A type system for complexity flow analysis of imperative programs

Romain Péchoux
joint work with Jean-Yves Marion

Lorraine University
Inria project Carte, Loria

24 février 2012
Outline

Introduction

Ramified recursion and data flow

Secure flow typing

Informal treatment of Non-Interference

Examples

While Language and type system

Type soundness

Termination and Complexity

The conclusions
Implicit Computational Complexity (ICC)

The aim of ICC is to find machine independent characterizations of complexity classes:

- Function algebra (Bellantoni, Cook, Leivant ...)
- Linear Logic and lights logics (Girard, Lafont, Baillot, Gaboardi, Ronchi Della Rocca, ...)
- Interpretations methods for TRS (Marion, Péchoux, ...)
- Non-size increasing principle for HO functional programs (Hofmann, ...)
- Matrices flow type system for imperative programs (Jones, ...)
Simple While Language

Variables: $V ::= X | Y | \ldots$
Operators: $op ::= \text{Cons} | \text{Des}$
Expressions: $E ::= X | op(E)$
Commands: $C ::= X ::= E$
| if $E$ then $C$ else $C'$
| while $E$ do $C$
| $C; C'$

How to define a type system to control the computational complexity?
Main results

A type system for imperative programs such that:

- Terminating and typable programs are computable in polynomial time

- Each polynomial time function can be computed by a typed program

- Strongly normalizing multi-threads terminate in polynomial time

A multi-thread being a collection of programs running concurrently on a shared memory.
Ramified recursion and complexity

Tiers: \( \mathbb{N}(0), \mathbb{N}(1), \ldots, \mathbb{N}(k) \)

Ramification:
- \( g : \mathbb{N}(k) \rightarrow \mathbb{N}(0) \) and \( h : \mathbb{N}(k) \rightarrow \mathbb{N}(0) \rightarrow \mathbb{N}(0) \)
- Primitive recursion scheme:
  \[
  f(0, y) = g(y) \\
  f(x + 1, y) = h(x, f(x, y))
  \]
- \( f : \mathbb{N}(1) \rightarrow \mathbb{N}(k) \rightarrow \mathbb{N}(0) \)

Theorem (Bellantoni&Cook, Leivant)
The set of functions defined by ramified primitive recursion is exactly the set of polynomial time functions.
Example and counter-example

- Double:
  
  \[
  \begin{align*}
  \text{double}(0) &= 0 \\
  \text{double}(x + 1) &= 2 + \text{add}(x, y)
  \end{align*}
  \]

- \( double : \mathbb{N}(1) \rightarrow \mathbb{N}(0) \)

- Exponential:
  
  \[
  \begin{align*}
  \text{exp}(0) &= 1 \\
  \text{exp}(x + 1) &= \text{double}(\text{exp}(x))
  \end{align*}
  \]

- \( \text{exp} : \mathbb{N}(1) \rightarrow \mathbb{N}(1) \) but \( \mathbb{N}(1) \rightarrow \mathbb{N}(0) \) is required ! ! !

- There is a downward flow \( 1 \rightarrow 0 \)

- But no upward flow from \( 0 \rightarrow 1 \)
Ramification in imperative languages

Implicit flow from $x$ to $y$

```c
int copy(int x, int y)
{
    y = 0;
    while (x > 0)
    {
        x --;
        y ++;
    }
    return y;
}
```
Secure flow model

An information flow is defined by a lattice \((L, \leq)\) where \(L\) is a finite set of Security Classes.

\[ \text{Unclassified} < \text{Confidential} < \text{Secret} \]

Seminal works of Bell and LaPadula, Denning on security 
Biba’s model for integrity:

**Write down**
Subject \(S\) can write object \(O\) iff \(L(O) \leq L(S)\)

**Read up**
Subject \(S\) can read object \(O\) iff \(L(S) \leq L(O)\)
Secure flow typing

Implicit flow from $x$ to $y$

```c
int copy(int x, int y)
{
    y = 0;
    while (x > 0)
    {
        x --;
        y ++;
    }
    return y;
}
```

- Suppose that $\gamma(x) = 1$ and $\gamma(y) = 0$.
- Violation of the security law: No write down!
Flow policy

Informal treatment of Non-interference
Find an information flow type system to control complexity

Canonical Lattice ($\{0, 1\}, <$) with $0 < 1$

- There is no information flow from tier 0 to 1
- Tier 1 information controls loops
- Imperative tiers will correspond exactly to functional tiers
- The type system is dual to the secure information flow analysis of Volpano et al.
Typing Flow for complexity

**Type system**

\[ \gamma \vdash x : \tau \quad \text{if} \quad \gamma(x) = \tau \in \{0, 1\} \]
Typing Flow for complexity

Type system

\[ \gamma \vdash x : \tau \quad \text{if} \quad \gamma(x) = \tau \in \{0, 1\} \]

Destructors

\[ \gamma \vdash e : 0 \]

\[ \gamma \vdash e - 1 : 0 \]

\[ \gamma \vdash e : 1 \]

\[ \gamma \vdash e - 1 : 1 \]
Constructors

\[ \gamma \vdash e : 0 \]

\[ \frac{\gamma \vdash e : 0}{\gamma \vdash e + 1 : 0} \text{ OK} \]
Constructors

\[
\begin{align*}
\gamma \vdash e : 0 & \Rightarrow OK \\
\gamma \vdash e + 1 : 0 & \\
\gamma \vdash e : 1 & \Rightarrow NO \\
\gamma \vdash e + 1 : 1 &
\end{align*}
\]
Constructors

\[
\begin{align*}
\gamma \vdash e : 0 & \quad \text{OK} \\
\gamma \vdash e + 1 : 0 & \quad \text{NO} \\
\gamma \vdash e + 1 : 1 \\
\end{align*}
\]

Assign

\[
\begin{align*}
\gamma \vdash x : 0 & \quad \gamma \vdash e : 1 \\
\gamma \vdash x := e : 0 \\
\end{align*}
\]
Compose

\[ \gamma \vdash c : 0 \quad \gamma \vdash c' : 1 \]

\[ \gamma \vdash c ; c' : 1 \]
Compose

\[ \Gamma \vdash c : 0 \quad \Gamma \vdash c' : 1 \]

\[ \Gamma \vdash c ; c' : 1 \]

While

\[ \Gamma \vdash e : 1 \quad \Gamma \vdash c : 0 \]

\[ \Gamma \vdash \textbf{while} \ e \ \textbf{do} \ c : 1 \]
Examples: addition

```c
int add(int x, int y)
{
    while (x > 0)
    {
        x--;  
        y++;  
    }
    return y;
}
```

- $y$ is necessarily of tier 0
Examples: multiplication

```c
int mult(int x, int y)
{
    int z = 0;
    while (x)
    {
        x--;
        z = add(y, z);
    }
    return z;
}
```

- The output of add is 0
- Both x and y are of tier 1
Examples: exponential

```c
int expo(int x)
{
    int y = 1;
    while (x)
    {
        x--;  // x is of tier 1, but y is of tier 0
        y = add(y, y);
    }
    return y;
}
```

- x is of tier 1, but y is of tier 0
- The output of add is of tier 0
Operational semantics : Expressions

Base

\[ \mu \vdash n \Rightarrow n \]

Constants

\[ \mu \vdash x \Rightarrow \mu(x) \]

Operator

\[ \mu \vdash e \Rightarrow n \]

\[ \mu \vdash \text{op}(e) \Rightarrow [\text{op}](n) \]
Operational semantics: Commands

**Update**

\[
\mu \vdash e \Rightarrow n
\]

\[
\mu \vdash x := e \Rightarrow \mu[x \leftarrow n]
\]

**Sequence**

\[
\mu \vdash c \Rightarrow \mu' \quad \mu' \vdash c' \Rightarrow \mu''
\]

\[
\mu \vdash c; c' \Rightarrow \mu''
\]

**Branch**

\[
\mu \vdash e \Rightarrow \text{tt} \quad \mu \vdash c \Rightarrow \mu'
\]

\[
\mu \vdash \text{if } e \text{ then } c \text{ else } c' \Rightarrow \mu'
\]

\[
\mu \vdash e \Rightarrow \text{ff} \quad \mu \vdash c' \Rightarrow \mu''
\]

\[
\mu \vdash \text{if } e \text{ then } c \text{ else } c' \Rightarrow \mu''
\]
Operational semantics : Commands

**While**

\[
\mu \vdash e \Rightarrow \texttt{ff} \\
\mu \vdash \text{while } e \text{ do } c \Rightarrow \mu
\]

\[
\mu \vdash e \Rightarrow \texttt{tt} \quad \mu \vdash c \Rightarrow \mu' \quad \mu' \vdash \text{while } e \text{ do } c \Rightarrow \mu'' \\
\mu \vdash \text{while } e \text{ do } c \Rightarrow \mu''
\]

**Functions**
A notation for non-recursive definitions of functions
Functions

- A computation is given by a initial store $\mu$ and a command $c$. 
Functions

- A computation is given by an initial store $\mu$ and a command $c$.
- A computation ends if there is $\mu'$ s.t. $\mu \vdash c \Rightarrow \mu'$, otherwise it does not terminate.
Functions

- A computation is given by an initial store $\mu$ and a command $c$.
- A computation ends if there is $\mu'$ s.t. $\mu \vdash c \Rightarrow \mu'$, otherwise it does not terminate.
- A program is given by a code $c$, some input variables $x_1, \ldots, x_n$, and an output variable $y$. 

Functions

- A computation is given by a initial store $\mu$ and a command $c$.
- A computation ends if there is $\mu'$ s.t. $\mu \vdash c \Rightarrow \mu'$, otherwise it does not terminate.
- A program is given by a code $c$, some input variables $x_1, \ldots, x_n$, and an output variable $y$. 

Functions

- A computation is given by a initial store $\mu$ and a command $c$.
- A computation ends if there is $\mu'$ s.t. $\mu \vdash c \Rightarrow \mu'$, otherwise it does not terminate.
- A program is given by a code $c$, some input variables $x_1, \ldots, x_n$, and an output variable $y$. 

Functions
Functions

- A computation is given by a initial store $\mu$ and a command $c$.
- A computation ends if there is $\mu'$ s.t. $\mu \vdash c \Rightarrow \mu'$, otherwise it does not terminate.
- A program is given by a code $c$, some input variables $x_1, \ldots, x_n$, and an output variable $y$.
- A program computes a function $f$ iff
  
  $f(a_1, \ldots, a_n) = b$
  
  iff $\mu_0[x_i \leftarrow a_i]_{i \leq n} \vdash c \Rightarrow \mu'$ and $\mu'(y) = b$. 
Functions

- A computation is given by a initial store $\mu$ and a command $c$.
- A computation ends if there is $\mu'$ s.t. $\mu \vdash c \Rightarrow \mu'$, otherwise it does not terminate.
- A program is given by a code $c$, some input variables $x_1, \ldots, x_n$, and an output variable $y$.
- A program computes a function $f$ iff $f(a_1, \ldots, a_n) = b$ iff $\mu_0[x_i \leftarrow a_i]_{i\leq n} \vdash c \Rightarrow \mu'$ and $\mu'(y) = b$.
- A while-function is a function which is computable by a program.
Typing rules : expressions

**Variable**

\[ \gamma(x) = \tau \]

\[ \gamma \vdash x : \tau \]

**Data**

\[ \gamma \vdash n : \tau \]

**Destructor**

\[ \gamma \vdash e : \tau \]

\[ \gamma \vdash \text{op}(e) : \tau \]

**Constructor**

\[ \gamma \vdash e : \tau \]

\[ \gamma \vdash \text{op}(e) : \tau' \quad \tau' = 0 \]
Typing rules: commands

**Assign**

\[ \gamma \vdash x : \tau \quad \gamma \vdash e : \tau' \]

\[ \tau \leq \tau' \]

\[ \gamma \vdash x := e : \tau \]

**Compose**

\[ \gamma \vdash c : \tau \quad \gamma \vdash c' : \tau' \]

\[ \gamma \vdash c; c' : \tau \lor \tau' \]

**If**

\[ \gamma \vdash e : \tau \quad \gamma \vdash c : \tau \quad \gamma \vdash c' : \tau \]

\[ \gamma \vdash \text{if } e \text{ then } c \text{ else } c' : \tau \]

**While**

\[ \gamma \vdash e : 1 \quad \gamma \vdash c : \tau \]

\[ \tau < \tau' \]

\[ \gamma \vdash \text{while } e \text{ do } c : 1 \]
Simple Security

**Lemma**

If $\gamma \vdash e : \tau$, then for every variable $x$ in $e$, $\gamma(x) \geq \tau$.

**Démonstration.**

By induction on the structure of $e$. □
Simple Security

Lemma

If $\gamma \vdash e : \tau$, then for every variable $x$ in $e$, $\gamma(x) \geq \tau$.

Démonstration.

By induction on the structure of $e$.

- It says that if $e$ has level $\tau$, then every variable in $e$ stores information at level at least $\tau$. 
Lemma

If $\gamma \vdash e : \tau$, then for every variable $x$ in $e$, $\gamma(x) \geq \tau$.

Démonstration.

By induction on the structure of $e$.

- It says that if $e$ has level $\tau$, then every variable in $e$ stores information at level at least $\tau$.
- If $\tau = 1$, every variable in $e$ is of level 1.
Lemma

If $\gamma \vdash c : \tau$, then for every variable $x$ assigned to in $c$, $\gamma(x) \leq \tau$.

- It says that if $c$ has level $\tau$, then every variable assigned to in $c$ can be updated by information at level $\tau$.
- If $\tau = 0$, every variable assigned to in $c$ is of level $0$. 
Program equivalence

For a fixed typing environment $\gamma$:

- $c = d$ implies $c \approx d$
- $\gamma \vdash c : 0$ and $\gamma \vdash d : 0$ implies $c \approx d$
- $c \approx d$ and $c' \approx d'$ implies $c; c' \approx d; d'$
- $\mu \approx \sigma$ if for all $x$ s.t. $\gamma(x) = 1$, $\mu(x) = \sigma(x)$
- $\mu \approx \sigma$ and $c \approx d$ implies $\mu \vdash c \approx \sigma \vdash d$
Non-interference

Theorem

If

1. $\gamma \vdash c : \rho \text{ and } \gamma \vdash d : \rho$
2. $\mu \vdash c \approx \sigma \vdash d$
3. $\mu \vdash c \Rightarrow \mu'$

Then there exists $\sigma'$ s.t. $\sigma \vdash d \Rightarrow \sigma'$ and $\mu' \approx \sigma'$
Running time: Commands (1/2)

**Update**

\[
\mu \vdash e \Rightarrow n \\
\mu \vdash x := e \Rightarrow^0 \mu[x \leftarrow n]
\]

**Sequence**

\[
\mu \vdash c \Rightarrow^t \mu' \\
\mu' \vdash c' \Rightarrow^{t'} \mu'' \\
\mu \vdash c; c' \Rightarrow^{t+t'} \mu''
\]

**Branch**

\[
\mu \vdash e \Rightarrow tt \\
\mu \vdash c \Rightarrow^t \mu' \\
\mu \vdash if e then c else c' \Rightarrow^t \mu'
\]

\[
\mu \vdash e \Rightarrow ff \\
\mu \vdash c' \Rightarrow^{t} \mu'' \\
\mu \vdash if e then c else c' \Rightarrow^{t} \mu''
\]
Running time: Commands (2/2)

While

\[
\begin{align*}
    \mu \vdash e \Rightarrow \text{ff} \\
    \mu \vdash \text{while } e \text{ do } c \Rightarrow^0 \mu \\
    \mu \vdash e \Rightarrow \text{tt} \\
    \mu \vdash c \Rightarrow^t \mu' \quad \mu' \vdash \text{while } e \text{ do } c \Rightarrow^{t'} \mu'' \\
    \mu \vdash \text{while } e \text{ do } c \Rightarrow^{t+t'+1} \mu''
\end{align*}
\]
Temporal non-interference

Theorem

If

1. $\gamma \vdash c : \rho$ and $\gamma \vdash d : \rho$
2. $\mu \vdash c \approx \sigma \vdash d$
3. $\mu \vdash c \Rightarrow^t \mu'$

Then there exists $\sigma'$ s.t. $\sigma \vdash d \Rightarrow^t \sigma'$ and $\mu' \approx \sigma'$
Measuring time usage

Runtime of $c$ from $\mu$

$$\text{Time}_c(\mu) = \begin{cases} t & \mu \vdash c \Rightarrow^{t} \mu' \\ \text{undefined} & \text{otherwise} \end{cases}$$

A function $f$ is computed in polynomial time if there is a program $c$ and a polynomial $P$ s.t. for every $a_1, \ldots, a_n$,

$$\text{Time}_c(\mu_0(x_i \leftarrow a_i)) \leq P(|a_1|, \ldots, |a_n|)$$
Time soundness

Define

\[ \mu^{↑1}(x) = \begin{cases} \mu(x) & \gamma(x) = 1 \\ \text{undefined} & \text{otherwise} \end{cases} \]

- \( \text{Config}(\mu, c) = \{ (\mu', c) \mid \mu \vdash c \rightarrow^* \mu' \vdash c' \} \)
  (using a small step semantics)

- \( \text{Config}^{↑1}(\mu, c) = \{ (\mu'^{↑1}, c') \mid (\mu, c) \in \text{Config}(\mu, c) \} \)
Intermediate lemmata

Lemma
There is a constant $K$ s.t. for every store $\mu$,
\[
\text{Time}_c(\mu) \leq \begin{cases} 
K \cdot \#\text{Config}^{\uparrow 1}(c, \mu)^K & \mu \vdash c \Rightarrow \mu' \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

Lemma
Assume that $\gamma \vdash c : \tau$ over $(\{0, 1\}, \leq)$. There is $K'$ such that
\[
\#\text{Config}^{\uparrow 1}(c, \mu) \leq K' \cdot \sum_x |\mu^{\uparrow 1}(x)|
\]
Characterization of Ptime

Theorem

The set of functions computed by terminating and typed while programs is exactly the set of polynomial time computable functions.

Démonstration.

- Combining previous lemmata, a terminating while program is polynomial time computable.
- Conversely, every polynomial time computable function can be computed by a terminating while program using a simulation of your favorite model of computation.
Conclusions

- A flow type system to control time complexity of while-programs
- Scalable to Multi-threads (sets of commands sharing the same memory) including:
  - a polynomial time upper bound for strongly normalizing threads (ex: synchronization algorithms)
  - a polynomial time upper bound for weakly normalizing threads under a fixed fair scheduling policy (ex: round-robin scheduling)
- Operators expressivity can be improved
- Works in progress:
  - fork language (characterizing Pspace)
  - Include thread generation (Java applications)
  - Probabilistic scheduling policies