Prof. Dr. Peter Thiemann
Winter Term 2024/25

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 11

In this exercise sheet, we are going to extend the lambda calculus interpreter from the previous exercise sheet with built-in constants and operations for integers and booleans, and a type system with a type inference algorithm.

Exercise 1 (Extended Interpreter)

In this exercise, we are going to extend the lambda calculus interpreter with integer and booleans, but do not yet add a type system.

To be precise, we are going to add integer constants, boolean constants, addition, inequality, and conjunction. We can represent the abstract syntax tree of our language for example as

```
type Var = String

data Op2 = OAdd
         | OLess
         | OAnd
         deriving (Show, Eq)

data Exp = EVar Var
         | ELam Var Exp
         | EApp Exp Exp
         | EInt Int
         | EBool Bool
         | EOp2 Exp Op2 Exp
         deriving (Show, Eq)
```

We recommend to proceed in the following steps:

1. Extend the parser to support the parsing of the new language constructs.

   For simplicity, we recommend to to side-step operator precendence and binding strength, and simply require explicit parentheses around operator applications, e.g. that `x + 2 < 5` is written as `((x + 2) < 5)`.

   Care needs to be taken to avoid ambiguity with booleans and variables. Previously, we said everything is a variable that matches the regex `[a-zA-Z]+`. However, this would allow `true` to be parsed both as a boolean constant and as a variable with name `true`. To fix this we want to restrict the variable parser to only allow names which are neither `true` nor `false`. A simple way to achieve this is via `pVar >>= f`, where `pVar` is the previous variable parser and `f` is a function which lets the parser fail if `pVar` parsed `"true"` or `"false"`.

2. Extend the `free` and substitution function to also cover the new language constructs. This should be straightforward, because both of those functions are only concerned with variables, which our new language constructs do not use directly.

3. Extend the `step` function with the reduction rules for the new language constructs. In particular, we have new β-reduction rules, which state how introduction forms (like integer and boolean constants) interact with elimination forms (like operators), and congruency-rules, which state how reduction can be propagated into subexpressions.

   We have one β-reduction rule for each operator, e.g. an addition where the operands are integer constants `EOp2 (EInt n1) EAdd (EInt n2)` reduces to `EInt (n1 + n2)`.

   We have two congruency rules for each binary operator, one which propagates reduction into the left subexpression, and on which propagates reduction into the right subexpression. For example for addition we have

$$\frac{e_1 \to e_1'}{(e_1 + e_2) \to (e_1' + e_2)} \qquad \frac{e_2 \to e_2'}{(e_1 + e_2) \to (e_1 + e_2')}$$

4. Extend the `pretty` function to work with the new language constructs.

Exercise 2 (Type System & Type Inference)

By extending the language with non-function-values (like integers), we unfortunately gain the possibility to write bad programs, e.g. the program `8 3` tries to treat the integer `8` as a function by applying it to another integer `3`. As `8` is not a function, the program `8 3` does neither reduce nor is it a value – the program is stuck. Stuck programs correspond to runtime errors.

In this exercise, we add a type system to our interpreter, which statically rules out all programs, which might get stuck during reduction. Hence, a program that type-checks is guaranteed to not cause any runtime errors.

To avoid having type annotations in our programs, we use the type inference algorithm that was introduced in the lecture.

An interpreter for a typed language first parses the program, then type checks it, and then runs it. Typically, if the program does not type check, then it is also not run. However, since we are writing this interpreter for educational purposes, we are going to run it also if it does not type check such that we can see where it gets stuck.

For example, we want the interpreter to behave as followed:

```
lambda> (\x. (x + 1))
Type:
  (Int -> Int)
Reduction:
  (\x. (x + 1))
Reduced to value.

lambda> ((\x. (x + 1)) 5)
Type:
  Int
Reduction:
  ((\x. (x + 1)) 5)
  6
Reduced to value.

lambda> ((\x. (x + 1)) true)
Type:
  ERROR: Cannot unify Int and Bool!
Reduction:
  ((\x. (x + 1)) true)
  (true + 1)
Reduction stuck!
```

```
lambda> (x + x)
Type:
  ERROR: Undefined variable x of type Int
Reduction:
  (x + x)
Reduction stuck!

lambda> (\x. x)
Type:
  ERROR: Expression has type (alpha:1 -> alpha:1).
         This type is ambigous as it still contains type variables.
         To support this expression, we would need proper polymorphism
         in our type system.
Reduction:
  (\x. x)
Reduced to value.
```

To implement type inference, we first need to define the syntax of types:

```
data Type = TInt
          | TBool
          | TFun Type Type
          | TVar Var
          deriving (Eq, Show)
```

The constructors `TInt`, `TBool`, and `TFun` correspond to the actual types of the lambda calculus extended with booleans and integers. The `TVar` type is an artifact of the type inference algorithm, which temporarily needs to introduce type variables for parts of the types, which are not yet known. If type inference suceeds then the resulting type does not contain any type variables anymore (since we are implementing a simple type system without polymorphism).

To extend our interpreter repl with a type checker, we ultimately need a single function

```
infer :: Expr -> Either TypeError Type
```

which given an expression tries to infer the type of the expression or fails with a type error. We can then call this function in the `repl` function after parsing but before reduction.

We recommend to implement `infer` in terms of a more general `infer'` function of type:

```
infer' :: (MonadState Int m, MonadError TypeError m) -> Expr -> m (Ctx, Type)
```

The `infer'` function should implement the type inference algorithm discussed in the lecture: given an expression, it tries to infer a type and the set of typing assumptions (called a `Ctx` here). The computation to infer a type can have two side-effects: it can fail with a `TypeError`, and it can manipulate a state of type `Int`. The state of type `Int` allows to conveniently create fresh type variables by incrementing the state, e.g. we can define a function

```
freshVar :: MonadState Int m => m Var
```

such that when we start with a state of `0`, the first call of `freshVar` returns `"alpha:0"`, the second call returns `"alpha:1"`, and so on.

The list of typing assumptions (often called a *typing context*) can be defined as a map from variables to types, e.g.

```
type Ctx = [(Var, Type)]
```

The `infer` function should call `infer'` with an integer state `0` and ensure that the returned `Ctx` is empty. A non-empty context would mean that the program contains undefined variables.

Here are some examples of how `infer'` should behave:

```
run :: StateT Int (Either TypeError) a -> Either TypeError a
run mx = evalStateT mx (0 :: Int)

>>> run (infer' (EInt 2))
Right ([], TInt)

>>> run (infer' (EOp2 (EVar "x") OAdd (EInt 2)))
Right ([("x", TInt)], TInt)

>>> run (infer' (EVar "x"))
Right ([("x", TVar "alpha:0")], TVar "alpha:0")

>>> run (infer' (ELam "x" (EVar "x")))
Right ([("x", TInt)], TInt)

>>> run (infer' (ELam "x" (EVar "x")))
Right ([], TFun (TVar "alpha:0") (TVar "alpha:0"))

>>> run (infer' (EApp (ELam "x" (EVar "x")) (EBool True)))
Right ([], TBool)
```

The meaning of the first example is that the expression `2` has type `Int` without making any assumptions about the types of variables.

The meaning of the second example is that the expression `x + 2` has type `Int` assuming that `x` is a variable of type `Int`.

To replace type variables, we need to define a substitution operation, similar as we did for expression variables in the previous exercise. In contrast to the previous exercise, we use a substitution operation which replaces multiple variables at the same time:

```
type Sub = [(Var, Type)]
subst :: Sub -> Type -> Type
```

For example:

```
>>> subst [("a", TInt), ("b", TBool)] (TFun (TVar "a") (TVar "b"))
TFun TInt TBool
```

Note that this substitution is much simpler than on expressions, because there are no annoying lambda terms which can bind variables.

A substitution, which makes two types t_1 and t_2 equal is called a *unifer of* t_1 *and* t_2. For example, the types `TFun (TVar "a") TInt` and `TFun TBool TInt` can be made equal by the substitution `[("a", TBool)]`, so `[("a", TBool)]` is a unifier of those types.

The *most general unifier of two types* is a unifier which leaves the most degrees of freedom (for a formal definition see the slides). For example, the types `TVar "a"` and `TVar "b"` are unified by `u1 = [("a", TInt), ("b", TInt)]` and `u2 = [("a", TBool), ("b", TBool)]`. However, neither of these unifiers are most general. A most general unifier is `mgu = [("b", TVar "a")]`,

because it does not needlessly commit to `TInt` or `TBool`, and the effects of `u1` and `u2` can still be achieved after unifying the types with `mgu` by doing another substitution.

Sometimes, we need to compute a substitution which does is not only the most general unifier of a single pair of types, but for multiple pairs of types.

We recommend to define these functions mutually recursive, i.e.

```
mgu1 :: MonadError TypeError m => Type -> Type -> m Sub
mgu  :: MonadError TypeError m => [(Type, Type)] -> m Sub
```

An `mgu` for multiple type pairs is basically the concatenation of the `mgu1` of each individual type pair. However, in case multiple type pairs contain the same type variables, we need to take care that the unifiers do not replace them with different types. This can be achieved by first computing the `mgu1` for the next pair and then applying it to all remaining pairs before computing their `mgu`. This forces the `mgu1` for the remaining pairs to only add new replacements.

The type inference algorithm from the slides, does not only compute the `mgu` of types but also of type contexts by abuse of notation. In the implementation, this corresponds to a function which converts two type contexts to a list of type pairs:

```
ctxEqPairs :: Ctx -> Ctx -> [(Type, Type)]
```

This function should return all pairs of types for the same variables, e.g.

```
>>> ctxEqPairs [("x", t1), ("y", t2), ("z", t3)] [("x", t1'), ("z", t3')]
[(t1, t1'), (t3, t3')]
```

In other words: to unify two type contexts, all types of variables, which appear in both contexts, need to be unified pairwise.