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:

```java
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:

```whiley
int f = int
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:

```plaintext
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] | [n = m] | [n.f = m] | [n = m.f] | [return n]
    | while [n < m] { B }
V ::= {f1:v1, ..., fn:vn} | i
```
Language of Types

■ Definition of types being considered:

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

■ Understanding recursive types:

![Diagram of recursive types]

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

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

■ 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. These can be joined (e.g. $\Gamma_1 \sqcup \Gamma_2$).
Dataflow-Based Typing Rules

- Dataflow rules determine how environment is affected by statements:

\[
\begin{align*}
\Gamma \vdash v : T \\
\Gamma \vdash \lfloor n = v \rfloor^\ell : \Gamma[n \mapsto T]
\end{align*}
\]

\[
\begin{align*}
\Gamma(m) = \{\ldots, T \, f, \ldots\} \\
\Gamma \vdash \lfloor n = m \rfloor^\ell : \Gamma[n \mapsto T]
\end{align*}
\]

\[
\begin{align*}
\Gamma(n) = \{T_1 \, f_1, \ldots, T_n \, f_n\} \\
T = \Gamma(n)[f \mapsto \Gamma(m)] \\
\Gamma \vdash \lfloor n.f = m \rfloor^\ell : \Gamma[n \mapsto T]
\end{align*}
\]

\[
\begin{align*}
\Gamma(n) \leq \Gamma($) \\
\Gamma \vdash \lfloor \text{return} \, n \rfloor^\ell : \emptyset
\end{align*}
\]

\[
\begin{align*}
\Gamma_0 \sqcup \Gamma_1 \vdash B : \Gamma_1 \\
\Gamma_0 \sqcup \Gamma_1(n) = \text{int} \quad \Gamma_0 \sqcup \Gamma_1(m) = \text{int} \\
\Gamma_0 \vdash \text{while} \ \lfloor n < m \rfloor^\ell \{B\} : \Gamma_0 \sqcup \Gamma_1
\end{align*}
\]

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

Consider this function:

```c
 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:

\[ \Gamma^1 = \{n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \text{int}\} \]
\[ \Gamma^1 = \{n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \text{int} \vee \{\text{int} g\}\} \]
\[ \Gamma^1 = \{n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \text{int} \vee \{\text{int} g\}\} \]

So ... how do we know it always terminates?
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 example causes dataflow-based checker to loop forever!

```plaintext
Γ^3 = \{n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \{\text{int } f\}\}
Γ^3 = \{n \mapsto \text{int}, m \mapsto \text{int}, x \mapsto \{\text{int } \lor \{\text{int } f\} f\}\}
Γ^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\}\}
**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 \sqcup 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

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

Definition (Typing)

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

\[ \mathcal{E}(\Sigma, T) = T \]  \hspace{1cm} (1)
\[ \mathcal{E}(\Sigma, n_\ell) = T \quad \text{if} \quad \{n_\ell \mapsto T\} \subseteq \Sigma \]  \hspace{1cm} (2)
\[ \mathcal{E}(\Sigma, e.f) = \bigvee T_i \quad \text{if} \quad \mathcal{E}(\Sigma, e) = \bigvee \{\ldots, T_i \ f, \ldots\} \]  \hspace{1cm} (3)
\[ \mathcal{E}(\Sigma, e_1[f\mapsto e_2]) = \] \[ \bigvee \{T \ f[f\mapsto T]\} \quad \text{if} \quad \mathcal{E}(\Sigma, e_1) = \bigvee \{T \ f\} \quad \text{and} \quad \mathcal{E}(\Sigma, e_2) = T \]  \hspace{1cm} (4)
\[ \mathcal{E}(\Sigma, \bigcup e_i) = \bigvee T_i \quad \text{if} \quad \mathcal{E}(\Sigma, e_1) = T_1, \ldots, \mathcal{E}(\Sigma, e_n) = T_n \]  \hspace{1cm} (5)
Constraint-Based Typing Rules

\[ \Gamma \vdash v : T \]

\[ \Gamma \vdash [n = v]^\ell : \Gamma[n \mapsto \ell] \downarrow \{n_{\ell} \sqsupseteq T\} \]

\[ \Gamma(m) = \kappa \]

\[ \Gamma \vdash [n = m]^\ell : \Gamma[n \mapsto \ell] \downarrow \{n_{\ell} \sqsupseteq m_{\kappa}\} \]

\[ \Gamma(n) = \kappa \quad \Gamma(m) = \lambda \]

\[ \Gamma \vdash [n.f = m]^\ell : \Gamma[n \mapsto \ell] \downarrow \{n_{\ell} \sqsupseteq \nu_{\kappa}[f \mapsto m_{\lambda}]\} \]

\[ \Gamma(n) = \kappa \]

\[ \Gamma \vdash [\text{return } n]^\ell : \emptyset \downarrow \{n \sqsupseteq \nu_{\kappa}\} \]

\[ \text{defs}(B) = n \]

\[ \Gamma^1 = \Gamma^0[n \mapsto \ell] \quad \Gamma^1 \vdash B : \Gamma^2 \downarrow C_1 \]

\[ \Gamma^0(n) = \kappa \quad \Gamma^2(n) = \lambda \]

\[ \Gamma^1(n) = \kappa \quad \Gamma^1(m) = \lambda \]

\[ C_2 = \{\text{int} \sqsupseteq n_{\kappa}, \text{int} \sqsupseteq m_{\lambda}\} \]

\[ C_3 = C_1 \cup C_2 \cup \{n_{\ell} \sqsupseteq n_{\kappa} \sqcup n_{\lambda}\} \]

\[ \Gamma^0 \vdash \text{while } [n < m]^\ell \{B\} : \Gamma^1 \downarrow C_3 \]

\[ \text{defs}(B) \text{ returns variables assigned in } B. \]
void loopy(int x, int y) { // x_0 \subseteq \text{int}, y_0 \subseteq \text{int},
// void \subseteq $
// z_0 \subseteq \{\text{int } f\}$
// z_1 \subseteq z_0 \sqcup z_2, \text{int} \sqsubset x_0,
// \text{int} \sqsubset y_0
// z_2 \subseteq z_1[f \mapsto z_1]

z = \{f : 1\}^1
while x < y^2 { 
    z.f = z^3
}
}

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

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

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

- May yield **recursive constraints**, e.g. z_1 \sqsubset \{\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 $n_\ell \sqsubseteq \text{int}$, type of $n_\ell$ is int.

- Recursive constraints are **challenging**:

  $$z_1 \sqsubseteq \{\text{int \ } f\} \sqcup z_1[f \mapsto z_1]$$

- 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
  \text{while } x < y^2 \{ \\
    z.f = z^3
  \}
  \text{while } x < y^2 \{ \\
    z.f = z^3
  \}
}
```

- After elimination, we end up with this constraint for z₃:

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

(where z₁ 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