Type systems for ICC analysis of imperative programs

Romain Péchoux

Université de Lorraine, LORIA, Nancy, France

Séminaire of the team Carte

ICC's last stand

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

- Function algebra (Bellantoni, Cook, Leivant, Marion, ...)
- Lights logics (Girard, Lafont, Baillot, Gaboardi, Ronchi Della Rocca, ...)
- Interpretations of TRS (Bonfante, Marion, Moyen, Péchoux, ...)
- Non-size increasing principle (Hofmann, ...)
- Matrices calculus for imperative programs(Jones, Kristiansen, Wunderlich, Moyen,...)
- Imperative pointer graph languages for subpolynomial classes (Hofmann, Schoepp, ...)

Mixture

Marion's idea (Lics 2011) is to take advantage of two well-known lines of work:

- Safe (or tiered) recursion by Bellantoni and Cook [1992]
- Non-interference by Volpano et al [1996]

in order to obtain a polynomial time characterization on imperative languages.

Safe recursion

The class of functions that can be defined using:

- constants, projections, successor, predecessor, conditional,
- safe composition:

$$f(\overline{x};\overline{a}) = h(r(\overline{x};);t(\overline{x};\overline{a}))$$

- and safe recursion (on notation):

$$\begin{aligned} f(0,\overline{x};\overline{a}) &= g(\overline{x};\overline{a}) \\ f(i(x),\overline{y};\overline{a}) &= h_i(x,\overline{y};f(x,\overline{y};\overline{a})) \qquad i \in \{0,1\}, \end{aligned}$$

provided h, r, t, g, h_i are already defined in the class,

is exactly the set of functions computable in polynomial time (FPtime).

The tiered viewpoint

The class of functions that can be defined using:

- constants, projections, successor, predecessor, conditional,
- safe composition:

$$f(\overline{x}^{\mathbf{1}}; \overline{a}^{\mathbf{0}}) = h(r(\overline{x}^{\mathbf{1}};); t(\overline{x}^{\mathbf{1}}; \overline{a})^{\mathbf{0}})$$

- and safe recursion (on notation):

$$f(0, \overline{x}^{\mathbf{1}}; \overline{a}^{\mathbf{0}}) = g(\overline{x}^{\mathbf{1}}; \overline{a}^{\mathbf{0}})$$
$$f(i(x)^{\mathbf{1}}, \overline{y}^{\mathbf{1}}; \overline{a}) = h_i(x^{\mathbf{1}}, \overline{y}^{\mathbf{1}}; f(x^{\mathbf{1}}, \overline{y}^{\mathbf{1}}; \overline{a})^{\mathbf{0}}) \qquad i \in \{0, 1\},$$

  provided h, r, t, g, h_i are already defined in the class,

is exactly the set of functions computable in polynomial time (FPtime).

Non-interference

Two security levels:

- H for high
- L for low

and typing rules of the shape:

$$\frac{\Gamma \vdash E : \tau \quad \Gamma \vdash I : \tau\ Cmd}{\Gamma \vdash \texttt{while}(E)\{I\} : \tau\ Cmd}\ (Wh)$$

+ command subtyping:

$$\frac{\Gamma \vdash I : \tau\ Cmd \quad \tau < \tau'}{\Gamma \vdash I : \tau'\ Cmd}\ (Sub)$$

Non-interference example

It prevent us from typing the following program:

```
while(x>0 : H) {
    x = x-- ; : H Cmd
    y = y++ ; : L Cmd
}
```

if x is High and y is Low (Indeed there is a flow from x to y) and provided that $H < L$.

Duality of non-interference and tiering

We would like to type following program:

```
while(x>0 : 1) {
    x = x-- ; : 1 Cmd
    y = y++ ; : 0 Cmd
}
```

if x is of tier **1** (High) and y is of tier **0** Low (preventing flows from y to x) and provided that $\mathbf{0} < \mathbf{1}$.

Small imperative language

Every data type is encoded by words over $\mathbf{W}$.
The size $|w|$ of a word $w \in \mathbf{W}$ is standard.

- Expressions :
  $E ::= \quad \mathtt{x} \mid c \mid \mathtt{true} \mid \mathtt{false} \mid op(\overline{E})$
- Instructions :
  $I ::= \quad ; \mid [\tau]\ \mathtt{x}{:=}E; \mid I_1\ I_2 \mid \mathtt{while}(E)\{I\}$
  $\mid \mathtt{if}(E)\{I_1\}\mathtt{else}\{I_2\}$

The types τ will be tiers in $\{\mathbf{0}, \mathbf{1}\}$ such that $\mathbf{0} < \mathbf{1}$.

Typing rules : expressions

Variable

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

Constant

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

Destructor

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

Constructor

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash op(e) : \mathbf{0}}$$

Typing rules : commands

Assign

$$\frac{\Gamma \vdash \mathtt{x} : \tau \qquad \Gamma \vdash E : \tau'}{\Gamma \vdash \mathtt{x} := E : \tau} \tau \leq \tau'$$

Compose

$$\frac{\Gamma \vdash I_1 : \tau \qquad \Gamma \vdash I_2 : \tau'}{\Gamma \vdash I_1 \ I_2 : \tau \vee \tau'}$$

If

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash I_i : \tau}{\Gamma \vdash \mathtt{if}(E)\{I_1\}\mathtt{else}\{I_2\} : \tau}$$

While

$$\frac{\Gamma \vdash E : \mathbf{1} \qquad \Gamma \vdash I : \tau}{\Gamma \vdash \mathtt{while}(E)\{I\} : \mathbf{1}}$$

Improvements

We can extend the type system to more general operators $op :: \tau_1 \times \ldots \times \tau_n \to \tau$ such that $\tau \leq \wedge_i \tau_i$.

- Neutral operators:
  - either a computable predicate
  - or a subword operator:

$$\forall \overline{w},\ \exists i \in \{1, \ldots, n\},\ [\![op]\!](\overline{w}) \trianglelefteq w_i$$

- Positive operators:

$$\forall \overline{w},\ |[\![op]\!](\overline{w})| \leq \max_{i \in [1,n]} |w_i| + c,\ \text{for } c \geq 0$$

  - In this case $\tau = \mathbf{0}$.

We can also add procedure calls.

Example: addition

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

- y is necessarily of tier **0**
- x is necessarily of tier **1**
- and, consequently, add :: $\mathbf{1} \times \mathbf{0} \to \mathbf{0}$

Example: 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**. Consequently, z is of tier **0**.
- both x and y are of tier **1**
- and, consequently, `mult` :: $\mathbf{1} \times \mathbf{1} \to \mathbf{0}$

Example: exponential

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

- x is of tier **1**,
- the output of add is of tier **0**,
- but y has to be of tier **1** in the first argument of add !!!

Results

We have a (weak) subject reduction property:

Theorem [Marion and Péchoux (TAMC 2014?)]

If $\sigma \vDash l \to \sigma' \vDash l'$ and $\Gamma \vdash l : \tau$ then $\Gamma \vdash l' : \tau'$ where $\tau' \leq \tau$.

We obtain a characterization of FPtime:

Theorem [Marion (Lics 2011)]

The set of functions computable by a typable and terminating program with FPtime computable operators is exactly FPtime.

Moreover, type inference is decidable:

Theorem [Hainry, Marion and Péchoux (Fossacs 2013)]

Type inference can be done in polynomial time.

Mechanism

FPtime soundness:

- No flow from **0** to **1**: tier **1** variables cannot increase
- Only tier **1** arguments in the guards
- At most n^k configurations under termination assumption

FPtime completeness:

- Any polynomial can be computed
- We simulate polynomial time TMs by an imperative typable (and terminating) program

Type inference:

- All the constraints are inequalities over 2 tiers
- That can be reduced to a 2-SAT formula

Multi-threaded

Now we consider multi-threads M to be a fixed collection of commands:

$$M(\alpha) = I,\ \alpha \in dom(M)$$

and non-deterministic reduction:

$$\frac{M(\alpha) = I \quad \sigma \vDash I \to \sigma_1 \vDash I_1}{\sigma \vDash M \ \to\ \sigma_1 \vDash M[\alpha := I_1]}\ (Step) \qquad \frac{M(\alpha) = I \quad \sigma \vDash I \to \sigma_1}{\sigma \vDash M \ \to\ \sigma_1 \vDash M - \alpha}\ (Stop)$$

and we extend the typing rule by:

$$\frac{\forall \alpha \in dom(M),\ \exists \tau,\ \Gamma \vdash M(\alpha) : \tau}{\Gamma \vdash M : \diamond}\ (Multi)$$

Results

We obtain a polynomial time soundness criterion:

Theorem [Marion and Péchoux (TAMC 2014?)]

A typable and strongly normalizing multi-thread terminates in a polynomially bounded number of transitions.

The strong normalization assumption can be weakened under a fair scheduling policy (depending only on M and tier **1** values):

Theorem [Marion and Péchoux (TAMC 2014?)]

A typable a multi-thread terminating under a fair scheduling policy terminates in a polynomially bounded number of transitions.

Moreover, type inference remains decidable:

Theorem [Hainry, Marion and Péchoux (Fossacs 2013)]

Type inference can be done in polynomial time.

Forks: motivation

- May the analysis be generalized to more expressive languages ?
- Can we analyze parallelism ?
- Is it possible to jump from time (FPtime) to space (Pspace or FPspace) ?

In [Fossacs 2013], we have presented an extension to forks.
The syntax of the language is extended by two commands:

$$X = \texttt{fork}() \mid X = \texttt{wait}\{E\}$$

Forks informal semantics

On the execution of $X = \mathit{fork}(); I$ in a parent process:

- a new son of (fresh) pid n and instruction I is created (by default, $X := 0$)
- the father has instruction I and knows the pid of its new son ($X := n$)

On the execution of $X = \mathit{wait}(E); I$ in a parent process:

- if E evaluates to n and the process of pid n returns v then $X := v$ in the parent process
- otherwise the father has to wait.

Forks typing rules

We need to add an extra tier $-\mathbf{1}$ ($-\mathbf{1} < \mathbf{0} < \mathbf{1}$) in order to prevent accumulation.

$$\frac{\Gamma \vdash \mathtt{x} : \mathbf{0}}{\Gamma \vdash \mathtt{x{:=}fork()} : \mathbf{0}}\ (F) \qquad \frac{\Gamma \vdash E : \mathbf{0} \quad \Gamma \vdash \mathtt{x} : -\mathbf{1}}{\Gamma \vdash \mathtt{x{:=}wait}(E) : -\mathbf{1}}\ (W)$$

Operators $op :: \tau_1 \times \ldots \times \tau_n \to \tau$ are extended to max operations :

$$\forall \overline{w},\ |[\![op]\!](\overline{w})| \leq \max_{i \in [1,n]} |w_i|$$

provided that $\tau < \mathbf{1}$.

- It means that forks' pid cannot be used as guards
- The values returned by sons cannot be accumulated (at most max or neutral operators).

Example "rien que pour les yeux"

```
max_reduce(n^1, A^0) ::= r^0 := 0: 0; f^-1 := A[r]^0: -1;
  flag^0 := tt: 0;
  while (n^1 ≠ 1)^1 do {
      if flag^0 then {     // not finished
          pidl^0 := fork(): 0
          if (pidl>0)^0 then {     // father process
              r^0 := 2*r+2: 0;
              pidr^0 := fork(): 0}
              else { r^0 := 2*r+1: 0 }    // left son
          if (pidr==0)^0 or (pidl==0)^0 then { f^-1 := A[r]^0:
              else {
                  flag^0 := ff: 0;    // father
                  xl^-1 := wait(pidl): 0;
                  xr^-1 := wait(pidr): 0;
                  f^-1 := max(f^-1, max(xl,xr)): 0; } }
      n^1 := half(n)^1: 1 }  // end of while
    return f:-1
```

Results

We obtain a characterization of Pspace computable functions:

Theorem [Hainry, Marion and Péchoux (Fossacs 2013)]

The set of functions computed by typable, strongly normalizing and confluent processes is exactly the set of polynomial space computable functions *FPspace*.

Soundness:

- As for multi-threads, the computation tree has a polynomially bounded depth
- tier -1 prevents accumulation
- the considered programs are confluent, consequently, we can perform a "in depth" evaluation

Completeness:

- Each FPspace function can be bitwise computed
- We show that QBF can be encoded and typed in our formalism.

OO State of the art

Some techniques and programs to bound resource consumptions

- Amortised analysis for linear heap (Hofmann & Jost)
- "Costa" for analyzing Java bytecode (Albert, Arenas, Genaim, Puebla & Zanardini)
- "Speed" for C++ (Gulwani *et al.*)
- "ResAna" analyzes Java programs (Shkaravska *et al.*)
- Non-interference and tiering for a graph based imperative language (Leivant & Marion)

OO: motivation

- Extend our results to a "daily-life" real programming language
- Analyze the complexity of the OO paradigm
- Obtain "practical" upper bound on both the heap and stack space usage
- Analyze OO features:
  - mixture of while loops and recursive method calls
  - objects in loop guards
  - inheritance
  - control flow statements such as `break` or `continue`

Core Java

In [FOPARA2013], we have considered the Java-like language:

- Expressions $E ::= \ldots \mid \texttt{null} \mid \texttt{this} \mid \texttt{new C}(\overline{E}) \mid E.m(\overline{E})$
- Instructions $I ::= \ldots \mid E.m(\overline{E});$
- Methods $M_C ::= \tau\ m(\tau_1\ \texttt{x}_1, \ldots, \tau_n\ \texttt{x}_n)\{I[\texttt{return x;}]\}$
- Cons $K_C ::= C(\tau_1\ \texttt{y}_1, \ldots, \tau_n\ \texttt{y}_n)\{\texttt{x}_1{:=}\texttt{y}_1; \ldots \texttt{x}_n{:=}\texttt{y}_n;\}$
- Classes $\mathfrak{C} ::= C\{\tau_1\ \texttt{x}_1; \ldots; \tau_n\ \texttt{x}_n;\ K_C\ M_C^1 \ldots M_C^k\}$

Core Java Programs

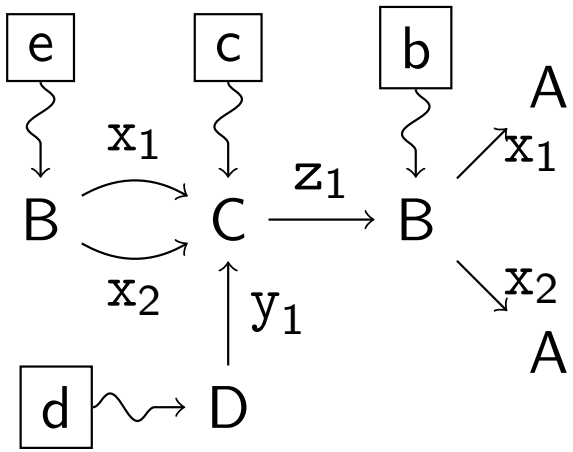
Definition [Core Java Program]

A Core Java Program is a collection of classes and exactly one executable:

$$\mathrm{Exe}\{\texttt{main}()\{\underbrace{\tau_1\ \mathtt{x}_1 := E_1; \ldots; \tau_n\ \mathtt{x}_n := E_n;}_{\text{Initialization}}\ \underbrace{I}_{\textit{Computation}}\ \}\}.$$

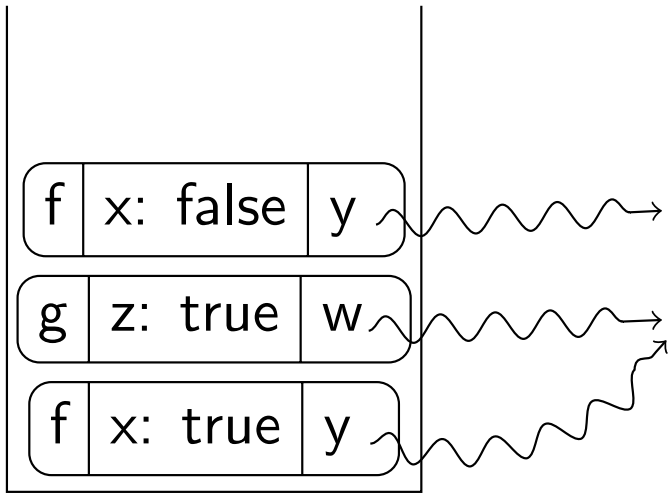
Heap

- Where objects are created and kept in memory.
- Maximal heap space is defined at the launch of the JVM.
- Pointers to the objects, arrows between objects and their attributes.



Stack

- Where arguments of a method call are put.
- Primitive types are put by value.
- Object types are put by reference, *i.e.* a pointer to the heap.
- May grow indefinitely because of recursive calls.



Tiered types

- Expressions, Instructions, Constructors and Methods are annotated by tiered types (i.e. a type and a tier (0 or 1)).
- For instructions, the type will always be `void`.
- For Constructors and methods the tiered type is functional:

$$\texttt{boolean}(1) \times \texttt{BList}(1) \to \texttt{BList}(0)$$

- For methods, the tiered type of the caller object is included: e.g. for `void setQueue(BList q) {...}`

$$\texttt{BList}(0) \times \texttt{BList}(1) \to \texttt{void}(0)$$

Typing Simple Expressions

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{boolean}(\mathbf{1})}\ (True) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{boolean}(\mathbf{1})}\ (False)$$

$$\frac{}{\Gamma \vdash \texttt{null} : \mathsf{C}(\mathbf{1})}\ (Null)$$

$$\frac{\alpha \preceq \min\{\text{tiers of the attributes}\}}{(m^{\mathsf{C}}, \Delta) \vdash \texttt{this} : \mathsf{C}(\alpha)}\ (Self) \qquad \frac{\Delta(m^{\mathsf{C}})(\mathtt{x}) = \tau(\alpha)}{(m^{\mathsf{C}}, \Delta) \vdash \mathtt{x} : \tau(\alpha)}\ (Var)$$

$$\frac{\forall i\ \Gamma \vdash E_i : \tau_i(\alpha) \qquad op :: \tau_1 \times \cdots \times \tau_n \to \texttt{boolean}}{\Gamma \vdash op(E_1, \ldots, E_n) : \texttt{boolean}(\alpha)}\ (Op)$$

Typing Instructions

$$\frac{}{\Gamma \vdash ; : \mathtt{void}(\mathbf{0})}\ (Skip)$$

$$\frac{\Gamma \vdash \mathtt{x} : \tau(\alpha) \qquad \Gamma \vdash E : \tau(\beta) \quad \alpha \preceq \beta}{\Gamma \vdash [\tau]\ \mathtt{x}{:=}E; : \mathtt{void}(\alpha)}\ (Ass)$$

$$\frac{\Gamma \vdash I : \mathtt{void}(\alpha) \quad \alpha \preceq \beta}{\Gamma \vdash I : \mathtt{void}(\beta)}\ (Sub)$$

$$\frac{\forall i\ \Gamma \vdash I_i : \mathtt{void}(\alpha_i)}{\Gamma \vdash I_1\ I_2 : \mathtt{void}(\alpha_1 \vee \alpha_2)}\ (Seq)$$

$$\frac{\Gamma \vdash E : \mathtt{boolean}(\alpha) \quad \forall i\ \Gamma \vdash I_i : \mathtt{void}(\alpha)}{\Gamma \vdash \mathtt{if}(E)\{I_1\}\mathtt{else}\{I_2\} : \mathtt{void}(\alpha)}\ (If)$$

$$\frac{\Gamma \vdash E : \mathtt{boolean}(\mathbf{1}) \quad \Gamma \vdash I : \mathtt{void}(\mathbf{1})}{\Gamma \vdash \mathtt{while}(E)\{I\} : \mathtt{void}(\mathbf{1})}\ (Wh)$$

Typing Constructors

$$\frac{\forall i\ (m^{\mathsf{C}}, \Delta) \vdash E_i : \tau_i(\beta_i) \quad \alpha_i \preceq \beta_i \qquad (\epsilon, \Delta) \vdash \mathsf{C}(\ldots \tau_i\ \mathtt{y}_i \ldots)\{\ldots \mathtt{x}_i := \mathtt{y}_i; \ldots\} : \cdots \times \tau_i(\alpha_i) \times \cdots \to \mathsf{C}(\mathbf{0})}{(m^{\mathsf{C}}, \Delta) \vdash \mathtt{new}\ \mathsf{C}(E_1, \ldots, E_n) : \mathsf{C}(\mathbf{0})}\ (New)$$

$$\frac{\forall i\ (\epsilon, \Delta) \vdash \mathtt{y}_i : \tau_i(\alpha_i)}{(\epsilon, \Delta) \vdash \mathsf{C}(\ldots, \tau_i\ \mathtt{y}_i, \ldots)\{\ldots \mathtt{x}_i := \mathtt{y}_i; \ldots\} : \cdots \times \tau_i(\alpha_i) \times \cdots \to \mathsf{C}(\mathbf{0})}\ (K_C)$$

Constructors make the heap increase, hence output something of tier **0**.

Safety assumption

Definition [Safety]

A well-typed program with respect to a typing environment Δ is safe if for each recursive method $M_C = \tau\ m(\ldots)\{I\ [\texttt{return x;}]\}$:

- there is exactly one call (even nested) to m,
- there is no while loop inside I,
- and the following judgment can be derived:

$$(\epsilon, \Delta) \vdash M_C : \mathsf{C}(\mathbf{1}) \times \tau_1(\mathbf{1}) \times \cdots \times \tau_n(\mathbf{1}) \to \tau(\mathbf{1}).$$

Results

Theorem [Hainry and Péchoux]

In the execution of a safe Core Java program terminating on input $\mathcal{C}$, the size of the heap and of the stack are in $O(|\mathcal{C}|^{n_1((\nu+1)\lambda)})$.

- n_1 the number of variables and attributes of tier **1**,
- λ the maximum number of nested while and
- ν the maximum number of nested methods.

We are still complete wrt FPtime and type inference is decidable:

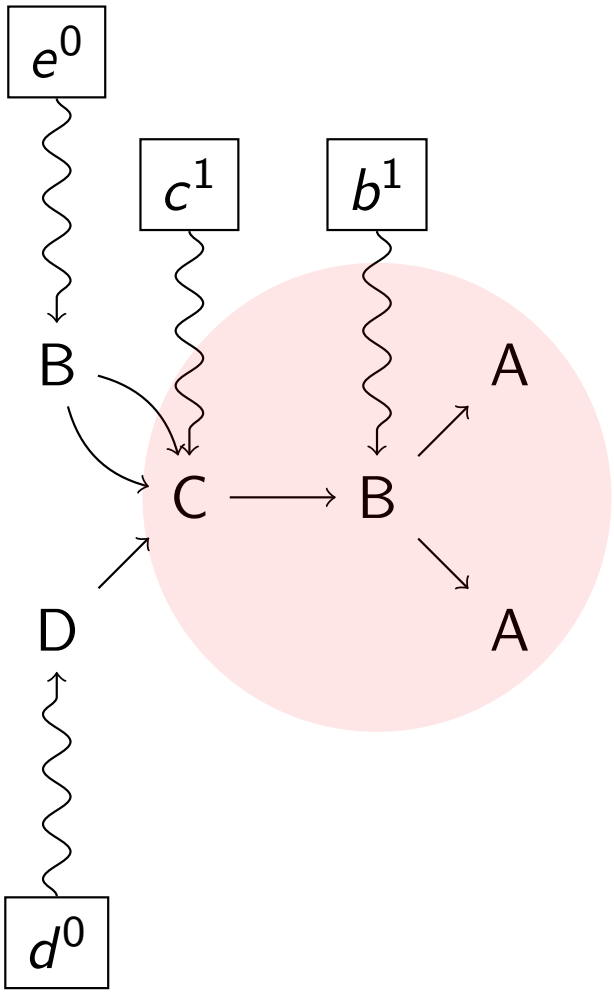
Proposition [Hainry and Péchoux]

The set of functions computable by typable, safe and terminating programs is exactly FPtime

Proposition [Type inference]

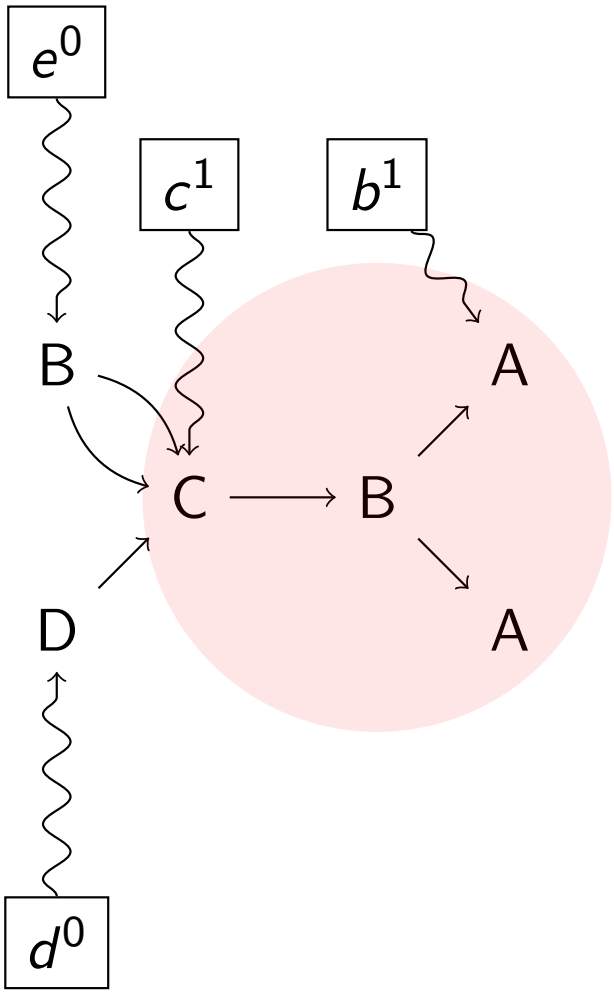
The type inference can be done in time linear in the size of the program.

Idea of the proof



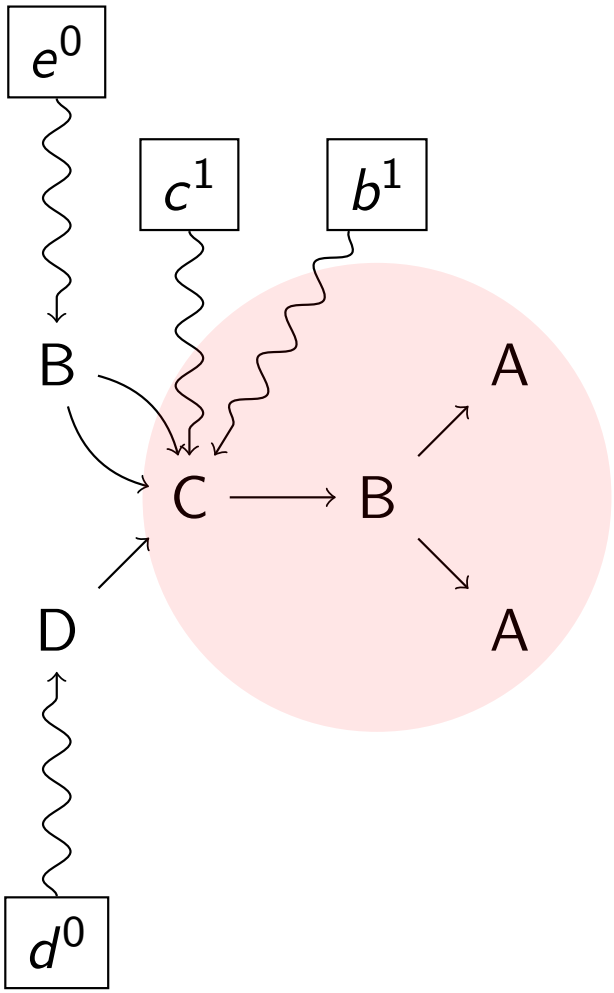
- The subheap of tier **1** never grows.
- Only tier **1** variables control `while` and recursive functions.
- The number of tier **1** configurations is bounded by $|\mathcal{C}|^{2\times n_1}$.
- Hence a bound on the stack and heap.

Idea of the proof



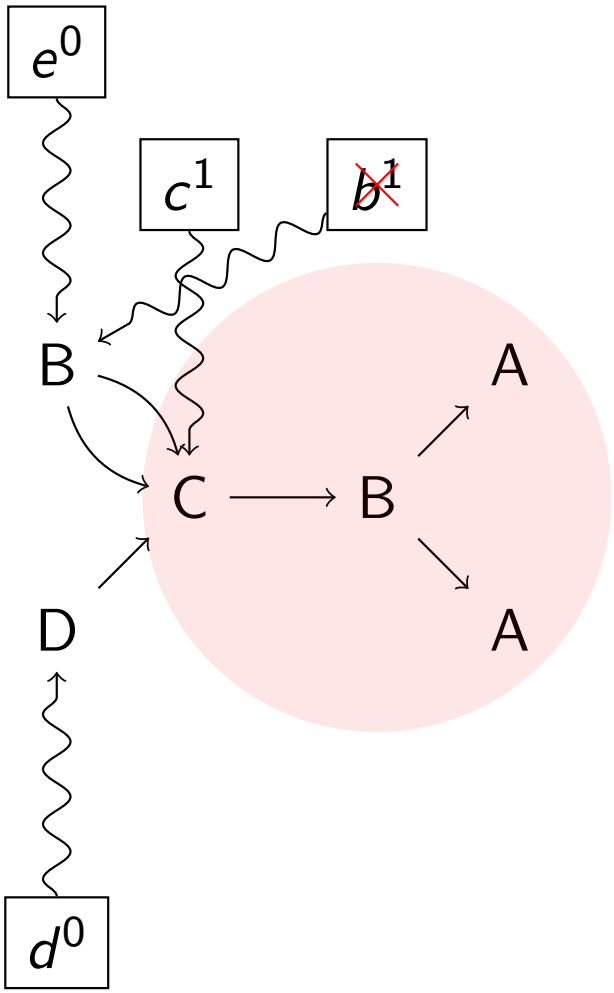
- The subheap of tier **1** never grows.
- Only tier **1** variables control `while` and recursive functions.
- The number of tier **1** configurations is bounded by $|\mathcal{C}|^{2\times n_1}$.
- Hence a bound on the stack and heap.

Idea of the proof



- The subheap of tier **1** never grows.
- Only tier **1** variables control `while` and recursive functions.
- The number of tier **1** configurations is bounded by $|\mathcal{C}|^{2\times n_1}$.
- Hence a bound on the stack and heap.

Idea of the proof



- The subheap of tier **1** never grows.
- Only tier **1** variables control `while` and recursive functions.
- The number of tier **1** configurations is bounded by $|\mathcal{C}|^{2\times n_1}$.
- Hence a bound on the stack and heap.

Conclusion

Result

A static analysis for resource consumption dealing with:

- several languages (imperative, fork, multi-thread, OO, ...)
- several classes (FPtime, FPspace,...)
- both extensional and intensional (heap, stack) properties

Drawbacks and Open questions

- Not intentionnally complete: improve expressiveness by program transformation
- Capture Thread creation (work in progress)
- Do the implementation
- Extend the characterizations (PP, BPP, ...)