O pracy magisterskiej

Analiza statyczna programów w języku Python

Plan prezentacji

- Cel pracy
- Zrobione
  - Teoria V. Bono, M. Benke, A. Shubert
  - Implementacje M. Benke, M. Oniszczuk
- Do zrobienia

Motywacja

```
x = EmptyClass()
x.a = 42

print x.a # OK
print x.b # Error
```

Motywacja

```
x = EmptyClass()
if cond:
    x.a = 42
else:
    x.a = "hello"
print x.a + 7 # Error
```

Cel pracy

- Type-checker do Pythona
- System typów
- Annotation inference (Inferencja anotacji)

Teoria

Teoria – język Lucretia

```
x = EmptyClass()
x.a = 42
print x.b # Error
```

```
let x = new in
  let _ = (x.a = 42) in
    x.b -- Error
```

Teoria – reguły

$$\frac{X \text{ fresh}}{\Psi; \Gamma \vdash \mathbf{new} : X; X <\# \{\}, \Psi}$$

$$\frac{\vdash u \quad \Psi \ni X <\# \{a : u, \overline{l : t}\}}{\Gamma, x : X; \vdash x.a : u; \Psi}$$

Teoria – reguły

$$\frac{\Gamma; \Psi \vdash e_1 : \mathit{bool}; \Psi_1 \quad \Gamma; \Psi_1 \vdash e_2 : t_2; \Psi_2 \quad \Gamma; \Psi_1 \vdash e_3 : t_3; \Psi_2'}{\Gamma; \Psi \vdash \mathsf{if}\ (e_1)\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : u; \Psi_2 \uplus \Psi_2'}$$

Implementacja do języka Lucretia

Type-checker

$$\frac{X \text{ fresh}}{\Psi; \Gamma \vdash \mathbf{new} : X; X <\# \{\}, \Psi}$$

```haskell
data Env = M.Map Var Type

findType :: Env -> Exp -> M Type
findType env ENew = do
  t <- freshTVar
  emptyConstraint t
  return $ TVar t
```

Co działa

- Reguły dla instrukcji:
  - na rekordach (x.a = _, itp.)
  - if _ then _ else _
  - funkcje z anotacjami

Czego nie ma

- funkcji bez anotacji (Annotation inference)

Implementacja do Goto-języka

Goto-język

```
x = EmptyClass()      INew "x"
x.a = 42              ISetField "x" "a" (EInt 42)
print x.b # Error     IPrint (EGetField "x" "b")
```

Bloki proste

```
x = EmptyClass()
if cond:
  x.a = 42
else:
  x.a = "hello"
print x.a + 7 # Error
```

```
L0:
  x = new;
  if cond goto L1 else L2;
L1:
  x.a = 42;
  goto L3;
L2:
  x.a = "hello";
  goto L3;
L3:
  print x.a + 7;
```

Bloki proste

```
x = EmptyClass()
if cond:
  x.a = 42
else:
  x.a = "hello"
print x.a + 7 # Error
```

```
L0:
  x = new;
  if cond goto L1 else L2;
L1:
  x.a = 42;
  goto L3;
L2:
  x.a = "hello";
  goto L3;
L3:
  print x.a + 7;
```

```
ILabel "L0"
INewRec "x"
IGotoIf (Var "cond") "L1" "L2"
```

Type-checker

$$\frac{\Gamma; \Psi \vdash e_1 : bool; \Psi_1 \quad \Gamma; \Psi_1 \vdash e_2 : t_2; \Psi_2 \quad \Gamma; \Psi_1 \vdash e_3 : t_3; \Psi_2'}{\Gamma; \Psi \vdash \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 : u; \Psi_2 \uplus \Psi_2'}$$

```haskell
data Env = M.Map Var Type
data Fact = (Env, Constraints)


factTransfer :: Insn -> Env -> [(Label, Env)]
factTransfer (IGotoIf _ tLbl fLbl) env = [(tLbl, env), (fLbl, env)]

join :: Fact -> Fact -- łączenie faktów
join (e1, c1) (e2, c2) = (mergeEnvs e1 e2, mergeConstraints c1 c2)
```

Co się kompiluje

- Reguły dla instrukcji
  - na rekordach (x.a = _, itp.)
  - if _ then _ else _
  - funkcje **bez anotacji (czyli z annotation inference)**

Czego nie ma

- funkcji **z anotacjami**

Izomorfizm między implementacjami

$$\frac{X \text{ fresh}}{\Psi; \Gamma \vdash \mathbf{new} : X; X <\# \{\}, \Psi}$$

```haskell
data Env = M.Map Var Type
data Fact = (Env, Constraints)

-- Lucretia: ------------------------

findType :: Env -> Exp -> M Type
findType env ENew = do
  t <- freshTVar
  emptyConstraint t
  return $ TVar t

-- Goto: ----------------------------

factTransfer :: Insn -> Fact -> Fact
factTransfer (INew x) (env, cs) =
  ( M.insert x (TVar t) env
  , M.insert t emptyConstraint
  )
  where t <- freshTVar
```

$$\frac{\Gamma;\Psi \vdash e_1 : bool;\Psi_1 \quad \Gamma;\Psi_1 \vdash e_2 : t_2;\Psi_2 \quad \Gamma;\Psi_1 \vdash e_3 : t_3;\Psi_2'}{\Gamma;\Psi \vdash \mathsf{if}\ (e_1)\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : u;\Psi_2 \uplus \Psi_2'}$$

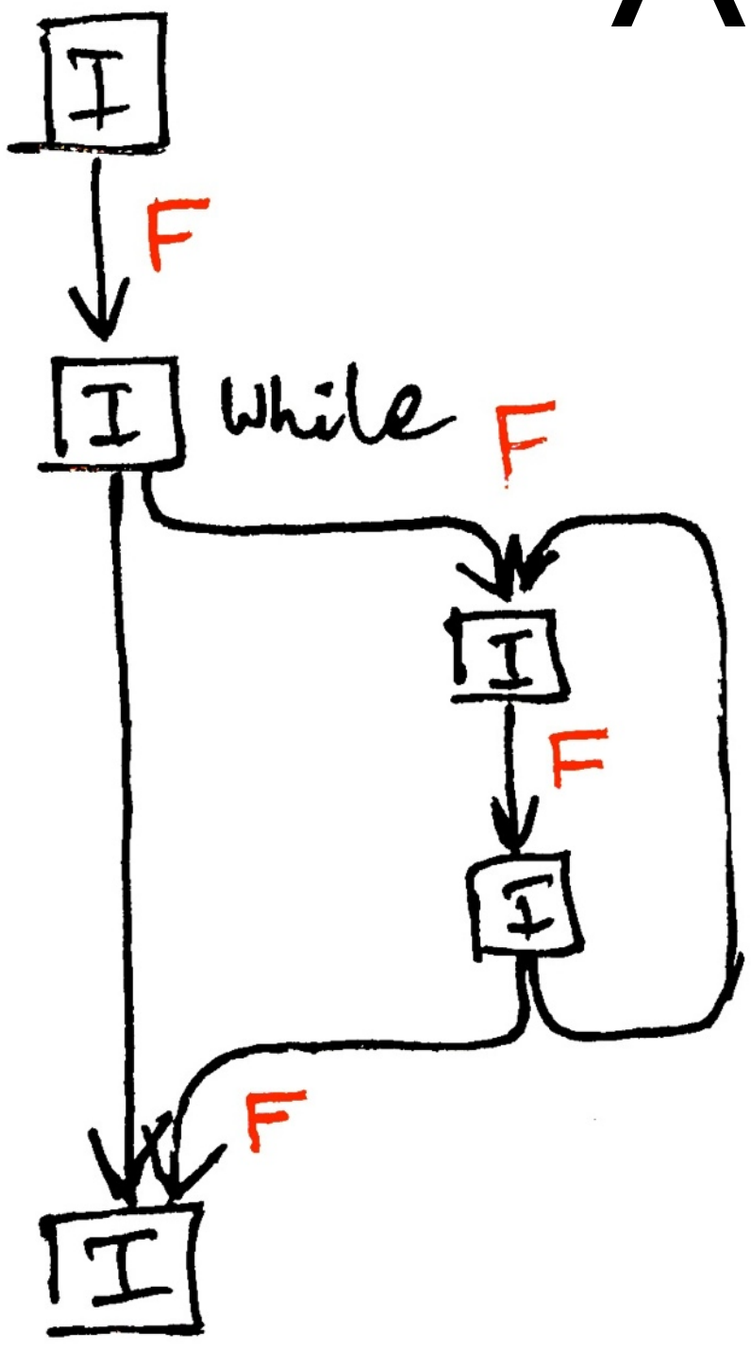
```haskell
data Env = M.Map Var Type
data Fact = (Env, Constraints)

-- Lucretia: ------------------------------------------------

findType :: Env -> Exp -> M Type
findType env (EIf eCond eThen eElse) = do
  TBool <- findType env eCond
  constraintsBeforeBody <- get

  put constraintsBeforeBody
  tThen <- findTypeClean env eThen
  constraintsAfterThen <- get

  put constraintsBeforeBody
  tElse <- findTypeClean env eElse
  constraintsAfterElse <- get

  put $ mergeConstraints constraintsAfterThen constraintsAfterElse
  return $ mergeTypes tThen tElse

-- Goto: ----------------------------------------------------

factTransfer :: Insn -> Env -> [(Label, Env)]
factTransfer (IGotoIf _ tLbl fLbl) env = [(tLbl, env), (fLbl, env)]

join :: Fact -> Fact -- łączenie faktów
join (e1, c1) (e2, c2) = (mergeEnvs e1 e2, mergeConstraints c1 c2)
```

Do zrobienia

Do zrobienia

- System typów
- .py => język pośredni
  - Sposób zapisu anotacji w .py
- Annotation inference

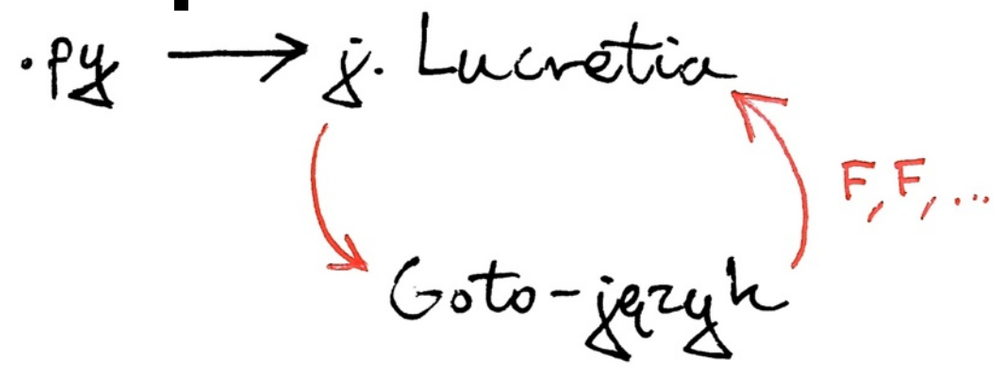
Annotation inference: Abstrakcja



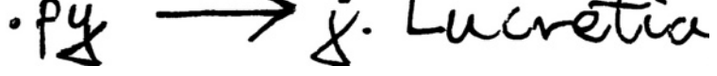
- Typeflow analysis
- Instrukcje I
- Fakty (punkt stały) F
- Przepływ sterowania ↓

Annotation inference: Sposoby implementacji

1. **System typów wymusza język + Hoopl**
   - punkt stały – Hoopl
   - *impl. Goto–j.* + *impl. j. Lucretia*



2. **System typów wymusza język + własny fixpoint**
   - punkt stały – własny algorytm
   - *impl. j. Lucretia*



3. **Hoopl wymusza język**
   - punkt stały – Hoopl
   - *impl. Goto–j.*

Annotation inference: Sposoby implementacji

- type-checking ~ type-inference
- lepiej 1 niż 2 języki pośrednie
- lepiej skorzystać z gotowego framework'a do punktu stałego
- lepiej zachować aktualny język z papieru

Plan A

Iteracyjnie:

1. Dokończyć impl. dla j. Lucretia
   - Proste programy .py -> j. Lucretia
   - Bez Annotation inference
2. Dokończyć impl. dla Goto–j.
   - Z Annotation inference

Plan B

Pominąć A.1:

1. Dokończyć impl. dla Goto–j.
   - Z Annotation inference

Lepiej A, bo

- A.1 prawdopobodnie szybciej skończona niż B
- A.1 — język z papieru, zrozumiałe przełożenie systemu typów
- w A.2 można wykorzystać wiedzę z A.1

Dziękuję