Small step semantics for the STLC

Christian Kjær Larsen

September 10, 2022

1 Changes for adding float

We start by extending the base syntax with decimal literals m $(1.0, 42.42, -3.14, \ldots)$ with a decimal dot. Types and value are extended to be

$$\tau ::= \ldots \mid \textbf{float} \qquad v ::= \ldots \mid m$$

We add new syntactic constructs to the language for casts and floating point literals.

$$e ::= \ldots \mid m \mid \textbf{float}(e) \mid \textbf{int}(e)$$

and we add new typing rules

$$\textsc{T-Float}\frac{}{\Gamma \vdash m : \textbf{float}} \qquad \textsc{T-ToFloat}\frac{\Gamma \vdash e : \textbf{int}}{\Gamma \vdash \textbf{float}(e) : \textbf{float}} \qquad \textsc{T-ToInt}\frac{\Gamma \vdash e : \textbf{float}}{\Gamma \vdash \textbf{int}(e) : \textbf{int}}$$

and new reduction rules

$$\textsc{Add-F}\frac{}{m_1 + m_2 \to m}m = m_1 + m_2 \qquad \textsc{Int}\frac{}{\textbf{int}(m) \to n}n = \lfloor m \rfloor \qquad \textsc{Float}\frac{}{\textbf{float}(n) \to m}m = n$$

And context rules

$$E ::= \ldots \mid \textbf{int}(E) \mid \textbf{float}(E)$$

2 Implementation

Included is a Scala 3 implementation of the lambda calculus with the above changes.

The abstract syntax is implemented using Scala 3 enums. A trait is used to encode the values. We have a type parameter A for annotations that are used for tracking source locations.

```
enum Ty { ... }
type Name = String
sealed trait Value
enum Stlc[A] {
  case Lam(x: Name, t: Ty, body: Stlc[A], a: A) extends Stlc[A] with Value
  ...
}
```

In the program we can then use intersection types to require that a term is a value like

```
def foo[A](v: Stlc[A] & Value): ...
```

2.1 Type checking

We include two functions. One for checking that a term has one of the required types and one for inferring the type of a term.

```
type LStlc = Stlc[SourceLocation]
def check(term: LStlc, ctx: Map[Name, Ty], ts: Set[Ty]): Either[TypeError, Ty]
def infer(term: LStlc, ctx: Map[Name, Ty]): Either[TypeError, Ty]
```

This is a trivial example of bidirectional type checking.

2.2 Reduction

We implement small step reduction with two functions. One for substituting a value for a variable, and one for taking a small step reduction. The stepping function will return nothing if we are stuck or if we have reached a value.

```
def substValue[A](x: Name, e: Stlc[A], v: Stlc[A] & Value): Stlc[A]
def stepCBV[A](term: Stlc[A]): Option[Stlc[A]]
```

3 Running the interpreter

To make the casts a bit more ergonomic and to avoid changing the parser we add them to the ”standard library”. This is done by type checking in a context

```
val stdlib = Map(
  "int" -> Ty.Arrow(Ty.Float, Ty.Int),
  "float" -> Ty.Arrow(Ty.Int, Ty.Float)
)
```

and we have small step rules to reduce these functions to the η-expansion of the built-in casts.

```
def stepCBV[A](term: Stlc[A]): Option[Stlc[A]] = {
  ...
  case Var("float", a) => Some(Lam("x", Ty.Int, ToFloat(Var("x", a), a), a))
  case Var("int", loc) => Some(Lam("x", Ty.Float, ToInt(Var("x", a), a), a))
```

This makes sure that we can treat the casts as regular functions.

How to run the project can be found in the README.md file included with the source code.