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 Biharie
- Philippos Boon Alexaki
- Luc Everse
- Boris Janssen
- Rembrandt Klazinga
- Mirco Kroon
- Chris Lemaire
- Sterre Lutz
- Yaniv Oren
- Wouter Polet
- Thijs Raijmakers
- Jim van Vliet
- Yoshi van den Akker
- Paul van der Stel
- Eric van der Toorn

1.6. Week 6: Type Checker

In this week's assignments, you should implement a type checker and an interpreter for the Paret language from week 5, extended with type annotations and *pairs*. How to implement a type checker is discussed in Chapter 15 of the book.

1 New Features

1.1 Safe Interpretation

The safe interpretation function defined in the solution template composes the type checker and interpreter to guard against interpreting non-well-typed expressions:

```
object SafeInterp {
  def interp(e: ExprExt): Value = {
    val t = TypeChecker.typeOf(e, List())
    Interp.interp(Desugar.desugar(e), List())
  }
}
```

1.2 Type Checker

The type checker has a function `typeOf` that takes an `ExprExt` as input and returns either the type of that expression, or, if the expression is not well-typed, raises an exception derived from `TypeException`.

Similarly to how an `Environment` was passed around in the `interp` implementation that mapped identifiers to values, a *type environment* `TEnvironment` that maps identifiers to *types* should be passed around.

```
object TypeChecker {
  type TEnvironment = List[TBind]

  def typeOf(e: ExprExt, nv: TEnvironment): Type =
    NotImplementedException("TODO")
}
```

For type checking the base language, note the following:

- **Functions**: The syntax of functions has been extended to support type annotations on function parameters. Instead of `(lambda (x y) (+ x y))`, we will write `(lambda ((x : Num) (y : Num)) (+ x y))`. The type of this anonymous function is `((Num Num) -> Num)`. See the grammar for the full syntax definition.
- **Recursive functions**: The syntax of recursive functions (`rec-lam`) have been extended to support type annotations on `rec-lam`. Instead of `(rec-lam f (x) (f x))` we will write `(rec-lam (f : Num -> Bool) (x) (f x))`. The type of this function is `((Num) -> Bool)`.
- **Recursive lets**: The syntax of recursive let bindings has also been extended with type annotations on identifiers. Instead of `(letrec ((x 1)) x)` we will write `(letrec (((x : Num) 1)) x)`. The type of this expression is `Num`.
- **Lists**: To disambiguate what the type of a `nil` list is, the empty list is explicitly annotated, using the syntax `(nil : T)` where `T` is a type. List expressions are also explicitly annotated. Instead of `(list 1 2)` we write `(list : Num (1 2))`. Note that, since we are now working with a type checked language, the `is-list` construct from previous weeks no longer serves a useful purpose, and so has been dropped from the language of this week.
- **Boxes**: Boxes have the type `(Ref T)`, where `T` is the type of the expression that is inside the box.
- **If expressions**: Your type checker should require that both branches of an `if` statement have the same type. So, for instance, `(if true 3 true)` should not type check.
- **Other types**: For each type of expression, you have to check the types of the arguments. It is up to find out what type each expression is supposed to return.
- **Subtyping**: You do *not* need to implement function subtyping, and should not write test cases that assume it will be implemented.

1.3 Pairs

You should extend your interpreter with support for *pairs*:

- `(pair e1 e2)` evaluates to a pair of two values, corresponding to the results of evaluating each of `e1` and `e2`.
- `(fst e)` and `(snd e)` evaluates `e` to a pair value and projects the *first* or *second* value from the pair.

Pairs should be typed using the `Pair` type. For example, `(pair 1 true)` should have type `(Pair Num Bool)`. See the syntax definition below for more details.

1.4 Typed `let`, `letrec`, and `rec-lam`

The syntax of `let` is the same as in previous weeks. `let` expressions should be type checked in the obvious way: type check each binding of the `let`; and then type check the body in a type context that has been extended with an appropriate type for each name that the `let` binds.

`letrec`s are more challenging. For example, what is an appropriate type for `x` in a `letrec` expression like this one?:

```
(letrec ((x x)) x)
```

The challenge of type checking `letrec`s is that binder expressions can refer to identifiers whose type we are currently in the process of computing (like `x` in the program above). We tackle this by requiring programmers to add type annotations to all names that a `letrec` binds. E.g.:

```
(letrec (((x : Num) x))
  x)
```

To type check a `letrec`, we should first construct the type environment under which all expressions and the body will be type checked; and then verify that each expression type checks under this environment and matches the type annotation. E.g., the identifier `x` in the binder expression of `((x : Num) x)` is type checked in an environment that associates `x` with the type `Num`.

The body of recursive functions (`rec-lam`) must be type checked under a type environment where *both* the name of the function being defined *and* the function argument is in scope. E.g., the following should type as a function of type `((Num) -> Bool)`:

```
(rec-lam (f : Num -> Bool) (n) (f n))
```

1.5 Desugaring Erases Types

`ExprExt`s include type annotations, whereas `ExprC`s do not. Type checking happens *statically* (i.e., before the program is run), so type annotations are desugared away when we transform `ExprExt` expressions into `ExprC` core language expressions that interpreters take as input.

1.6 New Exceptions

The new features introduce new possibilities for exceptions. You should raise specific exceptions just like in previous weeks' assignments, to ease yoour debugging process. The specific exceptions will not be explicitly graded, but taken in consideration in the code quality check. Note that your type checker should raise *only* exceptions that inherit from `TypeException`.

2 Grammar

The grammar of this week's language is an extension of the language of week 5 (Mutation). The specification of the constructs we saw before (such as `let` and function application) are as before, but note that the parameters of `lambda`, `letrec`, `rec-lam`, and `nil` now have explicit type annotations.

Furthermore, the language has been extended with pairs and project expressions as summarized above.

```
module types

imports Common

context-free syntax

  Expr.NumExt       = INT      // integer literals
  Expr.TrueExt      = [true]
  Expr.FalseExt     = [false]
  Expr.IdExt        = ID

  Expr.UnOpExt      = [([UnOp] [Expr])]
  Expr.BinOpExt     = [([BinOp] [Expr] [Expr])]

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

  UnOp.FST          = [fst]
  UnOp.SND          = [snd]

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

  BinOp.PAIR        = [pair]

  Expr.IfExt        = [(if [Expr] [Expr] [Expr])]

  Expr.NilExt       = [(nil : [Type])]
  Expr.ListExt      = [(list : [Type] ([Expr*]))]

  Param.Param       = [([ID] : [Type])]

  Expr.FdExt        = [(lambda ([Param*]) [Expr])]
  Expr.AppExt       = [([Expr] [Expr*])]
  Expr.LetExt       = [(let ([LetBind+]) [Expr])]
  Expr.LetRecExt    = [(letrec ([LetRecBind+]) [Expr])]
  Expr.Set          = [(set [ID] [Expr])]

  LetBind.LetBindExt = [([ID] [Expr])]
  LetRecBind.LetRecBind = [([Param] [Expr])]

  Expr.RecLamExt    = [(rec-lam ([ID] : [Type] -> [Type]) ([ID]) [Expr])]

  Type.NumT         = [Num]
  Type.BoolT        = [Bool]
  Type.ListT        = [(List [Type])]
  Type.FunT         = [(([Type*]) -> [Type])]
  Type.PairT        = [(Pair [Type] [Type])]
  Type.RefT         = [(Ref [Type])]
```

Note that you must put spaces around the colon `:` and arrow `->` for them to parse correctly.

Also note that [Expr+] denotes one or more of [Expr], and that [Expr*] denotes zero or more of [Expr].

3 Classes

3.1 Abstract Syntax

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

```
case class TrueExt() extends ExprExt
case class FalseExt() extends ExprExt
case class NumExt(num: Int) extends ExprExt
case class BinOpExt(s: String, l: ExprExt, r: ExprExt) extends ExprExt
case class UnOpExt(s: String, e: ExprExt) extends ExprExt
case class IfExt(c: ExprExt, t: ExprExt, e: ExprExt) extends ExprExt
case class NilExt(listTy: Type) extends ExprExt
case class ListExt(listTy: Type, es: List[ExprExt]) extends ExprExt
case class AppExt(f: ExprExt, args: List[ExprExt]) extends ExprExt
case class IdExt(c: String) extends ExprExt
case class FdExt(params: List[Param], body: ExprExt) extends ExprExt
case class LetExt(binds: List[LetBindExt], body: ExprExt) extends ExprExt
case class SetExt(id: String, e: ExprExt) extends ExprExt
case class RecLamExt(name: String,
                     paramTy: Type,
                     retTy: Type,
                     param: String,
                     body: ExprExt) extends ExprExt
case class LetRecExt(binds: List[LetRecBindExt],
                     body: ExprExt) extends ExprExt

case class LetBindExt(name: String, value: ExprExt)
case class LetRecBindExt(name: String, ty: Type, value: ExprExt)

object ExprExt {
  val binOps = Set("+", "*", "-", "and", "or", "num=", "num<", "num>",
    "cons", "setbox", "seq", "pair")
  val unOps = Set("-", "not", "head", "tail", "is-nil", "box", "unbox", "fst", "snd")
  val reservedWords = binOps ++ unOps ++ Set("list", "if", "lambda",
    "let", "true", "false", "rec-lam", "set", "letrec",
    ":", "->", "Num", "Bool", "List", "Pair", "Ref")
}
```

3.2 Desugared Syntax

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

```
abstract class ExprC
case class TrueC() extends ExprC
case class FalseC() extends ExprC
case class NumC(num: Int) extends ExprC
case class PlusC(l: ExprC, r: ExprC) extends ExprC
case class MultC(l: ExprC, r: ExprC) extends ExprC
case class IfC (c: ExprC, t: ExprC, e: ExprC) extends ExprC
case class EqNumC(l: ExprC, r: ExprC) extends ExprC
case class LtC(l: ExprC, r: ExprC) extends ExprC
case class NilC() extends ExprC
case class ConsC(l: ExprC, r: ExprC) extends ExprC
case class HeadC(e: ExprC) extends ExprC
case class TailC(e: ExprC) extends ExprC
case class IsNilC(e: ExprC) extends ExprC
case class AppC(f: ExprC, args: List[ExprC]) extends ExprC
case class IdC(c: String) extends ExprC
case class FdC(params: List[String], body: ExprC) extends ExprC
case class BoxC(v: ExprC) extends ExprC
case class UnboxC(b: ExprC) extends ExprC
case class SetboxC(b: ExprC, v: ExprC) extends ExprC
case class SetC(v: String, b: ExprC) extends ExprC
case class SeqC(b1: ExprC, b2: ExprC) extends ExprC
case class UninitializedC() extends ExprC
case class PairC(l: ExprC, r: ExprC) extends ExprC
case class FstC(e: ExprC) extends ExprC
case class SndC(e: ExprC) extends ExprC
```

3.3 Values

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

```
abstract class Value
case class NumV(v: Int) extends Value
case class BoolV(v: Boolean) extends Value
case class NilV() extends Value
case class ConsV(hd: Value, tl: Value) extends Value
case class PointerClosV(f: FdC, env: List[Pointer]) extends Value
case class BoxV(l: Int) extends Value
case class UninitializedV() extends Value
case class PairV(l: Value, r: Value) extends Value
```

3.4 Types

These case classes are also provided! You'll need `import Typed._` in your code for everything to work correctly. Do not copy these to your own solution.

```
sealed abstract class Type
case class NumT() extends Type
case class BoolT() extends Type
case class FunT(paramTy: List[Type],
                retTy: Type) extends Type
case class ListT(expTy: Type) extends Type
case class PairT(fst: Type, snd: Type) extends Type
case class RefT(t: Type) extends Type
```

3.5 Other

```
case class Pointer(name: String, location: Int)
case class Cell(location: Int, value: Value)
case class TBind(name: String, ty: Type)
case class Param(name: String, ty: Type)
```

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 TypeException    extends RuntimeException
```