Lucrare de atestat la informatica

Matei Adriel Profesor Tiberiu Koos

September 6, 2022

Contents

1 Introducere

2 Motivatii

Sistemele de tipuri sunt una dintre cele mai interesante si complexe parti ale compilatoarelor moderne. Deducerea tipurilor incepe incet-incet sa nu mai fie doar un subiect discutat in cercetare, ci o necesitate pentru orice limaj de programare ce spera sa fie folosit in industrie. Sistemele de tipuri pot fi implementate in multe moduri diferite, fiecare cu avatajele si dezavatajele lui. Eu consider ca pentru a putea zice ca "intelegi" un concept din programare cu adevarat, este necesar sa poti sa il implementezi tu insusi. La inceputul lui 2020, mi-am inceput asadar aventurile in lumea programarii functionale si a sistemelor de tipuri.

```
auto a = 2 + 3;
```

Figure 1: Deducerea tipului unei variabile in C++

3 Scurta descriere

Aplicatia lunarbox prezinta un mediu vizual pentru programarea functionala. Toate tipurile sunt verificate, asa ca utilizatorul primeste feedback in timp real cu privire la validitatea programului sau. In plus, numele de tiprui (ex: Int, String...) pot aparea putin ciudate pentru incepatori, asa ca fiecare dintre ele este reprezentat vizual prin o culare diferita. Pentru a ajuta persoanele ce vor sa invete programarea functionala, lunarbox ofera 3 moduri de operare.

3.1 Proiecte

Sectiunea Projects opereaza ca un sandbox, in care utilizatorul poate sa experimenteze cu orice nod oferit de lunarbox.

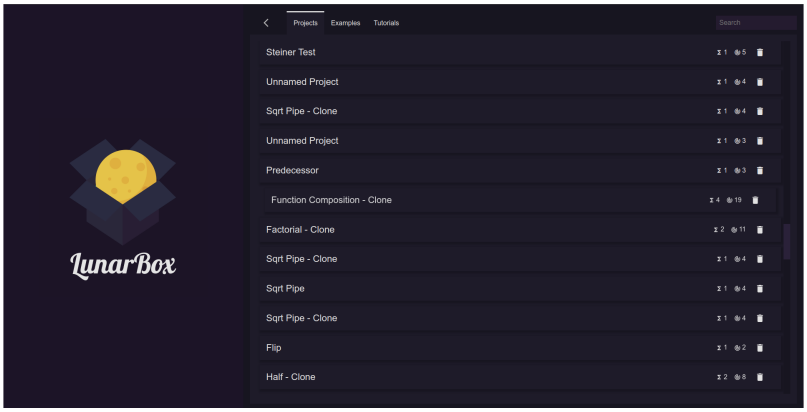

Figure 2: Exemplu de pagina cu proiecte al unui utilizator

3.2 Exemple

Sectiunea Exemple contine o serie de programe create si mentinute de adminii lunarbox. Orice utilizator poate sa creeze o copie locala a oricarui exemplu cu un simplu click, lucru ce incurajeaza experimentarea.

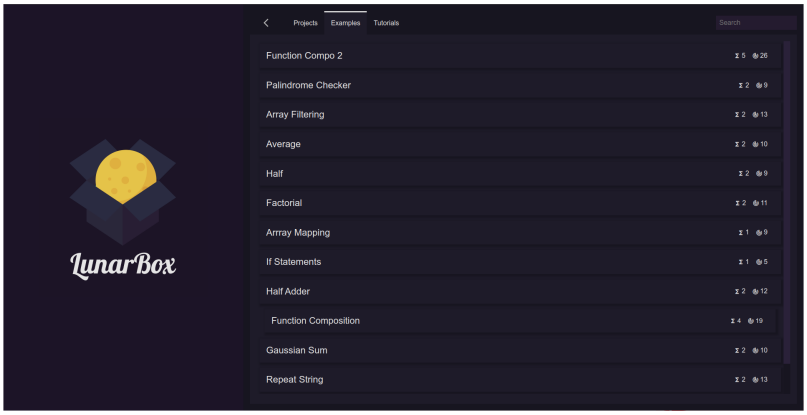

Figure 3: Meniul cu exemple

3.3 Tutoriale

Fiecare tutorial incepe cu o serie de "slide"-uri, care explica conceptul despre care este vorba. Multe dintre slideuri contin animatii, iar utilizatorul poate chiar apasa pe unele dintre imagini pentru a deschide exemplul respectiv intr-un sandbox. Utilizatorul este apoi confruntat cu un exercitiu. Solutia propusa de utilizator este verificata contra unei serii de teste (similar cu siteuri precum pbinfo), iar daca raspunsul este corect, tutorialul este considerat complet. Utilizatorul are apoi posibilitatea sa continue sa se joace cu tutorialul curent, sau sa treaca la urmatorul. Tutorialele sunt create si intretiunte de adminii lunarbox.

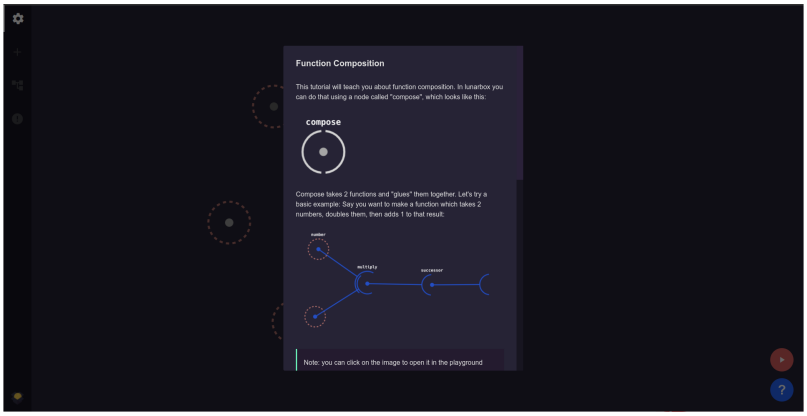

Figure 4: Tutorial-ul despre compunerea functiilor

4 Cerinte hardware si software

Lunarbox ruleaza pe orice calculator cu un broswer modern instalat si o conexiune la internet

5 Descrierea aplicatiei

Proiectul este scris intr-o combinatie dintre limbajele PureScript si TypeScript. Doar contributia mea la proiect depaseste bariera de 10 mii de linii de cod, asa ca nu ar fi practic sa prezint toata sursa aici. Sursa completa poate fi citita pe pagina de github.

5.1 Deducerea tipurilor

Deducerea tipurilor este un proces relativ complex. Am ales sa folosesc un sistem de tipuri Hindley–Milner. Functia principala pentru deducerea tipurilor, implementata in PureScript, este atasata in figura 5

5.2 Creerea nodurilor

Una dintre cele mai importante probleme aparute in dezvoltarea acestui proiect a fost creerea unui mod generic de a "impacheta" functii deja existene in noduri ce pot fi folosite in editorul lunarbox. Acest lucru este mai complex decat pare - spre exemplu, sistemul de tipuri al editorului trebuie sa stie cum sa reactioneze la astfel de noduri. Nu as avea loc pentru a arata toate fisierele ce au de aface cu aceasta problema aici. In figura 6 se poate observa implementarea unei clase care primeste ca input un tip PureScript si returneaza un tip lunarbox.

5.3 Interpretarea programelor

La prima vedere, interpretarea programelor este o problema triviala (comparativ cu alte module cum ar fi sistemul de tipuri). Singurul loc unde acest lucru devine complicat este rularea functiilor recursive. Utilizatorul este liber sa defineasca astfel de functii, iar interpretatorul trebuie sa stie cum sa le execute. In figura 7 am atasat codul pentru evaluarea unei expresii lunarbox

5.4 De la grafuri la arbori ai sintaxei

Majoritatea algoritmilor pentru interpretarea limbajelor de programare, sisteme de tipuri, etc, lucreaza pe arbori de sintaxa abstracta (AST). Programele create de utilizatori in editor iau forma unui graf orientat, asa ca am creat un algoritm care sa converteasca astfel de structuri inr-un limbaj intern standard, pe care urmeaza sa rulez mai apoi restul modulelor. Figura 8 exemplifica structura de graf a programelor lunarbox, iar figura 9 arata AST-ul in care aceste structuri sunt transformate.

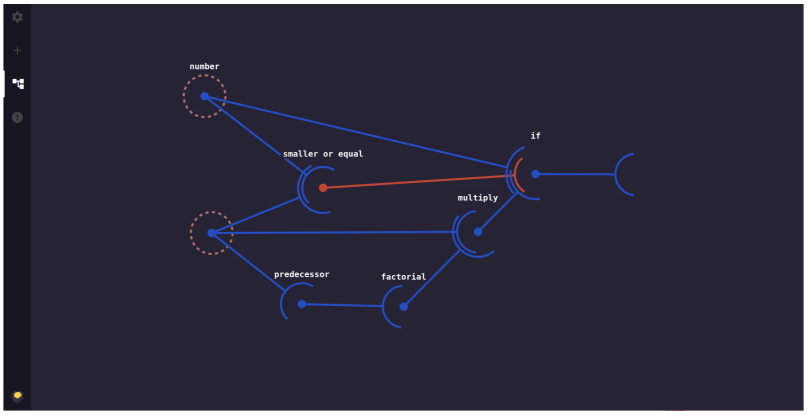

Figure 8: Program lunarbox ce ilustreaza structura de graf

```
newtype VarName
  = VarName String

data NativeExpression
  = NativeExpression Scheme RuntimeValue

data Expression l
  = Variable l VarName
  | FunctionCall l (Expression l) (Expression l)
  | Lambda l VarName (Expression l)
  | Let l VarName (Expression l) (Expression l)
  | If l (Expression l) (Expression l) (Expression l)
  | FixPoint l VarName (Expression l)
  | Expression l (Expression l)
  | Native l NativeExpression
  | TypedHole l
```

Figure 9: Tipul de date pentru AST-ul lunarbox

6 Concluzii

6.1 Monazii - o solutie eleganta pentru lucrul cu efecte

Cand zic efecte, nu ma refer la sensul colocvial al cuvantului, ci la cel din domeniul programarii functionale. Un efect este o actiune ce nu e pura (ex: afisarea unei propozitii pe ecran). Codul din figura 5 foloseste multe astfel de efecte (generarea de id-uri unice, colectarea de "constrangeri", memorarea rezultatelor intermediare, indexarea unui "context", etc), toate aceste efecte fiind atat verificate de sistemul de tiprui al limbajului PureScript cat si suficient de "ne-intrusive" incat codul arata ca si cum ar fi scris intr-un DSL (domain-specific-language) special facut pentru aceasta problema. Consider ca monazii sunt o solutie excelenta pentru astfel de probleme.

6.2 Programarea functionala ca metoda de implementare a sistemelor complexe

Imi doresc ca acest proiect sa fie o dovada vie ca limbajele de programare precum PureScript (care la randul sau este foarte similar cu Haskell) sunt o solutie excelenta pentru creerea si intretinerea de sisteme ce prezinta complexitate ridicata.

Proiectul meu contine multe alte module pe care nu le-asi fi putut prezenta fara a ocupa foarte mult spatiu (un sistem de conturi, optiuni pentru admini, un sistem de "warninguri" si un optimizator pentru AST-ul lunarbox, etc). Paradigma programarii functionale este o solutie excelenta pentru astfel

de proiecte, deoarece folosita corect poate duce la programe usor de extins si intretinut.

7 Limits

7.1 Definition

$$\lim_{x \to \alpha} f(x) = L \iff$$
$$\forall \epsilon, \epsilon > 0 \rightarrow$$
$$\exists \sigma, \sigma > 0,$$
$$\forall x, 0 < |x - a| < \sigma \rightarrow |f(x) - L| < \epsilon$$

7.2 Using neighbourhoods

$$\lim_{x \to \alpha} f(x) = L \iff$$
$$\forall \epsilon, \epsilon > 0 \rightarrow$$
$$\exists \sigma, \sigma > 0,$$
$$\forall x, x \in I_0(a, \sigma) \rightarrow f(x) \in I(L, \epsilon)$$

```haskell
-- Infers a type and marks it's location on the typeMap
infer :: forall l. Ord l => Show l => Expression l -> Infer l Type
infer expression =
  withLocation (getLocation expression) do
    type' <- case expression of
      Variable _ name -> do
        lookupEnv name
      Lambda _ param body -> do
        tv <- fresh true
        t <- createClosure param (Forall [] tv) $ infer body
        pure $ typeFunction tv t
      FunctionCall _ func input -> do
        funcType <- infer func
        inputType <- infer input
        tv <- fresh true
        createConstraint funcType (typeFunction inputType tv)
        pure tv
      If _ cond then' else' -> do
        tyCond <- infer cond
        tyThen <- infer then'
        tyElse <- infer else'
        tv <- fresh true
        createConstraint tyCond typeBool
        createConstraint tv tyThen
        createConstraint tv tyElse
        pure tv
      Let location name value body -> do
        env <- ask
        Tuple valueType (InferOutput { constraints }) <- listen $ infer value
        let
          (Tuple subst (SolveState { errors })) = runSolve (SolveContext { location }) $ sol
        for_ errors $ createError <<< Stacked
        generalized <- local (const $ apply subst env) $ generalize $ apply subst valueType
        createClosure name generalized $ infer body
      FixPoint loc name body -> do
        tv <- fresh true
        ty <- createClosure name (Forall [] tv) $ infer body
        createConstraint tv ty
        pure ty
      Expression _ inner -> infer inner
      TypedHole _ -> fresh false
      Native _ (NativeExpression scheme _) -> instantiate scheme
    rememberType type'
```

Figure 5: Deducerea tipurilor implementata in PureScript

```
class Typeable (a :: Type) where
  typeof :: Proxy a -> Type

instance typeableNumber :: Typeable Number where
  typeof _ = typeNumber

instance typeableInt :: Typeable Int where
  typeof _ = typeNumber

instance typeableString :: Typeable String where
  typeof _ = typeString

instance typeableBool :: Typeable Boolean where
  typeof _ = typeBool

instance typeableArray :: Typeable a => Typeable (Array a) where
  typeof _ = typeArray (typeof (Proxy :: Proxy a))

instance typePair :: (Typeable a, Typeable b) => Typeable (Tuple a b) where
  typeof _ = typePair (typeof _a) (typeof _b)
    where
    _a :: Proxy a
    _a = Proxy

    _b :: Proxy b
    _b = Proxy

instance typeableSymbol :: IsSymbol sym => Typeable (SProxy sym) where
  typeof _ = TVariable true $ TVarName $ reflectSymbol (SProxy :: SProxy sym)

instance typeableArrow :: (Typeable a, Typeable b) => Typeable (a -> b) where
  typeof _ = typeFunction (typeof (Proxy :: Proxy a)) (typeof (Proxy :: Proxy b))

-- | Get the type of a purescript value
getType :: forall a. Typeable a => a -> Type
getType _ = typeof (Proxy :: Proxy a)
```

Figure 6: "Impachetarea" tipurilor native din PureScript ca tipuri pentru noduri din editor

```haskell
interpret :: forall l. Ord l => Default l => Expression l -> Interpreter l (Term l)
interpret expression = do
  overwrites <- asks $ view _overwrites
  -- ... mai multe definitii irelevante scoase pentru a aveam spatiu
  value <- case maybeOverwrite of
    Just overwrite -> pure overwrite
    Nothing -> case expression of
      TypedHole _ -> pure $ Term Null
      Variable _ name -> getVariable $ show name
      Lambda _ _ _ -> do
        env <- getEnv
        pure $ Closure env expression
      Expression _ inner -> interpret inner
      If _ cond then' else' -> interpret cond >>= go
        -- ... mai multe definitii irelevante scoase pentru a aveam spatiu
      Let _ name value body -> do
        runtimeValue <- interpret value
        withTerm (show name) runtimeValue $ interpret body
      expr@(FixPoint l name body) -> do
        env <- getEnv
        let self = Closure env expr
        withTerm (show name) self $ interpret body
      Native _ (NativeExpression _ inner) -> pure $ Term inner
      FunctionCall _ function argument -> do
        runtimeArgument <- interpret argument
        runtimeFunction <- interpret function
        let
          go = case _ of
            Closure env (Lambda _ name expr) ->
              scoped $ withEnv env $ withTerm (show name) runtimeArgument
                $ interpret expr
            Closure env expr -> call >>= go
              where
              call = scoped $ withEnv env $ interpret expr
            Term (Function call) -> do
              ctx <- ask
              pure $ Term $ call $ termToRuntime ctx runtimeArgument
            Term _ -> pure def
        go runtimeFunction
  toplevel <- asks $ view _toplevel
  when toplevel $ tell $ ValueMap $ Map.singleton location value
  pure value
```

Figure 7: Evaluarea expresiilor in lunarbox