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
```

- Groovy 2.0 includes flow-typing static checker
Another Example

- Non-null type checking provides another example:

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

- 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:

```plaintext
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:

```c
int f(int x) {
    y = 1
    z = {f: 1}
    while x < y { x = z.f
    return x
}
```

Syntax:

```
F ::= T f(T₁ n₁,...,Tₙ nₙ){B}
B ::= S B | ε
S ::= [n = v] ℓ | [n = m] ℓ | [n.f = m] ℓ | [n = m.f] ℓ | [return n] ℓ
     | while [n < m] ℓ {B}
V ::= {f₁:v₁,...,fₙ:vₙ} | i
```
Language of Types

Definition of types being considered:

\[ T ::= \text{void} \mid \text{any} \mid \text{int} \mid \{T_1 f_1, \ldots, T_n f_n\} \mid T_1 \lor T_2 \mid \mu X. T \mid X \]

Understanding recursive types:

\[ \mu X. \text{int} \lor \{X f\} \quad \{\mu X. \text{int} \lor \{X f\} f\} \]

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
Dataflow-Based Typing Rules

- Dataflow rules determine how environment is affected by statements:

\[ \Gamma \vdash v : T \]
\[ \Gamma \vdash [n=v]^{\ell} : \Gamma[n \mapsto T] \]
\[ \Gamma(m) = \{\ldots, T \ f, \ldots\} \]
\[ \Gamma \vdash [n=m.f]^{\ell} : \Gamma[n \mapsto T] \]
\[ \Gamma(n) \leq \Gamma($) \]
\[ \Gamma \vdash [\text{return } n]^{\ell} : \emptyset \]

\[ \Gamma(m) = v \]
\[ \Gamma \vdash [n=m]^{\ell} : \Gamma[n \mapsto v] \]
\[ \Gamma(n) = \{T_1 \ f_1, \ldots, T_n \ f_n\} \]
\[ T = \Gamma(n)[f \mapsto \Gamma(m)] \]
\[ \Gamma \vdash [n.f=m]^{\ell} : \Gamma[n \mapsto T] \]

\[ \Gamma_0 \sqcup \Gamma_1 \vdash B : \Gamma_1 \]
\[ \Gamma_0 \sqcup \Gamma_1(n) = \text{int} \quad \Gamma_0 \sqcup \Gamma_1(m) = \text{int} \]
\[ \Gamma_0 \vdash \text{while } [n < m]^{\ell} \{B\} : \Gamma_0 \sqcup \Gamma_1 \]

- Rule for \texttt{while} loops must iterate until \textbf{fixed point} reached
Consider this function:

```c
int ∨{int g} fun(int n, int m, int x) {
    while n < m {
        x = {g : 1}
        n = m
    }
    return x
}
```

Dataflow checker iterates this loop to produce type for x:

- \( \Gamma^1 = \{n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \text{int}\} \)
- \( \Gamma^1 = \{n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \text{int} \lor \{\text{int g}\}\} \)
- \( \Gamma^1 = \{n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \text{int} \lor \{\text{int g}\}\} \)

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**

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

- This causes dataflow-based checker to loop forever!

\[ \Gamma^3 = \{ n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \{ \text{int } f \} \} \]
\[ \Gamma^3 = \{ n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \{ \text{int } \lor \{ \text{int } f \} f \} \} \]
\[ \Gamma^3 = \{ n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \{ \text{int } \lor \{ \text{int } f \} \lor \{ \text{int } \lor \{ \text{int } f \} f \} f \} \} \]
... 

- **Fixed-point** exists: \( \{ n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \mu X.\{ \text{int } \lor X f \} \} \)
Constraint-Based Flow Typing

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

```c
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

\[ c ::= n_\ell \sqsubseteq e \mid T \sqsupseteq e \]
\[ e ::= T \mid n_\ell \mid e.f \mid e_1[f\mapsto e_2] \mid \bigcup e_i \]

Definition (Typing)

A typing, \( \Sigma \), maps variables to types and satisfies a constraint set \( C \), denoted by \( \Sigma \models C \), if for all \( e_1 \sqsupseteq e_2 \in C \) we have \( \mathcal{E}(\Sigma, e_1) \geq \mathcal{E}(\Sigma, e_2) \). Here, \( \Sigma(e) \) is defined as follows:

\[ \mathcal{E}(\Sigma, T) = T \]
\[ \mathcal{E}(\Sigma, n_\ell) = T \quad \text{if} \quad \{n_\ell \mapsto T\} \subseteq \Sigma \]
\[ \mathcal{E}(\Sigma, e.f) = \bigvee T_i \quad \text{if} \quad \mathcal{E}(\Sigma, e) = \bigvee\{\ldots, T_i \ f, \ldots\} \]
\[ \mathcal{E}(\Sigma, e_1[f\mapsto e_2]) = \bigvee\{T \ f\}[f \mapsto T] \quad \text{if} \quad \mathcal{E}(\Sigma, e_1) = \bigvee\{T \ f\} \ \text{and} \ \mathcal{E}(\Sigma, e_2) = T \]
\[ \mathcal{E}(\Sigma, \bigcup e_i) = \bigvee T_i \quad \text{if} \quad \mathcal{E}(\Sigma, e_1) = T_1, \ldots, \mathcal{E}(\Sigma, e_n) = T_n \]
Constraint-Based Typing Rules

\[
\Gamma \vdash \mathbf{n : T} \\
\Gamma \vdash \mathbf{[n = v]}^\ell : \Gamma[n \mapsto \ell] \upharpoonright \{n_\ell \sqsupseteq T\}
\]
\[
\Gamma(m) = \kappa \\
\Gamma \vdash \mathbf{[n = m]}^\ell : \Gamma[n \mapsto \ell] \upharpoonright \{n_\ell \sqsupseteq m_\kappa\}
\]
\[
\Gamma(n) = \kappa \\
\Gamma \vdash \mathbf{[n = m.f]}^\ell : \Gamma[n \mapsto \ell] \upharpoonright \{n_\ell \sqsupseteq m_\kappa.f\}
\]
\[
\Gamma(n) = \kappa \\
\Gamma \vdash \mathbf{[\text{return } n]}^\ell : \emptyset \upharpoonright \{\$, n_\kappa\}
\]
\[
\text{defs}(B) = \bar{n} \\
\Gamma^1 = \Gamma^0[n \mapsto \ell] \\
\Gamma^1 \vdash B : \Gamma^2 \upharpoonright C_1 \\
\Gamma^0(n) = \kappa \\
\Gamma^2(n) = \lambda \\
\Gamma^1(n) = \kappa \\
\Gamma^1(m) = \lambda \\
C_2 = \{\text{int} \sqsupseteq n_\kappa, \text{int} \sqsupseteq m_\lambda\} \\
C_3 = C_1 \cup C_2 \cup \{n_\ell \sqsupseteq n_\kappa \cup n_\lambda\}
\]
\[
\Gamma^0 \vdash \text{while } [n < m]_\ell \{B\} : \Gamma^1 \upharpoonright C_3
\]

\[\text{defs}(B) \text{ returns variables assigned in } B.\]
To determine type for a variable, we **eliminate** all other variables by substitution.

E.g. given $n_\ell \sqsubseteq e$, eliminate $n_\ell$ by substituting with $e$

After elimination, one constraint $n_\ell \sqsubseteq e$ remains, where $e$ is either constant or expressed only in terms of $n_\ell$

May yield **recursive constraints**, e.g. $z_1 \sqsubseteq \{\text{int } f\} \sqcup z_1[f \mapsto z_1]$
Type Extraction

- Elimination yields a **single constraint** for each variable.

- From these constraints, must **extract** the typing for each variable.

E.g. from \( n_\ell \sqsupseteq \text{int} \), type of \( n_\ell \) is \( \text{int} \).

- Recursive constraints are **challenging**:

\[
z_1 \sqsupseteq \{\text{int } f\} \sqcup z_1[f \mapsto z_1]
\]

- From above, must extract type \( \mu x.\{\text{int } \lor x \ f\} \) for \( z_1 \).
Limitations

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

```c
void loopier(int x, int y) { // x₀ ⊒ int, y₀ ⊒ int, void ⊒ $
    z = \{f : 1\}^1
    while x < y^2 { // z₀ ⊒ \{int f\}
        z.f = z^3
    }
    while x < y^2 { // z₁ ⊒ z₀ ∪ z₂, int ⊒ x₀,
        z.f = z^3
    }
```

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

\[
z_3 \supseteq \{\text{int } f\} \cup z_1[f \mapsto z_1] \cup z_3[f \mapsto z_3]
\]

(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...