A typed λ-calculus, TL

Principles of Programming Languages

Mark Armstrong

Fall 2020

1 Preamble

1.1 Notable references

- Benjamin Pierce, "Types and Programming Languages"
  - Chapter 9, Simply Typed Lambda-Calculus
    - Function types, the typing relation
  - Chapter 11, Simple Extensions
    - Unit, Tuples, Sums, Variants, Lists.

1.2 TODO Table of contents

2 Introduction

In this section we extend our previously considered untyped λ-calculus by defining a typing relation, essentially adding type checking (enforcement).

We then investigate adding some algebraic type formers to the language. This involves the introduction of a rudimentary form of pattern matching.

3 Recall: The untyped λ-calculus

Recall from section 3 of the notes the syntax of our untyped λ-calculus, *UL*.

⟨`term`⟩ ::= `var` | λ `var` $\rightarrow$ ⟨`term`⟩ | ⟨`term`⟩ ⟨`term`⟩

Recall that in this pure untyped λ-calculus, everything is a function, and abstractions (terms of the form λ `x` $\rightarrow$ `t`) are *values*.

3.1 The call-by-value semantics of the untyped λ-calculus

The call-by-value semantics we described in section 3 of the notes can be more succinctly described using inference rules.

- In fact, we only need three rules.
- Here the arrow $\longrightarrow$ defines a *reduction* relation, meaning that we may need to perform several $\longrightarrow$ "steps" to fully evaluate a term.
- The (meta)variables $\mathtt{t}_1$, $\mathtt{t}_2$, etc., range over λ-terms, and
- the (meta)variables $\mathtt{v}_1$, $\mathtt{v}_2$, etc., range over λ-terms *which are values*.

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1\ \mathtt{t}_2}\ \texttt{reduce-app}^l$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2}\ \texttt{reduce-App}^r$$

$$\frac{}{(\lambda\ \mathtt{x} \to \mathtt{t})\ \mathtt{v} \longrightarrow \mathtt{t}[\mathtt{x} \coloneqq \mathtt{v}]}\ \texttt{apply}$$

3.2 Only applications reduce

Notice, in the above semantics, that the only rules are for applications; remember that

- variables cannot be reduced, and
- under call-by-value semantics,
  - no evaluations take place inside abstractions, and
  - abstractions are only applied to values.

3.3 Explaining the rules

By using our naming conventions, we can see that

- the `reduce-app`^{l} rule says that if $\texttt{t}_1$ is the left side of an application and $\texttt{t}_1$ reduces to $\texttt{t}_1$, then the whole application reduces by replacing $\texttt{t}_1$ with $\texttt{t}_1$,
- the `reduce-app`^{r} rule says that if $\texttt{t}_1$ is the right side of an application *whose left side is a value*, and $\texttt{t}_2$ reduces to $\texttt{t}_2$, then the whole application reduces by replacing $\texttt{t}_2$ with $\texttt{t}_2$, and
- the `apply` rule says that if the left side of an application is an abstraction, and the right side is a value, then the application reduces to the body of the abstraction with the value substituted for the abstraction's variable.

3.4 Reduction as a function

It bears noting that the *reduction relation* here is, by design, *deterministic*; given a λ-term `t`, either

- `t` can be reduced by exactly *one* of the rules above, or
- `t` cannot be reduced (is irreducible) (by these semantics.)

A deterministic relation can be expressed as a *function*, as the following Scala-like pseudocode shows.

```
def ⟶(t) = t match {
  case t₁ t₂ if t₁ ⟶ t₁                  => t₁  t₂
  case v₁ t₂ if isValue(v₁) &&  t₂ ⟶ t₂  => v₁ t₂
  case (λ x → t) v if isValue(v)          => t[x ≔ v]
}
```

3.5 An example of a reduction sequence

```
  ((λ x → x) (λ y → y)) ((λ z → z) (λ u → u))
⟶⟨ reduce-app^l ⟩
  (λ y → y) ((λ z → z) (λ u → u))
⟶⟨ reduce-app^r ⟩
  (λ y → y) (λ u → u)
⟶⟨ apply ⟩
  λ u → u
```

The final term does not reduce.

Note that we can end with terms which do not reduce, but which are not values, such as

```
(λ x → x) y
```

Since free variables are not values (they are not λ-abstractions), this term does not fit any of the reduction rules.

3.6 Encodings of booleans, natural numbers and pairs

Recall the λ-encodings discussed in notes section 3, which allow us to represent booleans, natural numbers and pairs in the pure untyped λ-calculus.

```
tru  = λ t → λ f → t
fls  = λ t → λ f → f
test = λ l → λ m → λ n → l m n
pair = λ f → λ s → λ b → b f s
fst  = λ p → p tru
snd  = λ p → p fls
zero = λ s → λ z → z
scc  = λ n → λ s → λ z → s (n s z)
```

3.7 Enriching the (syntax of the) calculus

While λ-encodings of data in the pure untyped λ-calculus, such as those for the booleans, natural numbers and pairs, do allow us to construct programs working on any type data we might like, it is usually more convenient (even in this untyped system) to instead *enrich* the calculus with new primitive terms for the types we want to work with.

We will show here how this can be done for booleans. The enriched calculus's syntax is then

```
⟨term⟩ ::= var | λ var → ⟨term⟩ | ⟨term⟩ ⟨term⟩
         | true | false
         | if ⟨term⟩ then ⟨term⟩ else ⟨term⟩
```

3.8 The semantics of the extended calculus

:TODO:

4 The simply typed λ-calculus

:TODO:

5 "Simple extensions" to the simply typed λ-calculus

:TODO: