \mid$$
$$`zero \mid `suc\ M \mid case\ L[zero \Rightarrow M|suc\ x \Rightarrow N] \mid \mu\ x \Rightarrow M$$

And here it is formalized in Agda:

```
Id : Set
Id = String

infix 5 ƛ_⇒_
infix 5 μ_⇒_
infixl 7 _·_
infix 8 `suc infix 9 `_

data Term : Set where
```

```
`_                        : Id → Term
ƛ_⇒_                      : Id → Term → Term
_·_                       : Term → Term → Term
`zero                     : Term → Term
case_[zero⇒_—suc_⇒_] : Term → Term → Id → Term → Term
μ_⇒_                      : Id → Term → Term
```