DEADLOCK ANALYSIS OF CONCURRENT PROGRAMS

Cosimo Laneve

Department of Computer Science and Engineering
Università di Bologna

joint work with Elena Giachino

VERIFYING PROPERTIES OF PROGRAMS

1. define an abstract model of a program and provide an algorithm for checking the property

2. define a sound technique for associating an abstract model to the program

   - e.g. inference system, source-to-source transformation

3. 1 and 2 may introduce overapproximations (the verifier rejects safe codes)

VERIFYING DEADLOCK FREEDOM

a dependency pair between entities (names) is written (x, y)

- a *circular dependency* is a set of pairs such as $\{(x, y), (y, z), (z, x)\}$
- circular dependencies represent *deadlocks*

PLAN: developing an abstract model and an algorithm for the analysis of deadlocks in object-oriented languages

1. illustrate the abstract models of sample Java programs
2. define the abstract model -- *the language of lams*
3. specify a decision algorithm for detecting circular dependencies in linear lams
4. address the nonlinear case

A PARALLEL FACTORIAL

```
synchronized int fact(final int n) {
     if (n==0) return 1 ;
     else {
         final Maths x = new Maths() ;
         Thread t = new Thread(new Runnable() {
             public void run() { x.result = x.fact(n-1) ;
             }   }) ;
             t.start() ;
             t.join() ;
             return (n * x.result) ;
         }
    }

// exception handlers have been omitted
```

PARALLEL FACTORIAL AND LAMS

```
int fact(final int n) {
    if (n==0) return 1 ;
    else {
        final Maths x = new Maths() ;
        Thread t = new Thread(new Runnable() {
            public void run() { x.result = x.fact(n-1) ;
            }   }) ;
            t.start();
            t.join() ;
            return (n * x.result) ;
        }
    }
```

– assume that every function is synchronous

PARALLEL FACTORIAL AND LAMS

```
int fact(final int n) {
    if (n==0) return 1 ;
    else {
        final Maths x = new Maths() ;
                                    x.result = x.fact(n-1) ;
        return (n * x.result) ;
    }
}
```

- assume that every function is synchronous
- remove thread management

PARALLEL FACTORIAL AND LAMS

```
fact(final int n) {

    Maths x = new Maths() ;

                x.result = x.fact(n-1) ;

        return (n * x.result) ;
    }
}
```

- assume that every function is synchronous
- remove thread management
- remove (primitive) types and conditional and collect branches

PARALLEL FACTORIAL AND LAMS

```
fact( this ) {

    Maths x = new Maths() ;

                x.result = fact( x ) ;

        return (n * x.result) ;
    }
}
```

- assume that every function is synchronous
- remove thread management
- remove (primitive) types and conditional and collect branches
- highlight the owner of the method in the arguments (and remove integers)

PARALLEL FACTORIAL AND LAMS

```
fact( this ) {

    Maths x = new Maths() ;

                x.result = fact( x ) ;

        return (n * x.result) ;
    }
}
```

- assume that every function is synchronous
- remove thread management
- remove (primitive) types and conditional and collect branches
- highlight the owner of the method in the arguments (and remove integers)
- derive the lam function $\texttt{fact}(this) = (this, x)\&\texttt{fact}(x)$

ANOTHER PARALLEL FACTORIAL

```
synchronized int fact(final Maths x, final int n) {
      if (n==0) return 1 ;
      else {
          final Maths w = this ;
          Thread t = new Thread(new Runnable() {
              public void run() { x.result = x.fact(w, n-1) ;
              }   }) ;
          t.start() ;
          t.join() ;
          return (n * x.result) ;
      }
  }
```

- derive the lam function $\texttt{fact}(this, x) = (this, x)\&\texttt{fact}(x, this)$

LAMS

LAMS

- there are *names* ranged over by *x*, *y*, *z*, ...
- there are function names ranged over by $\mathtt{f}$, $\mathtt{g}$, $\mathtt{h}$, ...

a program is a tuple $\left(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x}_\ell) = \mathtt{L}_\ell, \mathtt{L}\right)$

- where $\mathtt{f}_i(\widetilde{x}_i) = \mathtt{L}_i$ are *function definitions* and $\mathtt{L}$ is the main lam
- *lams* $\mathtt{L}$ and $\mathtt{L}_i$ are defined by

$$\mathtt{L} \quad ::= \quad \mathtt{0} \;\mid\; (x, y) \;\mid\; \mathtt{f}(\widetilde{x}) \;\mid\; \mathtt{L}\&\mathtt{L} \;\mid\; \mathtt{L} + \mathtt{L}$$

such that (i) all function names occurring in $\mathtt{L}_i$ and $\mathtt{L}$ are defined and (ii) the arities of function invocations match function definitions

LAMS: EXAMPLES

1. the lam $(\mathtt{f}(x,y,z) = (x,y)\&\mathtt{f}(y,z,x),\ \mathtt{f}(u,v,w))$ evaluates

$$
\begin{aligned}
\mathtt{f}(u,v,w) &\longrightarrow (u,v)\&\mathtt{f}(v,w,u) \\
&\longrightarrow (u,v)\&(v,w)\&\mathtt{f}(w,u,v) \\
&\longrightarrow (u,v)\&(v,w)\&(w,u)\&\mathtt{f}(u,v,w) \qquad (1) \\
&\longrightarrow (u,v)\&(v,w)\&(w,u)\&\mathtt{f}(v,w,u) \\
&\longrightarrow (u,v)\&(v,w)\&(w,u)\&\mathtt{f}(w,u,v) \\
&\longrightarrow (1)
\end{aligned}
$$

this lam defines the relation $(u,v)\&(v,w)\&(w,u)$

- interpret & as "`,`" and omit "{" and "}"

which has a circular dependency

LAMS: EXAMPLES

2. lams may define several relations:
   the lam $(\mathbf{g}(x,y,z) = (x,y) + \mathbf{g}(y,z,x),\ \mathbf{g}(u,v,w))$ evaluates

$$
\begin{aligned}
\mathbf{g}(u,v,w) &\longrightarrow (u,v) + \mathbf{g}(v,w,u) \\
&\longrightarrow (u,v) + (v,w) + \mathbf{g}(w,u,v) \\
&\longrightarrow (u,v) + (v,w) + (w,u) + \mathbf{g}(u,v,w) \qquad (1) \\
&\longrightarrow (u,v) + (v,w) + (w,u) + \mathbf{g}(v,w,u) \\
&\longrightarrow (u,v) + (v,w) + (w,u) + \mathbf{g}(w,u,v) \\
&\longrightarrow (1)
\end{aligned}
$$

this lam defines the relations (u,v) and (v,w) and (w,u)

- there is no circular dependency

LAMS: EXAMPLES

3. lams may define relations that are not finite (**use name creation**):

   the lam $(\texttt{fact}(x) = (x, y)\&\texttt{fact}(y),\ \texttt{fact}(this))$ evaluates

$$\begin{aligned}
\texttt{fact}(this) &\longrightarrow (this, u)\&\texttt{fact}(u) \\
&\longrightarrow (this, u)\&(u, v)\&\texttt{fact}(v) \\
&\longrightarrow (this, u)\&(u, v)\&(v, w)\&\texttt{fact}(w) \\
&\longrightarrow \cdots
\end{aligned}$$

free names in method definitions are instantiated by run-time fresh names

this lam defines the relation $(this, u)\&(u, v)\&(v, w)\&\cdots$

- is it possible to predict that no circular dependency will be manifested?

LAMS: EXAMPLES

4. lams may define infinitely many different relations:
   the lam $\big(\mathtt{l}(x,y) = (x,y) + (x,y)\&\mathtt{l}(y,z),\ \mathtt{l}(u,v)\big)$ evaluates

$$\begin{aligned}\mathtt{l}(u,v) \quad &\longrightarrow \quad (u,v) + (u,v)\&\mathtt{l}(v,w) \\ &\longrightarrow \quad (u,v) + (u,v)\&(v,w) + (u,v)\&(v,w)\&\mathtt{l}(w,w') \\ &\longrightarrow \quad \ldots\end{aligned}$$

this lam defines the relations (u,v) and $(u,v)\&(v,w)$ and $(u,v)\&(v,w)\&(w,w')$ and …

- is it possible to predict that no circular dependency will be manifested in any relation?

LAMS: EXAMPLES

5. lams may be complex due to mutual recursion:

the **flh**-program

$$
\begin{array}{lcl}
(\quad \mathtt{f}(x,y,z,u) & = & (x,z)\&\mathtt{l}(u,y,z)\ , \\
\quad \mathtt{l}(x,y,z) & = & (x,y)\&\mathtt{f}(y,z,x,u)\ , \\
\quad \mathtt{h}(x,y,z,u) & = & (z,x)\&\mathtt{h}(x,y,z,u)\&\mathtt{f}(x,y,z,u)\ , \\
\quad \mathtt{h}(x_1,x_2,x_3,x_4) & &)
\end{array}
$$

[in this talk we keep programs as simple as possible]

MUTATIONS AND FLASHBACKS

THE PROBLEM

is it decidable whether the computations of a lam program will ever produce a circular dependency?

A SIMPLE ANSWER

simple cases:

i) functions are *linear*
(mutual) recursions have at most one recursive invocation -- such as **`fact`**

ii) function invocations do not have duplicate arguments and function definitions do not create new names

solution:

the function definition may be represented by a permutation; then evaluate up-to the *order of the permutation*

example: $(\mathtt{f}(x,y,z) = (x,y) \,\&\, \mathtt{f}(y,z,x),\ \mathtt{f}(u,v,w))$

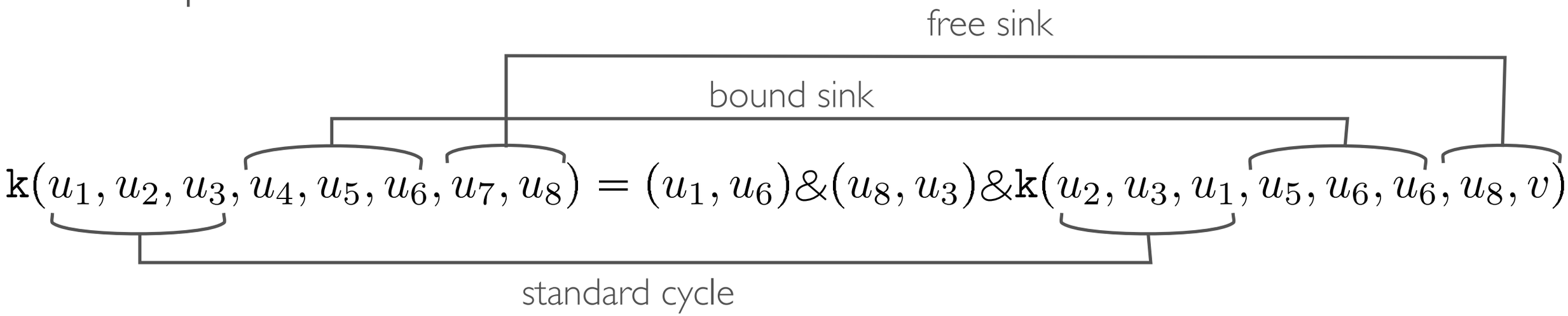
- the permutation has order 3
- the different invocations of **`f`** are collected in 3 steps

LINEAR LAMS AND MUTATIONS

drop the constraint

ii) function invocations do not have duplicate arguments and function definitions do not create new names

example:



k cannot be represented by a permutation

it is represented by a mutation

LAM SEMANTICS: THE PO $\mathbb{V}$

- let $\mathbb{V}$ be a *partial order* on names. Write
  - $x \in \mathbb{V}$ if there is a pair in $\mathbb{V}$ with the name x
  - $\mathbb{V} \oplus x < z$, with $x \in \mathbb{V}$ and $z \notin \mathbb{V}$, for the least partial order containing $\mathbb{V}$ and such that $x \leq y$ in $\mathbb{V}$ implies $y \leq z$
    (z is greater than every name in $\mathbb{V}$ greater than x)

EXAMPLES: - $\{(x,x)\} \oplus x<z \;=\; \{(x,x),(x,z),(z,z)\}$

- let $\mathbb{V} = \{(x,y),(x',y')\}$ (reflexive pairs are missing) then

$$\mathbb{V} \oplus y<z \;=\; \{(x,y),(x',y'),(y,z)\} \;=\; \mathbb{V} \oplus x<z$$

- generalise the notion to $\mathbb{V} \oplus \widetilde{x}<\widetilde{z}$, with $\widetilde{z} \notin \mathbb{V}$

LAM SEMANTICS

- lam contexts

$$\mathfrak{L}[\,] \quad ::= \quad [\,] \quad | \quad \mathtt{L}\&\mathfrak{L}[\,] \quad | \quad \mathtt{L} + \mathfrak{L}[\,]$$

the lam semantics is the least relation $\langle \mathbb{V},\ \mathtt{L}\rangle \longrightarrow \langle \mathbb{V}',\ \mathtt{L}'\rangle$ that satisfies the rule

$$\text{(RED)}\quad \frac{\mathtt{f}(\widetilde{x}) = \mathtt{L} \qquad var(\mathtt{L})\backslash\widetilde{x} = \widetilde{z} \qquad \widetilde{w} \text{ are fresh} \qquad \mathtt{L}[^{\widetilde{w}}/_{\widetilde{z}}][^{\widetilde{u}}/_{\widetilde{x}}] = \mathtt{L}'}{\langle \mathbb{V},\ \mathfrak{L}[\mathtt{f}(\widetilde{u})]\rangle \longrightarrow \langle \mathbb{V} \oplus \widetilde{u}{<}\widetilde{w},\ \mathfrak{L}[\mathtt{L}']\rangle}$$

with initial state $\langle \{(x,x) \mid x \in var(\mathtt{L})\},\ \mathtt{L}\rangle$ where $\mathtt{L}$ is the main term

ORDER OF MUTATIONS

$$\mathtt{k}(u_1,u_2,u_3,u_4,u_5,u_6,u_7,u_8) \longrightarrow (u_1,u_6)\&(u_8,u_3)\&\mathtt{k}(u_2,u_3,u_1,u_5,u_6,u_6,u_8,v)$$

$$\longrightarrow (u_1,u_6)\&(u_8,u_3)\&(u_2,u_6)\&(v,u_1)\& \&\mathtt{k}(u_3,u_1,u_2,u_6,u_6,u_6,v,w)$$

$$\longrightarrow (u_1,u_6)\&(u_8,u_3)\&(u_2,u_6)\&(v,u_1)\&(u_3,u_6)\&(w,u_2) \&\mathtt{k}(u_1,u_2,u_3,u_6,u_6,u_6,w,x)$$

$$\longrightarrow (u_1,u_6)\&(u_8,u_3)\&(u_2,u_6)\&(v,u_1)\&(u_3,u_6)\&(w,u_2) \&(u_1,u_6)\&(x,u_3)\&\mathtt{k}(u_2,u_3,u_1,u_6,u_6,u_6,x,y)$$

$$\longrightarrow (u_1,u_6)\&(u_8,u_3)\&(u_2,u_6)\&(v,u_1)\&(u_3,u_6)\&(w,u_2) \&(u_1,u_6)\&(x,u_3)\&(u_2,u_6)\&(y,u_1) \&\mathtt{k}(u_3,u_1,u_2,u_6,u_6,u_6,y,z)$$

$$\longrightarrow \ldots$$

order of the mutation = $\mathtt{max}(\ell+\ell_1, \ell_2)$

ℓ = order of standard cycles, ℓ_1 = length of max bound sink **-1**, ℓ_2 = length of max free sink

FREE SINKS AND FLASHBACKS

$\mathtt{k}(u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8) \longrightarrow (u_1, u_6)\&(u_8, u_3)\&\mathtt{k}(u_2, u_3, u_1, u_5, u_6, u_6, u_8, v)$

$\longrightarrow (u_1, u_6)\&(u_8, u_3)\&(u_2, u_6)\&(v, u_1)\&$
$\&\mathtt{k}(u_3, u_1, u_2, u_6, u_6, u_6, v, w)$

$\longrightarrow (u_1, u_6)\&(u_8, u_3)\&(u_2, u_6)\&(v, u_1)\&(u_3, u_6)\&(w, u_2)$
$\&\mathtt{k}(u_1, u_2, u_3, u_6, u_6, u_6, w, x)$

$\longrightarrow (u_1, u_6)\&(u_8, u_3)\&(u_2, u_6)\&(v, u_1)\&(u_3, u_6)\&(w, u_2)$
$\&(u_1, u_6)\&(x, u_3)\&\mathtt{k}(u_2, u_3, u_1, u_6, u_6, u_6, x, y)$

$\longrightarrow (u_1, u_6)\&(u_8, u_3)\&(u_2, u_6)\&(v, u_1)\&(u_3, u_6)\&(w, u_2)$
$\&(u_1, u_6)\&(x, u_3)\&(u_2, u_6)\&(y, u_1)$
$\&\mathtt{k}(u_3, u_1, u_2, u_6, u_6, u_6, y, z)$

$\longrightarrow \ldots$

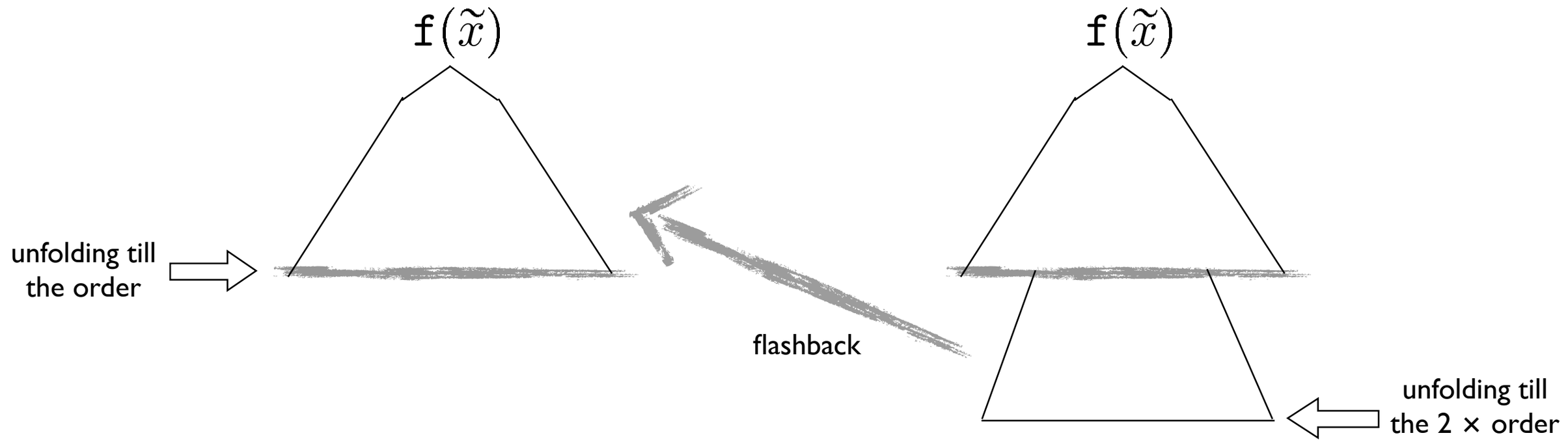
flashback
an injective renaming
such that $v, w < y, z$

point of creation

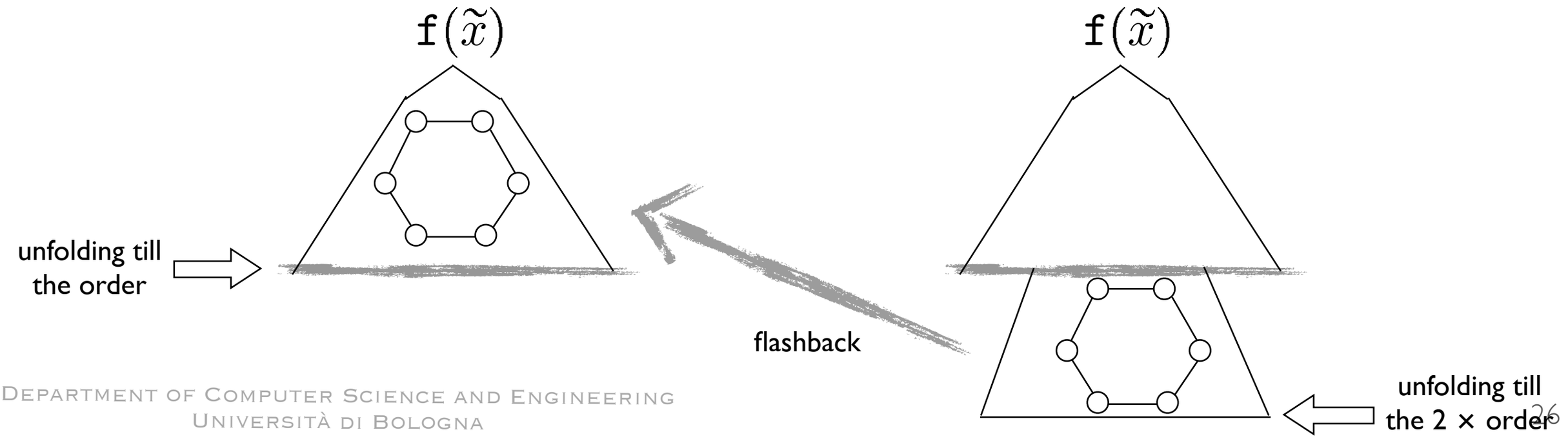
theorem: every linear lam function has an order $\mathscr{O}$ such that the recursive invocation after $\mathscr{O}$-unfoldings may be mapped back to a previous invocation by a flashback

ORDERS AND CIRCULARITIES

flashbacks allow one to map back unfoldings of linear functions

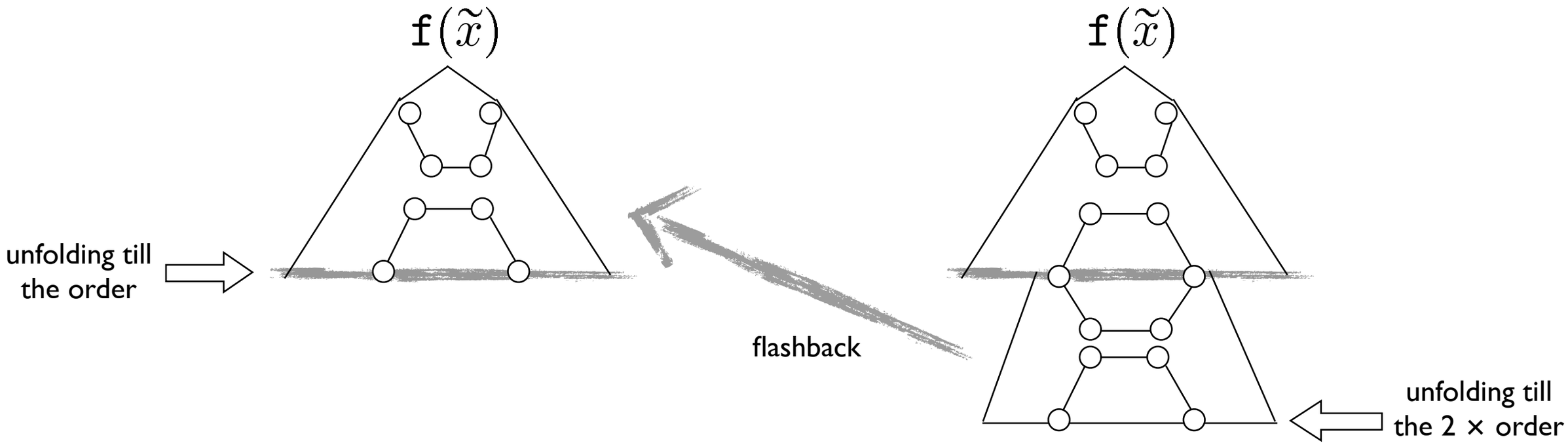


this is a good property for circularities



CROSSOVER CIRCULARITIES

the problematic cases are the circularities across the order



these circularities require 2 × order to be catched (saturated states)

THE DECISION ALGORITHM

STEP 1: *find recursive histories*.

- create a graph where nodes are function names and, for every invocation of `g` in the body of `f`, there is an edge from `f` to `g`
- use depth first search to associate to every node its recursive histories (the paths starting and ending at that node, if any)
- the Iam program is linear if every node has at most one associated recursive history

STEP 2: *computation of the orders*.

- given the recursive history associated to a function, we compute the corresponding mutation and its order

STEP 3: *evaluation process*. The main Iam is unfolded till the the saturated state

- every function invocation `f`(x) in the main Iam is evaluated up-to twice the order of the corresponding mutation
- the function invocation of `f` in the saturated state is erased and the process is repeated on every other function invocation (which, therefore, does not belong to the recursive history of `f`), till no function invocation is present in the state

STEP 4: *detection of circularities*.

- every `T` (a & of dependency pairs) may be represented as a graph where nodes are names and edges correspond to dependency pairs
- to detect whether `T` contains a circular dependency, we run Tarjan algorithm for connected components of graphs and we stop the algorithm when a circularity is found

NONLINEAR LAMS

the decision algorithm for detecting locks may be applied to nonlinear programs *after having transformed them to linear ones*

- the current *source-to-source transformation* introduces inaccuracies, e.g. pairs that are not present in the nonlinear program
- perhaps the transformation may be improved but the inaccuracies cannot be completely removed

EXAMPLE: the fibonacci-like lam function

$$\texttt{fib}(z) = (z, x)\&(z, y)\&\texttt{fib}(x)\&\texttt{fib}(y)$$

is transformed into the linear one

$$\texttt{fib}^{aux}(z, z') = (z, x)\&(z, y)\&(z', x)\&(z', y)\&\texttt{fib}^{aux}(x, y)$$

by introducing fake dependencies

EXERCISES

compute the order and verify the circularity freedom of the following functions (the main lam is the function itself):

1. (`g(x0, x1, x2, x3, x4, x5, x6) = (x3, x1)&(x0, x8)&(x8, x7)`
`& g(x2, x0, x1, x5, x6, x7, x8) ,`
`g(x0, x1, x2, x3, x4, x5, x6)`)

2. $$\left(\begin{array}{lcl} \mathtt{f}(x,y,z,u) & = & (x,z)\&\mathtt{l}(u,y,z)\ , \\ \mathtt{l}(x,y,z) & = & (x,y)\&\mathtt{f}(y,z,x,u)\ , \\ \mathtt{h}(x,y,z,u) & = & (z,x)\&\mathtt{h}(x,y,z,u)\&\mathtt{f}(x,y,z,u)\ , \\ \mathtt{h}(x_1,x_2,x_3,x_4) & & \end{array}\right)$$

3. (coffee competition) write your favourite concurrent program, extract the lam and verify whether it is deadlocked or not

coreABS

details at **cs.unibo.it/~laneve/papers/SoSyM.pdf**

coreABS

the theory is being applied to a concurrent object-oriented programming language **coreABS**:

1. an association programs/lams is defined by means of an inference type system that derives lams

2. the subject reduction theorem demonstrates that the lams of the initial program manifest more dependency pairs than the reduced one

coreABS

$P ::= \overline{C}\ \{\overline{T\ x\ \texttt{;}}\ s\}$	program
$T ::= \texttt{nat} \mid \texttt{bool} \mid \texttt{Fut<}T\texttt{>}$	type
$C ::= \texttt{class C}(\overline{T\ x})\ \{\overline{T\ x\ \texttt{;}}\ \overline{M}\}$	class
$M ::= T\ \texttt{m}(\overline{T\ x})\{\overline{T\ x\ \texttt{;}}\ s\}$	method definition
$s ::= \texttt{skip} \mid x = z \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\} \mid \texttt{return}\ e \mid s\ \texttt{;}\ s$	statement
$z ::= e \mid e\texttt{!m}(\overline{e}) \mid \texttt{new C}\ (\overline{e}) \mid \texttt{new local C}\ (\overline{e}) \mid x\texttt{.get}$	expression with side effects
$e ::= v \mid x \mid \texttt{this} \mid \textit{arithmetic-and-bool-exp}$	expression
$v ::= \texttt{null} \mid \textit{primitive values}$	value

remarks:

- assignments (alias analysis)
- asynchronous method invocations
- futures
- **`new`** vs **`new local`**
- explicit synchronizations

coreABS EXAMPLES

```
Int fact_g(Int n){
    Fut<Int> x ;
    Int m = 0;
    if (n==0) { m = 1; }
    else { x = this!fact_g(n-1); m = x.get; m = n*m; }
    return m;
}

Int fact_nc(Int n){
    Fut<Int> x ;
    Int m = 0;
    if (n==0) { m = 1 ; }
    else { Math z = new Math(); x = z!fact_nc(n-1); m = x.get; m = n*m; }
    return m;
}
```

CONSTRAINTS

- method invocations are synchronised in the same body
- fields of future types are read-only
- recursive object structures are problematic in the inference algorithm (unproblematic in this talk, where we discuss the type checking)

BEHAVIOURAL TYPES

$$\mathbb{r} ::= \_ \mid X \mid [cog{:}c, \overline{x}{:}\overline{\mathbb{r}}] \mid c \rightsquigarrow \mathbb{r} \qquad \text{future record}$$

$$\mathbb{c} ::= \mathtt{0} \mid \mathtt{0}.(c, c') \mid \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \mid \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c, c') \mid \mathbb{c} + \mathbb{c} \mid \mathbb{c}\&\mathbb{c} \qquad \text{contract}$$

$$\mathbb{x} ::= \_ \mid X \mid [cog{:}c, \overline{x}{:}\overline{\mathbb{x}}] \mid c \rightsquigarrow \mathbb{r} \mid f \qquad \text{extended future record}$$

$$\mathbb{z} ::= (\mathbb{r}, \mathbb{c}) \mid (\mathbb{r}, \mathbb{c})^{\checkmark} \qquad \text{future reference values}$$

we use *environments* Γ mapping

- variables x to extended future records
- futures references f to future reference values
- **`C.m`** to $[cog : c, \overline{x{:}\mathbb{r}}](\overline{\mathbb{s}})\{\mathbb{c}\}\mathbb{r}'$

TYPING JUDGMENTS

- $\Gamma \vdash_c e : \mathbb{x}$ for pure expressions
- $\Gamma \vdash_c f : \mathbb{z}$ for future references
- $\Gamma \vdash_c z : \mathbb{x}, \mathbb{c} \rhd \Gamma'$ for expressions with side effects
- $\Gamma \vdash_c s : \mathbb{c} \rhd \Gamma'$ for statements

we write $\Gamma(\mathtt{C.m}) =_\alpha [cog : c, \overline{x{:}\mathbb{r}}](\overline{\mathbb{s}})\{\mathbb{c}\}\mathbb{r}'$ when they are alpha-equivalent

[*cog*: c, *x*: *X*]([*cog*: *c'*, *x*: *Y*]){ **C!m** [*cog*: *c''*, *x*: *Y*]([*cog*: c, *x*: *X*])→[*cog*: *c'''*, *x*: *X*].(c, c'') } [*cog*: *c''*, *x*: *Y*]

$=\alpha$

[*cog*: *d*, *x*: *Z*]([*cog*: *d'*, *x*: *W*]){ **C!m** [*cog*: *d''*, *x*: *W*]([*cog*: *d*, *x*: *Z*])→[*cog*: *d'''*, *x*: *Z*].(d, d'') } [*cog*: *d''*, *x*: *W*]

TYPING JUDGMENTS (CONT.)

the judgment $\Gamma \vdash \mathtt{C.m} : [cog{:}c', \overline{x{:}\mathbb{r}}](\overline{\mathbb{s}})\{\mathbb{c}\}\mathbb{r}'$ means that

$[cog : c, \overline{x{:}\mathbb{r}}](\overline{\mathbb{s}})\{\mathbb{c}\}\mathbb{r}'$ is an instance of $\Gamma(\mathtt{C.m})$

example:

[*cog*: *d*, *x*: 𝕣]([*cog*: *d'*, *x*: 𝕣′]){ **C ! m** [*cog*: *d''*, *x*: 𝕣′]([*cog*: *d*, *x*: 𝕣])→[*cog*: *d'''*, *x*: 𝕣].(d, d'') } [*cog*: *d''*, *x*: 𝕣′]

is an instance of

[*cog*: *c*, *x*: *X*]([*cog*: *c'*, *x*: *Y*]){ **C ! m** [*cog*: *c''*, *x*: *Y*]([*cog*: *c*, *x*: *X*])→[*cog*: *c'''*, *x*: *X*].(c, c'') } [*cog*: *c''*, *x*: *Y*]

let also $unsync(\Gamma) \stackrel{def}{=} \&\{\mathbb{c} \mid \text{there are } f, \mathbb{r} : \ \Gamma(f) = (\mathbb{r}, \mathbb{c})\}$

TYPING RULES FOR EXPRESSIONS

expressions and addresses

(T-Var)
$$\frac{\Gamma(x) = \mathbb{x}}{\Gamma \vdash_c x : \mathbb{x}}$$

(T-Fut)
$$\frac{\Gamma(f) = \mathbb{z}}{\Gamma \vdash_c f : \mathbb{z}}$$

(T-Field)
$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma(\texttt{this}.x) = \mathbb{x}}{\Gamma \vdash_c x : \mathbb{x}}$$

(T-Value)
$$\frac{\Gamma \vdash_c e : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c})^{[\checkmark]}}{\Gamma \vdash_c e : \mathbb{r}}$$

(T-Val)
$$\frac{e \quad \textit{primitive value or arithmetic-and-bool-exp}}{\Gamma \vdash_c e : \_}$$

(T-Pure)
$$\frac{\Gamma \vdash_c e : \mathbb{r}}{\Gamma \vdash_c e : \mathbb{r}, \mathtt{0} \rhd \Gamma}$$

expressions with side effects

(T-Get)
$$\frac{\Gamma \vdash_c x : f \qquad \Gamma \vdash_c f : (c' \rightsquigarrow \mathbb{r}, \mathbb{c}) \qquad \Gamma' = \Gamma[f \mapsto (c' \rightsquigarrow \mathbb{r}, \mathbb{c})^{\checkmark}]}{\Gamma \vdash_c x.\texttt{get} : X, \mathbb{c}.(c, c') \& \mathit{unsync}(\Gamma') \rhd \Gamma'}$$

(T-Get-tick)
$$\frac{\Gamma \vdash_c x : f \qquad \Gamma \vdash_c f : (c' \rightsquigarrow \mathbb{r}, \mathbb{c})^{\checkmark}}{\Gamma \vdash_c x.\texttt{get} : \mathbb{r}, \mathtt{0} \rhd \Gamma}$$

(T-New)
$$\frac{\begin{array}{c}\Gamma \vdash_c \overline{e} : \overline{\mathbb{r}} \\ \mathit{fields}(\mathtt{C}) = \overline{T\ x} \qquad c' \text{ fresh}\end{array}}{\Gamma \vdash_c \texttt{new C}(\overline{e}) : [\mathit{cog}{:}c', \overline{x{:}\mathbb{r}}], \mathtt{0} \rhd \Gamma}$$

(T-NewLocal)
$$\frac{\begin{array}{c}\Gamma \vdash_c \overline{e} : \overline{\mathbb{r}} \\ \mathit{fields}(\mathtt{C}) = \overline{T\ x}\end{array}}{\Gamma \vdash_c \texttt{new local C}(\overline{e}) : [\mathit{cog}{:}c, \overline{x{:}\mathbb{r}}], \mathtt{0} \rhd \Gamma}$$

(T-AInvk)
$$\frac{\begin{array}{c}\Gamma \vdash_c e : [\mathit{cog}{:}c', \overline{x{:}\mathbb{r}}] \qquad \Gamma \vdash_c \overline{e} : \overline{\mathbb{s}} \qquad \Gamma \vdash \mathtt{C.m} : [\mathit{cog}{:}c', \overline{x{:}\mathbb{r}}](\overline{\mathbb{s}})\{\mathbb{c}\}\mathbb{r}' \\ \mathit{class}(\mathit{types}(e)) = \mathtt{C} \qquad \mathit{fields}(\mathtt{C}) = \overline{T\ x} \qquad f \text{ fresh}\end{array}}{\Gamma \vdash_c e!\mathtt{m}(\overline{e}) : f, \mathtt{0} \rhd \Gamma[f \mapsto (c' \rightsquigarrow \mathbb{r}', \mathtt{C}!\mathtt{m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}')]}$$

TYPING RULES FOR STATEMENTS

statements

(T-Field-Record)

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma(\texttt{this}.x) = \mathbb{r} \qquad \Gamma \vdash_c z : \mathbb{r}', \mathbb{c} \rhd \Gamma'}{\Gamma \vdash_c x = z : \mathbb{c} \rhd \Gamma'}$$

(T-Var-Record)

$$\frac{\Gamma \vdash_c z : \mathbb{r}, \mathbb{c} \rhd \Gamma'}{\Gamma \vdash_c x = z : \mathbb{c} \rhd \Gamma'[x \mapsto \mathbb{r}]}$$

(T-Var-Future)

$$\frac{\Gamma \vdash_c z : f, \mathbb{c} \rhd \Gamma'}{\Gamma \vdash_c x = z : \mathbb{c} \rhd \Gamma'[x \mapsto f]}$$

(T-If)

$$\frac{\Gamma \vdash_c e : \texttt{Bool} \qquad \Gamma \vdash_c s_1 : \mathbb{c}_1 \rhd \Gamma_1 \qquad \Gamma \vdash_c s_2 : \mathbb{c}_2 \rhd \Gamma_2 \qquad (\bigwedge_{x \in \mathrm{dom}(\Gamma)} \Gamma_1(x) = \Gamma_2(x)) \wedge (\bigwedge_{x \in \texttt{Fut}(\Gamma)} \Gamma_1(\Gamma_1(x)) = \Gamma_2(\Gamma_2(x))) \qquad \Gamma' = \Gamma_1 + \Gamma_2|_{\{f \mid f \notin \Gamma_2(\texttt{Fut}(\Gamma))\}}}{\Gamma \vdash_c \texttt{if } e \,\{\, s_1 \,\} \texttt{ else } \{\, s_2 \,\} : \mathbb{c}_1 + \mathbb{c}_2 \rhd \Gamma'}$$

(T-Seq)

$$\frac{\Gamma \vdash_c s_1 : \mathbb{c}_1 \rhd \Gamma_1 \qquad \Gamma_1 \vdash_c s_2 : \mathbb{c}_2 \rhd \Gamma_2}{\Gamma \vdash_c s_1; s_2 : \mathbb{c}_1 + \mathbb{c}_2 \rhd \Gamma_2}$$

(T-Return)

$$\frac{\Gamma \vdash_c e : \mathbb{r} \qquad \Gamma(\texttt{destiny}) = \mathbb{r}}{\Gamma \vdash_c \texttt{return } e : \texttt{0} \rhd \Gamma}$$

TYPING RULES FOR METHODS

(T-Method)

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{T_f\ x} \qquad c\ \mathit{fresh} \qquad \Gamma(\mathtt{C.m}) =_\alpha [\mathit{cog} : c, \overline{x{:}\mathbb{r}}](\mathbb{s})\{\mathbb{c}\}\mathbb{r}' \qquad \Gamma + \mathtt{this} : [\mathit{cog}{:}c,\ \overline{x{:}\mathbb{r}}] + \mathtt{destiny} : \mathbb{r}' \vdash_c s : \mathbb{c} \rhd \Gamma'}{\mathtt{C}, \Gamma \vdash T\ \mathtt{m}\ (\overline{T\ y})\{\overline{T_l\ w};\ s\}\ :\ [\mathit{cog} : c, \overline{x{:}\mathbb{r}}](\mathbb{s})\{\mathbb{c}\}\mathbb{r}'}$$

(the typing rules for classes and methods are omitted)

Theorem (Subject Reduction) *Let* $\Gamma \vdash cn : \mathbb{c}$ *and* $cn \rightarrow cn'$. *Then there exist* Γ', $\mathbb{c}'$ *such that*

- $\Gamma' \vdash cn' : \mathbb{c}'$ *and*
- *if* $\mathbb{c}'$ *has a circularity then also* $\mathbb{c}$ *has a circularity.*

DF4ABS

we have defined an inference system for **`coreABS`**:

1. **lams are extracted automatically**

2. lam analysis is performed with **three** different algorithms

for a prototype web interface, visit

`http://df4abs.nws.cs.unibo.it`

EXAMPLE

```
module Factorial ;
interface Math { Int fact_g(Int n); Int fact_nc(Int n);
 }
class Math implements Math {
 Int fact_g(Int n){
    Fut<Int> x ;
    Int m = 0;
    if (n==0) { m = 1; }
    else { x = this!fact_g(n-1); m = x.get; m = n*m; }
    return m ;
    }
 Int fact_nc(Int n){
    Fut<Int> x ;
    Int m = 0;
    if (n==0) { m = 1 ; }
    else { Math z = new Math(); x = z!fact_nc(n-1); m = x.get; m = n*m; }
    return m ;
    }
}
{ Math x = new Math(); Fut<Int> fut = x!fact_g(7); fut.get ;
}
```

```
### LOCK INFORMATION RESULTED BY THE ANALYSIS ###

Possible Deadlock in Main:    true
Current Version:              2
Analysis Duration:            34ms
```

EXAMPLE

```
module Factorial ;
interface Math { Int fact_g(Int n); Int fact_nc(Int n);
 }
class Math implements Math {
 Int fact_g(Int n){
    Fut<Int> x ;
    Int m = 0;
    if (n==0) { m = 1; }
    else { x = this!fact_g(n-1); m = x.get; m = n*m; }
    return m ;
    }
 Int fact_nc(Int n){
    Fut<Int> x ;
    Int m = 0;
    if (n==0) { m = 1 ; }
    else { Math z = new Math(); x = z!fact_nc(n-1); m = x.get; m = n*m; }
    return m ;
    }
}
{ Math x = new Math(); Fut<Int> fut = x!fact_nc(7); fut.get ;
}
```

```
### LOCK INFORMATION RESULTED BY THE ANALYSIS ###

Possible Deadlock in Main:    false
Current Version:              2
Analysis Duration:            32ms
```

EXERCISE

write the typing proof of the method

```
Int fact_nc(Int n){
    Fut<Int> x ;
    Int m = 0;
    if (n==0) { m = 1 ; }
    else { Math z = new Math(); x = z!fact_nc(n-1); m = x.get; m = n*m; }
    return m ;
    }
```

EPILOGUE

- the theory is also applied to **`pi-calculus`** (with Kobayashi):

  1. an association programs/lams is defined by means of an inference type system that derives lams

  2. we have a lam algorithm that covers linear and non-linear lams (circularity freedom of lams is decidable)

- we are applying the same technique for analysing resource deployments in cloud computing

- for related works and future research directions, see the papers at

`www.cs.unibo.it/~laneve`

QUESTIONS