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

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

$$
\begin{array}{ll}
\textit{Variables :} & V ::= X | Y | \ldots \\
\textit{Operators :} & op ::= \mathrm{Cons} | \mathrm{Des} \\
\textit{Expressions :} & E ::= X | op(E) \\
\textit{Commands :} & C ::= X := E \\
 & \mid \text{if } E \text{ then } C \text{ else } C' \\
 & \mid \textbf{while } E \text{ do } C \\
 & \mid C; C'
\end{array}
$$

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 :

- $g : \mathbb{N}(k) \to \mathbb{N}(0)$ and $h : \mathbb{N}(k) \to \mathbb{N}(0) \to \mathbb{N}(0)$
- Primitive recursion scheme :

$$f(0, y) = g(y)$$
$$f(x+1, y) = h(x, f(x, y))$$

- $f : \mathbb{N}(1) \to \mathbb{N}(k) \to \mathbb{N}(0)$

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 :

$$double(0) = 0$$
$$double(x + 1) = 2 + add(x, y)$$

- $double : \mathbb{N}(1) \to \mathbb{N}(0)$
- Exponential :

$$exp(0) = 1$$
$$exp(x + 1) = double(exp(x))$$

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

- There is a downward flow $1 \to 0$
- But no upward flow from $0 \to 1$

Ramification in imperative languages

Implicit flow from *x* to *y*

```
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

```
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

$$\gamma \vdash x : \tau \qquad\qquad \textit{if } \gamma(x) = \tau \in \{0, 1\}$$

Typing Flow for complexity

Type system

$$\gamma \vdash x : \tau \qquad \text{if } \gamma(x) = \tau \in \{0, 1\}$$

Destructors

$$\frac{\gamma \vdash e : 0}{\gamma \vdash e - 1 : 0}$$

$$\frac{\gamma \vdash e : 1}{\gamma \vdash e - 1 : 1}$$

Constructors

$$\frac{\gamma \vdash e : 0}{\gamma \vdash e + 1 : 0} \; OK$$

Constructors

$$\frac{\gamma \vdash e : {\color{blue}0}}{\gamma \vdash e + 1 : {\color{blue}0}}\ OK$$

$$\frac{\gamma \vdash e : {\color{red}1}}{\gamma \vdash e + 1 : {\color{red}1}}\ NO$$

Constructors

$$\frac{\gamma \vdash e : 0}{\gamma \vdash e + 1 : 0} \; OK$$

$$\frac{\gamma \vdash e : 1}{\gamma \vdash e + 1 : 1} \; NO$$

Assign

$$\frac{\gamma \vdash x : 0 \qquad \gamma \vdash e : 1}{\gamma \vdash x := e : 0}$$

Compose

$$\frac{\gamma \vdash c : \color{blue}{0} \qquad \gamma \vdash c' : \color{red}{1}}{\gamma \vdash c; c' : \color{red}{1}}$$

Compose

$$\frac{\gamma \vdash c : \color{blue}{0} \qquad \gamma \vdash c' : \color{red}{1}}{\gamma \vdash c; c' : \color{red}{1}}$$

While

$$\frac{\gamma \vdash e : \color{red}{1} \qquad \gamma \vdash c : \color{blue}{0}}{\gamma \vdash \textbf{while } e \text{ do } c : \color{red}{1}}$$

Examples : addition

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

- y is necessarily of tier 0

Examples : multiplication


```
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

```
int expo(int x)
{
int y=1;
while (x)
          {
          x--;
          y = add(y,y);
          }
return y;
}
```

- x is of tier 1, but y ? ? ? ?
- The output of add is of tier 0

Operational semantics : Expressions

Base

$$\frac{}{\mu \vdash n \Rightarrow n}$$

Constants

$$\frac{}{\mu \vdash x \Rightarrow \mu(x)}$$

Operator

$$\frac{\mu \vdash e \Rightarrow n}{\mu \vdash \mathbf{op}(e) \Rightarrow [\![\mathbf{op}]\!](n)}$$

Operational semantics : Commands

Update

$$\frac{\mu \vdash e \Rightarrow n}{\mu \vdash x := e \Rightarrow \mu[x \leftarrow n]}$$

Sequence

$$\frac{\mu \vdash c \Rightarrow \mu' \qquad \mu' \vdash c' \Rightarrow \mu''}{\mu \vdash c; c' \Rightarrow \mu''}$$

Branch

$$\frac{\mu \vdash e \Rightarrow \mathbf{tt} \qquad \mu \vdash c \Rightarrow \mu'}{\mu \vdash \text{if } e \text{ then } c \text{ else } c' \ \Rightarrow \ \mu'}$$

$$\frac{\mu \vdash e \Rightarrow \mathbf{ff} \qquad \mu \vdash c' \Rightarrow \mu''}{\mu \vdash \text{if } e \text{ then } c \text{ else } c' \ \Rightarrow \ \mu''}$$

Operational semantics : Commands

While

$$\frac{\mu \vdash e \Rightarrow \mathbf{ff}}{\mu \vdash \mathbf{while}\ e\ \mathrm{do}\ c \ \Rightarrow\ \mu}$$

$$\frac{\mu \vdash e \Rightarrow \mathbf{tt} \qquad \mu \vdash c \ \Rightarrow\ \mu' \qquad \mu' \vdash \mathbf{while}\ e\ \mathrm{do}\ c \ \Rightarrow\ \mu''}{\mu \vdash \mathbf{while}\ e\ \mathrm{do}\ c \ \Rightarrow\ \mu''}$$

Functions
A notation for non-recursive definitions of functions

Functions

- A computation is given by a initial store μ and a command c.

Functions

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

Functions

- A computation is given by a initial store μ and a command c.
- A computation ends if there is μ' 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 μ and a command c.
- A computation ends if there is μ' 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 μ and a command c.
- A computation ends if there is μ' 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

$$\frac{\gamma(x) = \tau}{\gamma \vdash x : \tau}$$

Data

$$\frac{}{\gamma \vdash n : \tau}$$

Destructor

$$\frac{\gamma \vdash e : \tau}{\gamma \vdash \mathbf{op}(e) : \tau}$$

Constructor

$$\frac{\gamma \vdash e : \tau}{\gamma \vdash \mathbf{op}(e) : \tau'} \qquad \tau' = 0$$

Typing rules : commands

Assign

$$\frac{\gamma \vdash x : \tau \qquad \gamma \vdash e : \tau'}{\gamma \vdash x := e : \tau}\ \tau \leq \tau'$$

Compose

$$\frac{\gamma \vdash c : \tau \qquad \gamma \vdash c' : \tau'}{\gamma \vdash c; c' : \tau \vee \tau'}$$

If

$$\frac{\gamma \vdash e : \tau \qquad \gamma \vdash c : \tau \qquad \gamma \vdash c' : \tau}{\gamma \vdash \text{if } e \text{ then } c \text{ else } c' : \tau}$$

While

$$\frac{\gamma \vdash e : 1 \qquad \gamma \vdash c : \tau}{\gamma \vdash \textbf{while } e \text{ do } c : 1}\ \tau < \tau'$$

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*. $\square$

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*. $\square$

- It says that if *e* has level τ, then every variable in *e* stores information at level at least τ.

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. $\square$

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

Confinement

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 τ, then every variable assigned to in c can be updated by information at level τ.
- If $\tau = 0$, every variable assigned to in c is of level 0.

Program equivalence

For a fixed typing environment γ :

- $c = d$ 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. $\gamma \vdash c : \rho$ *and* $\gamma \vdash d : \rho$
2. $\mu \vdash c \approx \sigma \vdash d$
3. $\mu \vdash c \Rightarrow \mu'$

Then there exists σ' *s.t.* $\sigma \vdash d \Rightarrow \sigma'$ *and* $\mu' \approx \sigma'$

Running time : Commands (1/2)

Update

$$\frac{\mu \vdash e \Rightarrow n}{\mu \vdash x := e \Rightarrow^0 \mu[x \leftarrow n]}$$

Sequence

$$\frac{\mu \vdash c \Rightarrow^t \mu' \qquad \mu' \vdash c' \Rightarrow^{t'} \mu''}{\mu \vdash c; c' \Rightarrow^{t+t'} \mu''}$$

Branch

$$\frac{\mu \vdash e \Rightarrow \mathbf{tt} \qquad \mu \vdash c \Rightarrow^t \mu'}{\mu \vdash \text{if } e \text{ then } c \text{ else } c' \ \Rightarrow^t \ \mu'}$$

$$\frac{\mu \vdash e \Rightarrow \mathbf{ff} \qquad \mu \vdash c' \Rightarrow^t \mu''}{\mu \vdash \text{if } e \text{ then } c \text{ else } c' \ \Rightarrow^t \ \mu''}$$

Running time : Commands (2/2)

While

$$\frac{\mu \vdash e \Rightarrow \mathbf{ff}}{\mu \vdash \mathbf{while}\ e\ \mathrm{do}\ c \ \Rightarrow^{0}\ \mu}$$

$$\frac{\mu \vdash e \Rightarrow \mathbf{tt} \qquad \mu \vdash c \ \Rightarrow^{t}\ \mu' \qquad \mu' \vdash \mathbf{while}\ e\ \mathrm{do}\ c \ \Rightarrow^{t'}\ \mu''}{\mu \vdash \mathbf{while}\ e\ \mathrm{do}\ c \ \Rightarrow^{t+t'+1}\ \mu''}$$

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 σ' *s.t.* $\sigma \vdash d \Rightarrow^{t} \sigma'$ *and* $\mu' \approx \sigma'$

Measuring time usage

Runtime of c from μ

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

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

$$Time_c(\mu_0(x_i \leftarrow a_i)) \leq P(|a_1|, \ldots, |a_n|)$$

Time soundness

Define

$$\mu^{\uparrow 1}(x) = \begin{cases} \mu(x) & \gamma(x) = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

- $\mathit{Config}(\mu, c) = \{(\mu', c) \mid \mu \vdash c \rightarrow^* \mu' \vdash c'\}$ (using a small step semantics)
- $\mathit{Config}^{\uparrow 1}(\mu, c) = \{(\mu'^{\uparrow 1}, c') \mid (\mu, c) \in \mathit{Config}(\mu, c)\}$

Intermediate lemmata

Lemma

There is a constant K s.t. for every store μ,

$$Time_c(\mu) \leq \begin{cases} K.\#Config^{\uparrow 1}(c,\mu)^K & \mu \vdash c \Rightarrow \mu' \\ undefined & otherwise \end{cases}$$

Lemma

Assume that $\gamma \vdash c : \tau$ over $(\{0, 1\}, \leq)$.
There is K' such that

$$\#Config^{\uparrow 1}(c,\mu) \leq K'.\sum_x |\mu^{\uparrow 1}(x)|$$

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