Concepts of Programming Languages

Course: CSE2120 Edition: 2019-2020

Available from January 26, 2020 until July 3, 2020

Your Enrollment

You're taking this course for a grade

Your Course Dossier

Your Submissions

Unenroll

Course Information

Home

All editions

News archive

Course rules

Lecture notes

Assignments

Course staff

Lecturers

- Casper Poulsen
- Eelco Visser

Assistants

- Ali Al-Kaswan
- Yana Angelova
- Wesley Baartman
- Kirti Bihare
- Philippos Boon Alexaki
- Luc Everse
- Boris Janssen
- Rembrandt Klazinga
- Mirco Kroon
- Chris Lemaire
- Sterre Lutz
- Yanv Oren
- Wouter Polet
- Thijs Raijmakers
- Jim van Vliet
- Yoshi van den Akker
- Paul van der Stel
- Eric van der Toorn

1.9. Week 8.5: Type Inference (Challenge)

In this optional extra assignment you should implement type inference for an eager variant of the Paret language with lists. As opposed to the language you wrote a type checker for in week 6, there are no longer type annotations on functions or `nil`s, because you will be inferring types.

Please note that this is a challenging assignment. If you are up for an extra challenge and you like the course or if you are interested in the topic of type inference, then this is the assignment for you. **There are no hints for this assignment**, but of course you are still free to ask questions to our TAs.

Types

Unlike in the type checker assignment, we can now have types with type variables in them, when the type of an expression is not fully constrained.

The type inference algorithm is not particularly dependent on what kind of types you have in your language, so it makes sense that we pick a generic representation that can easily be extended later. Thus the `Type` data definition has just two forms:

```
sealed abstract class Type
case class TVar(tyVar: String) extends Type                  // Type variable
case class TCon(con: TConstructor, fields: List[Type]) extends Type  // Concrete type

sealed abstract class TConstructor
case class NumTC()  extends TConstructor
case class BoolTC() extends TConstructor
case class FunTC()  extends TConstructor
case class ListTC() extends TConstructor
```

We have provided you with a `typeOf` function that takes an expression `expr` and produces its inferred `Type`. It requires you to implement three functions: `generate`, `unify` and `lookup`.

```
def typeOf(expr: ExprExt, env: TEnvironment): Type = {
  val v = freshTVar()
  val cs = generate(expr, v, env)
  val subs = unify(cs, Nil)
  lookup(subs, v)
}
```

We also defined helper functions `numT()`, `boolT()`, `funT()` and `listT()` to work like the constructors for `Type` from the type checker assignment, so you can more easily port your test cases. But keep in mind you cannot pattern match with these functions using `case` since they are not real case classes. For example, `funT(listT(numT()), boolT())` evaluates to:

```
TCon(FunTC(), List(TCon(NumTC(), List()), TCon(BoolTC(), List())))
```

And finally we give you a function `prettyprintType()` that you can use while debugging to get a more readable string representation of a type. For example, the above type would be represented as:

```
((Num) -> Bool)
```

Constraints

The first step in type inference is to generate a set of type equality constraints for all the expressions in the program by implementing the `generate` function. This function takes an expression `expr`, its expected type `t_expr`, and a type environment `tenv`, and produces a set of type equality constraints as a list.

```
def generate(expr: ExprExt, t_expr: Type, tenv: TEnvironment): List[TEq] = // ...
```

In order to enumerate the relationships between the types of the expressions, you can associate a unique *type variable* with an expression. New unique type variables can be generated with `freshTVar()`.

You will also need to associate identifiers with their type. For this purpose, you can use a `TEnvironment` during constraint generation to map from an identifier name (e.g. `x`) to a type variable (made using `freshTVar()`) or concrete type:

```
type TEnvironment = List[TBind]

case class TBind(name: String, ty: Type)
```

A constraint equates a type (either a concrete type or a type variable) with another type. These types possibly contain type variables associated with other expressions.

```
case class TEq(lty: Type, rty: Type)
```

Inferring types

Once you have generated a set of constraints, you need to unify them to get a substitution: a coherent mapping from type variables to (possibly generic) concrete types. Your job is to implement a `unify` function that takes a set of type equality constraints `cs`, a substitution `sub`, and returns a substitution.

```
type TSubstitution = List[TEq]

def unify(cs: List[TEq], sub: TSubstitution): TSubstitution = // ...
```

Type lookup

You have to implement the `lookup` function, which takes a substitution `sub` and a type variable `v`, and returns the type associated with `v` in `subs`, or `v` if not found.

```
def lookup(sub: TSubstitution, v: TVar): Type = // ...
```

Testing

As usual, you should write tests for everything, including your implementation of the type inference algorithm. The test should provide unit tests with good coverage of the language, including corner cases. A typical corner case that is being overlooked, is the fact that where the grammar states that an expression is expected, you should really test with arbitrary expressions (of the right type). For example, the function position of the application operator can be any expression:

```
(let ((inc (lambda (y) (+ y 1))))
  (inc (+ 3 4)))
```

This holds for all constructs in the language that take an `<expr>` in the grammar.

To demonstrate your understanding of the language that you implement, we ask you to write at least three tests at the top of your Test file that exercise the language beyond simple unit tests.

New Exceptions

The type inference introduces new possibilities for exceptions. You should raise specific exceptions just like in every assignment. Note that your type inferrer should raise only exceptions that inherit from `TypeInferenceException`.

Grammar

The language is similar to the language from Week 4, but simplified, as it does not have, e.g., syntactic sugar for recursive functions.

```
module inference

imports Common

context-free syntax

  Exp.NumExt        = INT      // integer literals
  Exp.TrueExt       = [true]
  Exp.FalseExt      = [false]
  Exp.IdExt         = ID
  Exp.NilExt        = [nil]

  Exp.UnOpExt       = [[UnOp] [Exp]]
  Exp.BinOpExt      = [[BinOp] [Exp] [Exp]]

  UnOp.MIN          = [-]
  UnOp.NOT          = [not]
  UnOp.HEAD         = [head]
  UnOp.TAIL         = [tail]
  UnOp.ISNIL        = [is-nil]

  BinOp.PLUS        = [+]
  BinOp.MULT        = [*]
  BinOp.MINUS       = [-]
  BinOp.AND         = [and]
  BinOp.OR          = [or]
  BinOp.NUMEQ       = [num=]
  BinOp.NUMLT       = [num<]
  BinOp.NUMGT       = [num>]
  BinOp.CONS        = [cons]

  Exp.IfExt         = [(if [Exp] [Exp] [Exp])]
  Exp.FdExt         = [(lambda ([ID*]) [Exp])]
  Exp.AppExt        = [([Exp] [Exp*])]
  Exp.LetExt        = [(let ([LetBind+]) [Exp])]
  LetBind.LetBind   = [([ID] [Exp])]
```

Note that [Exp+] denotes one or more of [Exp], and that [Exp*] denotes zero or more of [Exp].

Abstract syntax

The ExprExt case classes are already defined and imported via `import Parser._` and `import Untyped._`. You should not put these in your solution!

```
sealed abstract class ExprExt
case class BinOpExt(s: String,  l: ExprExt, r: ExprExt)     extends ExprExt // (+ e1 e2)
case class UnOpExt (s: String,  e: ExprExt)                 extends ExprExt // (- e)
case class IfExt   (c: ExprExt, t: ExprExt, e: ExprExt)     extends ExprExt // (if c t e)
case class LetExt  (binds: List[LetBindExt], body: ExprExt) extends ExprExt // (let ((x 4)(y 5)) e)
case class FdExt   (params: List[String], body: ExprExt)    extends ExprExt // (lambda (x) e)
case class AppExt  (f: ExprExt, args: List[ExprExt])        extends ExprExt // (f a b c)
case class IdExt   (c: String)                              extends ExprExt // x
case class NumExt  (num: Int)                               extends ExprExt // 4
case class TrueExt ()                                       extends ExprExt // true
case class FalseExt()                                       extends ExprExt // false
case class NilExt()                                         extends ExprExt // nil


case class Bind      (name: String, value: Value)
case class LetBindExt(name: String, value: ExprExt)
case class TBind     (name: String, ty: Type)
case class TEq       (lty: Type, rty: Type)


object ExprExt {
  val binOps       = Set("+", "*", "-", "and", "or", "num=", "num<", "num>", "cons")
  val unOps        = Set("-", "not", "head", "tail", "is-nil")
  val reservedWords = binOps ++ unOps ++
    Set("if", "lambda", "let", "true", "false", "nil")
}

object Untyped {
  type Environment = List[Bind]
  case class FdExt(params: List[String], body: ExprExt)          extends ExprExt
  case class NilExt()                                            extends ExprExt
}
```

Desugared syntax

These case classes are also provided! No import is needed. Do not copy these to your own solution.

```
sealed abstract class ExprC
case class TrueC ()                                 extends ExprC // true
case class FalseC()                                 extends ExprC // false
case class NumC  (num: Int)                         extends ExprC // 4
case class PlusC (l: ExprC, r: ExprC)               extends ExprC // (+ e1 e2)
case class MultC (l: ExprC, r: ExprC)               extends ExprC // (* e1 e2)
case class EqNumC(l: ExprC, r: ExprC)               extends ExprC // (num= e1 e2)
case class LtC   (l: ExprC, r: ExprC)               extends ExprC // (num< e1 e2)
case class IfC   (c: ExprC, t: ExprC, e: ExprC)     extends ExprC // (if c t e)
case class AppC  (f: ExprC, args: List[ExprC])      extends ExprC // (f args)
case class IdC   (c: String)                        extends ExprC // x
case class FdC   (params: List[String], body: ExprC) extends ExprC // (lambda (x) e)
case class NilC  ()                                 extends ExprC // nil
case class ConsC (e: ExprC, list: ExprC)            extends ExprC // (cons e list)
case class HeadC (list: ExprC)                      extends ExprC // (head list)
case class TailC (list: ExprC)                      extends ExprC // (tail list)
case class IsNilC(list: ExprC)                      extends ExprC // (is-nil list)
```

Values

These case classes are also provided! No import is needed. Do not copy these to your own solution.

```
sealed abstract class Value
case class NumV (v: Int)                        extends Value
case class BoolV(v: Boolean)                    extends Value
case class ClosV(f: FdC, env: Environment) extends Value
case class ConsV(head: Value, tail: Value) extends ListV
case class NilV()                           extends ListV
```

And the type alias `Environment` for a `List[Bind]`, defined in the `Interp` object:

```
type Environment = List[Bind]
```

Exceptions

Define your own *specific* exceptions that *inherit* from the given abstract exceptions. Throw only exceptions derived from `DesugarException` in the desugarer, and so on.

```
abstract class DesugarException       extends RuntimeException
abstract class InterpException        extends RuntimeException
abstract class TypeInferenceException extends RuntimeException
```