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*.

$\langle\texttt{term}\rangle$ ::= `var` | λ `var` $\rightarrow$ $\langle\texttt{term}\rangle$ | $\langle\texttt{term}\rangle$ $\langle\texttt{term}\rangle$

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.*

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

$$\frac{\mathtt{t}_2 \quad \longrightarrow \quad \mathtt{t}_2}{\mathtt{v}_1 \ \mathtt{t}_2 \quad \longrightarrow \quad \mathtt{v}_1 \ \mathtt{t}_2} \ \mathtt{reduce\text{-}app}^{r}$$

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

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 `t`$_1$ is the left side of an application and `t`$_1$ reduces to `t`$_1$, then the whole application reduces by replacing `t`$_1$ with `t`$_1$,
- the `reduce-app`^{r} rule says that if `t`$_1$ is the right side of an application *whose left side is a value*, and `t`$_2$ reduces to `t`$_2$, then the whole application reduces by replacing `t`$_2$ with `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-appˡ ⟩
  (λ y → y) ((λ z → z) (λ u → u))
⟶⟨ reduce-appʳ ⟩
  (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 untyped λ-calculus with booleans

The untyped λ-calculus extended with booleans semantics has, in addition to the rules $\texttt{reduce-app}^{l}$, $\texttt{reduce-app}^{r}$ and `apply`, these rules for the new basic primitive functions.

$$\frac{\mathtt{t}^{b} \;\longrightarrow\; \mathtt{t}^{b}}{\mathtt{if}\ \mathtt{t}^{b}\ \mathtt{t}_1\ \mathtt{t}_2 \;\longrightarrow\; \mathtt{if}\ \mathtt{t}^{b}\ \mathtt{t}_1\ \mathtt{t}_2}\ \texttt{reduce-if}$$

$$\frac{}{\texttt{if true t}_1\ \texttt{t}_2 \longrightarrow \texttt{t}_1}\ \texttt{if-then}$$

$$\frac{}{\texttt{if false t}_1\ \texttt{t}_2 \longrightarrow \texttt{t}_2}\ \texttt{if-else}$$

4 The typed λ-calculus

We define here the syntax and semantics for several stages of a typed λ-calculus.

- We begin with a "simply-typed" λ-calculus that has only unit and function types, and
- at each stage, we add new primitive terms, new types and typing rules, and new semantic rules.

These stages roughly correspond to those given in Pierce's "Types and Programming Languages" throughout chapters

- 9, "Simply Typed Lambda-Calculus", and
- 11, "Simple Extensions".

For the sake of page space, each stage will only show the grammar productions and semantic rule which are added, not the whole grammar or semantics.

- Those will be given at the end.

All semantics in this section are call-by-value semantics.

4.1 Typing rules

Like semantics, the typing rules of a language are presented here using inference rules.

These inference rules define a typing relation, written `_`$\vdash$`_:_` and read as "entails".

While the reduction relation, `_`$\longrightarrow$`_`, is a binary relation between terms

- i.e., `_⟶_ : term × term`
  - (in fact, since it is a single-valued relation, `_⟶_ : term` $\to$ `term`),

the typing relation is a *ternary* relation between a *typing context*, a term and a type.

- i.e., `_⊢_:_ : context × term × type`
  - (in fact, since it is also a single-valued relation, `_⟶_ : context × term` $\to$ `type`.)

The *typing context* referred to above is a set of variable, type pairs, used to *bind* certain variables to types.

- It can in fact be a sequence or similar datatype; so long as we can add and check bindings.

4.2 The simply-typed λ-calculus syntax

Our starting point is the simply-typed λ-calculus, which has only unit and function types.

- For the sake of noting which new terms are values, we add a non-terminal called ⟨`value`⟩ to the grammar.

```
⟨term⟩ ::= var
         | ⟨term⟩ ⟨term⟩
         | ⟨value⟩

⟨value⟩ ::= λ var : ⟨type⟩ → ⟨term⟩
          | unit

⟨type⟩ ::= Unit | ⟨type⟩ → ⟨type⟩
```

4.3 The simply-typed λ-calculus typing

"If a variable `x` is assigned type `A` by the context, then it has that type."

$$\frac{\mathtt{x} : \mathtt{A} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{A}}\ \texttt{T-Var}$$

Notice that otherwise, variables do not typecheck!

"The abstraction of a variable `x` of type `A` over a term `t` has type `A` $\to$ `B` if `t` has type `B` when assuming `x` has type `A`."

$$\dfrac{\Gamma\texttt{,(x : A)} \vdash \texttt{t : B}}{\Gamma \vdash \texttt{(}\lambda\texttt{ x : A} \to \texttt{t) : A} \to \texttt{B}}\ \texttt{T-Abs}$$

"If $\texttt{t}_1$ has type `A` $\to$ `B` and $\texttt{t}_2$ type `A`, then applying $\texttt{t}_1$ to $\texttt{t}_2$ has type `B`."

$$\dfrac{\Gamma \vdash \texttt{t}_1 \texttt{ : A} \to \texttt{B} \qquad \Gamma \vdash \texttt{t}_2 \texttt{ : A}}{\Gamma \vdash \texttt{t}_1\ \texttt{t}_2 \texttt{ : B}}\ \texttt{T-App}$$

"`unit` has type `Unit`."

$$\dfrac{}{\Gamma \vdash \texttt{unit : Unit}}\ \texttt{T-Unit}$$

4.4 The simply-typed λ-calculus semantics

The semantics of the language have not changed, except that the syntax of the λ-abstraction now has the type annotation.

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

$$\dfrac{\texttt{t}_2 \longrightarrow \texttt{t}_2}{\texttt{v}_1\ \texttt{t}_2 \longrightarrow \texttt{v}_1\ \texttt{t}_2}\ \texttt{reduce-app}^r$$

$$\dfrac{}{\texttt{(}\lambda\texttt{ x : A} \to \texttt{t) v} \longrightarrow \texttt{t[x} \coloneqq \texttt{v]}}\ \texttt{apply}$$

4.5 Exercise: Why do we need a `Unit` type in the simply-typed λ-calculus?

Recall that in the untyped λ-calculus, the only values were abstractions; all data was functions.

Why do we add a `Unit` type in the simply-typed λ-calculus? Is it required for some reason?

4.6 TODO More types...