A Calculus for Constraint-Based Flow Typing

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What is Flow Typing?

- Defining characteristic: *ability to retype variables*

- JVM Bytecode provides widely-used example:

```
public static float convert(int):

   iload 0      // load register 0 on stack
   i2f          // convert int to float
   fstore 0     // store float to register 0
   fload 0      // load register 0 on stack
   freturn      // return value on stack
```

Type of **r0** here is **int**

Type of **r0** here is **int**

Type of **r0** here is **float**

Type of **r0** here is **float**

- Groovy 2.0 includes flow-typing static checker

Another Example

- Non-null type checking provides another example:

```
int compare(String s1, @NonNull String s2) {
  if(s1 != null) {
    return s1.compareTo(s2);
  } else {
    return -1;
  }
}
```

Type of **s1** here is **@NonNull**

Type of **s1** here is **null**

- Many works in literature on this topic!

The Whiley Programming Language

- Statically typed using a flow-type algorithm
- Look-and-feel of dynamically-typed language:

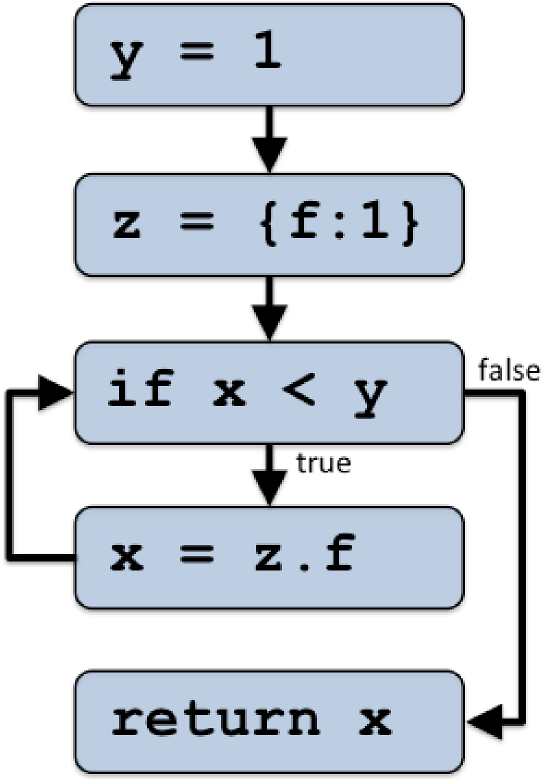
```
int|{int f} fun(bool flag):
    if flag:
        x = 1
    else:
        x = {f : 1}
    return x
```

- Question: *how to implement flow-type checker?*

A Simple Flow Typing Calculus

Example:

```
int f(int x) {
   y = 1¹
   z = {f : 1}²
   while x < y³ { x = z.f⁴ }
   return x⁵
}
```



Syntax:

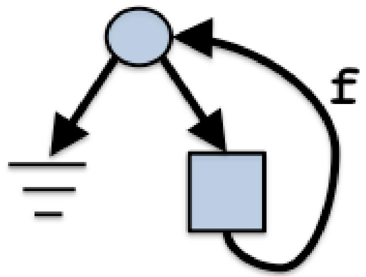
$$
\begin{array}{lcl}
\mathtt{F} & ::= & \mathtt{T\ f(T_1\ n_1, \ldots, T_n\ n_n)\ \{\ B\ \}} \\
\mathtt{B} & ::= & \mathtt{S\ B} \mid \epsilon \\
\mathtt{S} & ::= & [\![\mathtt{n = v}]\!]^{\ell} \mid [\![\mathtt{n = m}]\!]^{\ell} \mid [\![\mathtt{n.f = m}]\!]^{\ell} \mid [\![\mathtt{n = m.f}]\!]^{\ell} \mid [\![\mathtt{return\ n}]\!]^{\ell} \\
 & & \mid \mathtt{while}\ [\![\mathtt{n < m}]\!]^{\ell}\ \{\ \mathtt{B}\ \} \\
\mathtt{v} & ::= & \{\mathtt{f_1 : v_1, \ldots, f_n : v_n}\} \mid \mathtt{i}
\end{array}
$$

Language of Types

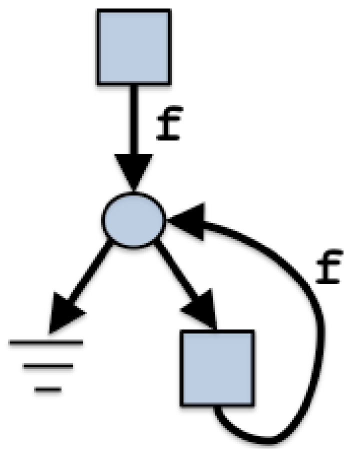
- Definition of types being considered:

$$\mathtt{T} \;::=\; \mathtt{void} \mid \mathtt{any} \mid \mathtt{int} \mid \{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \mid \mathtt{T}_1 \vee \mathtt{T}_2 \mid \mu\mathtt{X}.\mathtt{T} \mid \mathtt{X}$$

- Understanding **recursive types**:



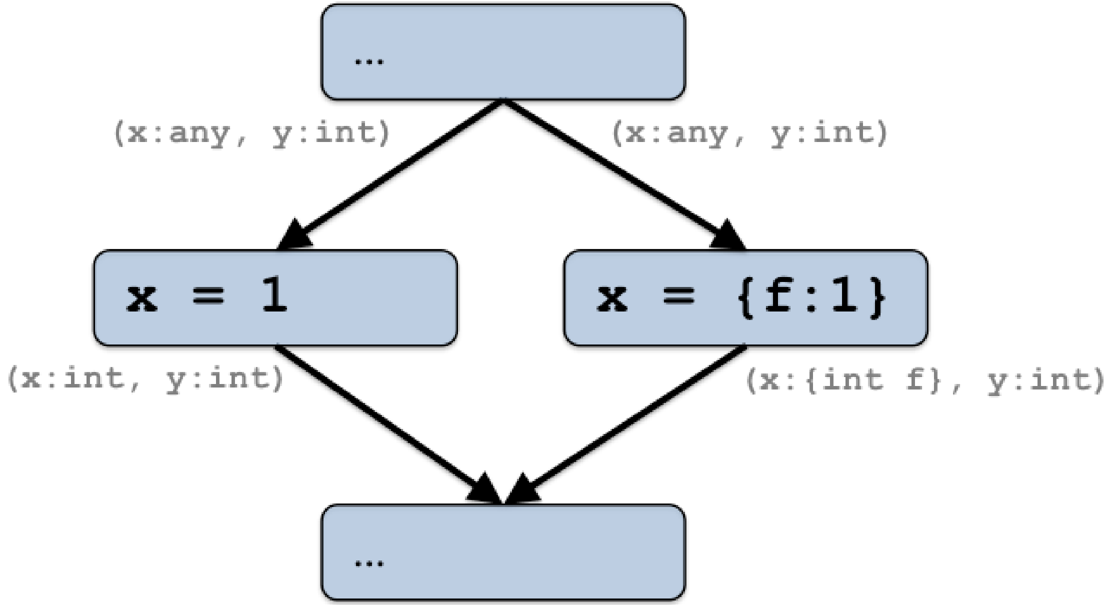
$\mu\mathtt{X}.\mathtt{int} \vee \{\ \mathtt{X}\ \mathtt{f}\ \}$



$\{\ \mu\mathtt{X}.\mathtt{int} \vee \{\ \mathtt{X}\ \mathtt{f}\ \}\ \mathtt{f}\ \}$

- **Note:** language above defines subset of types found in Whiley

Dataflow-Based Flow Typing



- **Dataflow Analysis** is commonly used for flow typing (e.g. JVM Bytecode Verifier)

- Dataflow algorithm maintains **environment** at each point mapping variables to types. These can be **joined** (e.g. $\Gamma_1 \sqcup \Gamma_2$).

Dataflow-Based Typing Rules

- Dataflow rules determine how environment is affected by statements:

$$\frac{\vdash \mathtt{v} : \mathtt{T}}{\Gamma \vdash [\![\mathtt{n}=\mathtt{v}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \mathtt{T}]}$$

$$\frac{\Gamma(\mathtt{m}) = \mathtt{T}}{\Gamma \vdash [\![\mathtt{n}=\mathtt{m}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \mathtt{T}]}$$

$$\frac{\Gamma(\mathtt{m}) = \{\ldots, \mathtt{T}\ \mathtt{f}, \ldots\}}{\Gamma \vdash [\![\mathtt{n}=\mathtt{m.f}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \mathtt{T}]}$$

$$\frac{\Gamma(\mathtt{n}) = \{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \qquad \mathtt{T} = \Gamma(\mathtt{n})[\mathtt{f} \mapsto \Gamma(\mathtt{m})]}{\Gamma \vdash [\![\mathtt{n.f}=\mathtt{m}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \mathtt{T}]}$$

$$\frac{\Gamma(\mathtt{n}) \leq \Gamma(\$)}{\Gamma \vdash [\![\mathtt{return}\ \mathtt{n}]\!]^{\ell} : \emptyset}$$

$$\frac{\Gamma_0 \sqcup \Gamma_1 \vdash \mathtt{B} : \Gamma_1 \qquad \Gamma_0 \sqcup \Gamma_1(\mathtt{n}) = \mathtt{int} \quad \Gamma_0 \sqcup \Gamma_1(\mathtt{m}) = \mathtt{int}}{\Gamma_0 \vdash \mathtt{while}\ [\![\mathtt{n} < \mathtt{m}]\!]^{\ell}\ \{\mathtt{B}\} : \Gamma_0 \sqcup \Gamma_1}$$

- Rule for **`while`** loops must iterate until **fixed point** reached

Fixed-Point Iteration

- Consider this function:

```
int∨{int g} fun(int n, int m, int x) {
    while n < m¹ {
      x = {g : 1}²
      n = m³
    }
    return x⁴
}
```

- Dataflow checker iterates this loop to produce type for `x`:

$$\Gamma^1 = \{\mathtt{n} \mapsto \mathtt{int},\ \mathtt{m} \mapsto \mathtt{int},\ \mathtt{x} \mapsto \mathtt{int}\}$$
$$\Gamma^1 = \{\mathtt{n} \mapsto \mathtt{int},\ \mathtt{m} \mapsto \mathtt{int},\ \mathtt{x} \mapsto \mathtt{int} \vee \{\mathtt{int\ g}\}\}$$
$$\Gamma^1 = \{\mathtt{n} \mapsto \mathtt{int},\ \mathtt{m} \mapsto \mathtt{int},\ \mathtt{x} \mapsto \mathtt{int} \vee \{\mathtt{int\ g}\}\}$$

- *So ... how do we know it always terminates?*

Termination

Question: *So ... how do we know it always terminates?*

Answer: it doesn't!

(thanks anonymous PLDI reviewer)

Termination Problem

- Unfortunately, lattice of types has **infinite height**

```
void loopy(int n, int m) {
  x = {f:1}¹
  while n < m² {
    x.f = x³
} }
```

- This example causes dataflow-based checker to loop forever!

$$\Gamma^3 = \{\mathtt{n} \mapsto \mathtt{int},\ \mathtt{m} \mapsto \mathtt{int}, \mathtt{x} \mapsto \{\mathtt{int\ f}\}\}$$
$$\Gamma^3 = \{\mathtt{n} \mapsto \mathtt{int},\ \mathtt{m} \mapsto \mathtt{int}, \mathtt{x} \mapsto \{\mathtt{int} \vee \{\mathtt{int\ f}\}\ \mathtt{f}\}\}$$
$$\Gamma^3 = \{\mathtt{n} \mapsto \mathtt{int},\ \mathtt{m} \mapsto \mathtt{int}, \mathtt{x} \mapsto \{\mathtt{int} \vee \{\mathtt{int\ f}\} \vee \{\mathtt{int} \vee \{\mathtt{int\ f}\}\ \mathtt{f}\}\ \mathtt{f}\}\}$$
$$\ldots$$

- **Fixed-point exists**: $\{\mathtt{n} \mapsto \mathtt{int},\ \mathtt{m} \mapsto \mathtt{int}, \mathtt{x} \mapsto \mu X.\{\mathtt{int} \vee X\ \mathtt{f}\}\}$

Constraint-Based Flow Typing

- **Idea:** instead of dataflow-based algorithm, use a constraint-based one!

```
void loopy(int x, int y) {      // x0 ⊒ int, y0 ⊒ int,
                                // void ⊒ $
   z = {f : 1}^1                // z0 ⊒ {int f}
   while x < y^2 {              // z1 ⊒ z0 ⊔ z2, int ⊒ x0,
                                // int ⊒ y0
     z.f = z^3                  // z2 ⊒ z1[f ↦ z1]
} }
```

- First, extract constraints as above. Then, solve to find valid typings.

- Constraint variables numbered in style of **static single assignment**

Language of Constraints

$$\mathtt{c} ::= \mathtt{n}_\ell \sqsupseteq \mathtt{e} \mid \mathtt{T} \sqsupseteq \mathtt{e}$$
$$\mathtt{e} ::= \mathtt{T} \mid \mathtt{n}_\ell \mid \mathtt{e.f} \mid \mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2] \mid \bigsqcup \mathtt{e}_\mathtt{i}$$

Definition (Typing)

A typing, Σ, maps variables to types and *satisfies* a constraint set $\mathcal{C}$, denoted by $\Sigma \models \mathcal{C}$, if for all $\mathtt{e}_1 \sqsupseteq \mathtt{e}_2 \in \mathcal{C}$ we have $\mathcal{E}(\Sigma, \mathtt{e}_1) \geq \mathcal{E}(\Sigma, \mathtt{e}_2)$. Here, $\mathcal{E}(\Sigma, \mathtt{e})$ is defined as follows:

$$\mathcal{E}(\Sigma, \mathtt{T}) = \mathtt{T} \tag{1}$$
$$\mathcal{E}(\Sigma, \mathtt{n}_\ell) = \mathtt{T} \ \textbf{if} \ \{\mathtt{n}_\ell \mapsto \mathtt{T}\} \subseteq \Sigma \tag{2}$$
$$\mathcal{E}(\Sigma, \mathtt{e.f}) = \bigvee \mathtt{T}_\mathtt{i} \ \textbf{if} \ \mathcal{E}(\Sigma, \mathtt{e}) = \bigvee\{\ldots, \mathtt{T}_\mathtt{i}\ \mathtt{f}, \ldots\} \tag{3}$$
$$\mathcal{E}(\Sigma, \mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]) = \bigvee\{\overline{\mathtt{T}\ \mathtt{f}}\}[\mathtt{f} \mapsto \mathtt{T}] \ \textbf{if} \ \mathcal{E}(\Sigma, \mathtt{e}_1) = \bigvee\{\overline{\mathtt{T}\ \mathtt{f}}\} \ \textbf{and} \ \mathcal{E}(\Sigma, \mathtt{e}_2) = \mathtt{T} \tag{4}$$
$$\mathcal{E}(\Sigma, \bigsqcup \mathtt{e}_\mathtt{i}) = \bigvee \mathtt{T}_\mathtt{i} \ \textbf{if} \ \mathcal{E}(\Sigma, \mathtt{e}_1) = \mathtt{T}_1, \ldots, \mathcal{E}(\Sigma, \mathtt{e}_\mathtt{n}) = \mathtt{T}_\mathtt{n} \tag{5}$$

Constraint-Based Typing Rules

$$\frac{\vdash \mathtt{v} : \mathtt{T}}{\Gamma \vdash [\![\mathtt{n}=\mathtt{v}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \ell] \mid \{\mathtt{n}_{\ell} \sqsupseteq \mathtt{T}\}} \qquad \frac{\Gamma(\mathtt{m}) = \kappa}{\Gamma \vdash [\![\mathtt{n}=\mathtt{m}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \ell] \mid \{\mathtt{n}_{\ell} \sqsupseteq \mathtt{m}_{\kappa}\}}$$

$$\frac{\Gamma(\mathtt{m}) = \kappa}{\Gamma \vdash [\![\mathtt{n}=\mathtt{m.f}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \ell] \mid \{\mathtt{n}_{\ell} \sqsupseteq \mathtt{m}_{\kappa}.\mathtt{f}\}} \qquad \frac{\Gamma(\mathtt{n}) = \kappa \quad \Gamma(\mathtt{m}) = \lambda}{\Gamma \vdash [\![\mathtt{n.f}=\mathtt{m}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \ell] \mid \{\mathtt{n}_{\ell} \sqsupseteq \mathtt{n}_{\kappa}[\mathtt{f} \mapsto \mathtt{m}_{\lambda}]\}}$$

$$\frac{\Gamma(\mathtt{n}) = \kappa}{\Gamma \vdash [\![\mathtt{return\ n}]\!]^{\ell} : \emptyset \mid \{\$ \sqsupseteq \mathtt{n}_{\kappa}\}}$$

$$\frac{\begin{array}{c} \mathbf{defs}(\mathtt{B}) = \overline{\mathtt{n}} \\ \Gamma^1 = \Gamma^0[\overline{\mathtt{n} \mapsto \ell}] \quad \Gamma^1 \vdash \mathtt{B} : \Gamma^2 \mid \mathcal{C}_1 \\ \overline{\Gamma^0(\mathtt{n}) = \kappa} \quad \overline{\Gamma^2(\mathtt{n}) = \lambda} \\ \Gamma^1(\mathtt{n}) = \kappa \quad \Gamma^1(\mathtt{m}) = \lambda \\ \mathcal{C}_2 = \{\mathtt{int} \sqsupseteq \mathtt{n}_{\kappa}, \mathtt{int} \sqsupseteq \mathtt{m}_{\lambda}\} \\ \mathcal{C}_3 = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\overline{\mathtt{n}_{\ell} \sqsupseteq \mathtt{n}_{\kappa} \sqcup \mathtt{n}_{\lambda}}\} \end{array}}{\Gamma^0 \vdash \mathtt{while}\ [\![\mathtt{n} < \mathtt{m}]\!]^{\ell} \{\mathtt{B}\} : \Gamma^1 \mid \mathcal{C}_3}$$

- `defs(B)` returns variables assigned in `B`.

Variable Elimination

```
void loopy(int x, int y) {      // x_0 ⊒ int, y_0 ⊒ int,
                                // void ⊒ $
  z = {f : 1}^1                 // z_0 ⊒ {int f}
  while x < y^2 {               // z_1 ⊒ z_0 ⊔ z_2, int ⊒ x_0,
                                // int ⊒ y_0
    z.f = z^3                   // z_2 ⊒ z_1[f ↦ z_1]
} }
```

- To determine type for a variable, we **eliminate** all other variables by substitution

  E.g. given $\mathtt{n}_\ell \sqsupseteq \mathtt{e}$, eliminate $\mathtt{n}_\ell$ by substituting with $\mathtt{e}$

- After elimination, one constraint $\mathtt{n}_\ell \sqsupseteq \mathtt{e}$ remains, where $\mathtt{e}$ is either constant or expressed only in terms of $\mathtt{n}_\ell$

- May yield **recursive constraints**, e.g. $\mathtt{z}_1 \sqsupseteq \{\mathtt{int\ f}\} \sqcup \mathtt{z}_1[\mathtt{f} \mapsto \mathtt{z}_1]$

Type Extraction

- Elimination yields a **single constraint** for each variable
- From these constraints, must **extract** the typing for each variable

  E.g. from $\mathtt{n}_\ell \sqsupseteq \mathtt{int}$, type of $\mathtt{n}_\ell$ is $\mathtt{int}$

- Recursive constraints are **challenging**:

$$\mathtt{z}_1 \sqsupseteq \{\mathtt{int\ f}\} \sqcup \mathtt{z}_1[\mathtt{f} \mapsto \mathtt{z}_1]$$

- From above, must extract type $\mu\mathtt{X}.\{\mathtt{int} \vee \mathtt{X\ f}\}$ for $\mathtt{z}_1$

Limitations

- Unfortunately, the approach **does not work** in all cases:

```
void loopier(int x, int y) { // x₀ ⊒ int, y₀ ⊒ int, void ⊒ $
  z = {f : 1}¹               // z₀ ⊒ {int f}
  while x < y² {             // z₁ ⊒ z₀ ⊔ z₂, int ⊒ x₀,
                             // int ⊒ y₀
    z.f = z³                 // z₂ ⊒ z₁[f ↦ z₁]
  }
  while x < y² {             // z₃ ⊒ z₁ ⊔ z₄, int ⊒ x₀,
                             // int ⊒ y₀
    z.f = z³                 // z₄ ⊒ z₃[f ↦ z₃]
} }
```

- After elimination, we end up with this constraint for z_3:

$$z_3 \sqsupseteq \{\texttt{int f}\} \sqcup z_1[f \mapsto z_1] \sqcup z_3[f \mapsto z_3]$$

(where z_1 has not been successfully eliminated)

Conclusions

- Have considered a **specific** flow typing problem, which arose from developing Whiley
- Dataflow-based solution is easy to express and implement, but **does not terminate** in all cases
- Constraint-based solution is more involved, but is **guaranteed to terminate** in all cases
- Want to **extend** constraint-based approach to cover all cases...

http://whiley.org