ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

A HYBRID APPROACH TO AUTOMATIC VERIFICATION OF CONCURRENT SYSTEMS

COSIMO LANEVE
cosimo.laneve@unibo.it

OPTC 2023 – BERTINORO

CONTENTS

presenting a technique for automatic verification of concurrent systems that has been used for designing analysis tools and demonstrating their correctness

the technique consists of

* associating behavioural types to programs that record relevant infos for a given property

  implement a type inference system

* define an analyzer of behavioural types or use off-the-shelf analyzers

  not hard: resort to well-established theories

BIBLIO

- Cosimo Laneve, Luca Padovani: **An Algebraic Theory for Web Service Contracts**. IFM **2013**: 301-315; Formal Aspects Comput. 27(4): 613-640 (2015)

finite transition systems

- Elena Giachino, Naoki Kobayashi, Cosimo Laneve: **Deadlock Analysis of Unbounded Process Networks**. CONCUR **2014**: 63-77; Inf. Comput. 252: 48-70 (2017)

lams

- Abel Garcia, Cosimo Laneve, Michael Lienhardt: **Static analysis of cloud elasticity**. PPDP **2015**: 125-136; Sci. Comput. Program. 147: 27-53 (2017)

presburger arithmetics eq.

- Elena Giachino, Einar Broch Johnsen, Cosimo Laneve, Ka I Pun: **Time Complexity of Concurrent Programs — A Technique Based on Behavioural Types**. FACS **2015**: 199-216; J. Log. Algebraic Methods Program. 105: 1-27 (2019)

presburger arithmetics eq.

- Cosimo Laneve: **A Lightweight Deadlock Analysis for Programs with Threads and Reentrant Locks**. FM **2018**: 608-624; Sci. Comput. Program. 181: 64-81 (2019)

lams

- Cosimo Laneve, Claudio Sacerdoti Coen: **Analysis of smart contracts balances**. In Blockchain: Research and Applications, vol 2(3) (**2021**)

presburger arithmetics eq.

- Silvia Crafa, Cosimo Laneve: **Liquidity Analysis in Resource-Aware Programming**. FACS **2022**: 205-221

finite-length abstract computations

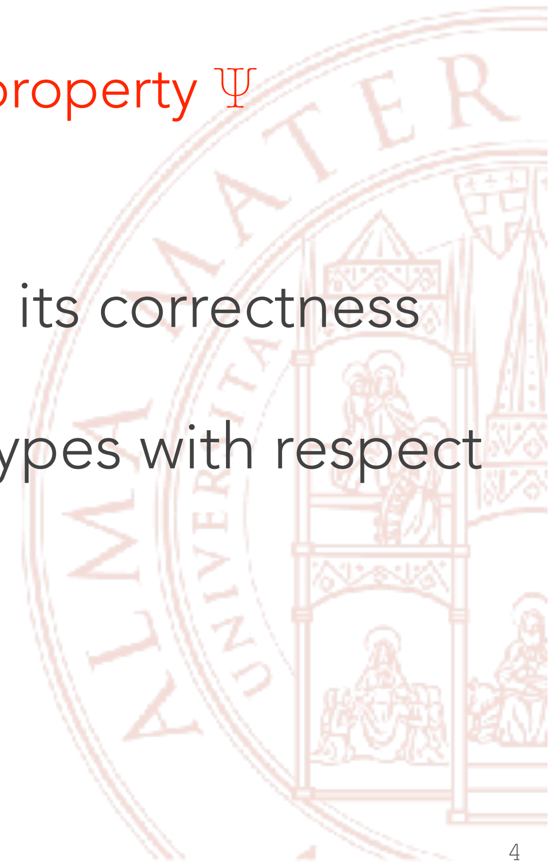
THE TECHNIQUE IN DETAIL

INPUT: a programming language with an operational semantics $\rightarrow$

demonstrate that a program $\mathtt{P}$ owns a property Ψ

FIRST HALF: the behavioural type inference and its correctness

SECOND HALF: the correctness of behavioural types with respect to the property Ψ



THE TECHNIQUE IN DETAIL — THE FIRST HALF

the behavioural type inference and its correctness

1. define **behavioural types** that are abstract specifications of programs detailed enough to verify Ψ

2. define a **relation**, say $\sqsupseteq$ and called *greater-than*, on behavioural types, e.g. $\mathbb{b} \sqsupseteq \mathbb{b}'$

3. define an **inference system** such that $\Gamma \vdash \mathtt{P} : \mathbb{b}$

   the difficulty of implementing $\Gamma \vdash \mathtt{P} : \mathbb{b}$ is the same of standard semantic analysis

   comments: (1) there are as many rules as the syntactic constructs in the language

   (2) additional rules are needed for the runtime configurations of the program, c.f. $\rightarrow$

4. THEOREM (**subject reduction**) if $\Gamma \vdash \mathtt{P} : \mathbb{b}$ and $\mathtt{P} \rightarrow \mathtt{P}'$ then there are Γ', $\mathbb{b}'$ such that $\Gamma' \vdash \mathtt{P}' : \mathbb{b}'$ and $\mathbb{b} \sqsupseteq \mathbb{b}'$

   in standard type systems $\mathbb{b} = \mathbb{b}'$

THE TECHNIQUE IN DETAIL — THE SECOND HALF

the correctness of behavioural types

5. define a property Ψ' over behavioural types and specify $\mathbb{b} \Vdash \Psi'$

   comments: (1) Ψ' is different from Ψ because behavioural types and programs have different models

   (2) the algorithm for $\mathbb{b} \Vdash \Psi'$ must terminate

   (2) usually $\mathbb{b}$ is either a finite model or one uses a terminating fixpoint technique

   (3) $\mathbb{b} \Vdash \Psi'$ requires no work if there is a well-established theory

6. demonstrate that, if $\mathbb{b} \Vdash \Psi'$ and $\mathbb{b} \sqsupseteq \mathbb{b}'$, then $\mathbb{b}' \Vdash \Psi'$

7. demonstrate that, if $\Gamma \vdash \mathtt{P} : \mathbb{b}$ and $\mathbb{b} \Vdash \Psi'$ then $\mathtt{P}$ owns the property Ψ

A CASE STUDY

- Cosimo Laneve, Luca Padovani: **An Algebraic Theory for Web Service Contracts**. IFM **2013**: 301-315; Formal Aspects Comput. 27(4): 613-640 (2015)

  finite transition systems

- Elena Giachino, Naoki Kobayashi, Cosimo Laneve: **Deadlock Analysis of Unbounded Process Networks**. CONCUR **2014**: 63-77; Inf. Comput. 252: 48-70 (2017)

  lams

- Abel Garcia, Cosimo Laneve, Michael Lienhardt: **Static analysis of cloud elasticity**. PPDP **2015**: 125-136; Sci. Comput. Program. 147: 27-53 (2017)

  presburger arithmetics eq.

- Elena Giachino, Einar Broch Johnsen, Cosimo Laneve, Ka I Pun: **Time Complexity of Concurrent Programs — A Technique Based on Behavioural Types**. FACS **2015**: 199-216; J. Log. Algebraic Methods Program. 105: 1-27 (2019)

  presburger arithmetics eq.

- Cosimo Laneve: **A Lightweight Deadlock Analysis for Programs with Threads and Reentrant Locks**. FM **2018**: 608-624; Sci. Comput. Program. 181: 64-81 (2019)

  lams

- Cosimo Laneve, Claudio Sacerdoti Coen: **Analysis of smart contracts balances**. In Blockchain: Research and Applications, vol 2(3) (**2021**)

  presburger arithmetics eq.

- Silvia Crafa, Cosimo Laneve: **Liquidity Analysis in Resource-Aware Programming**. FACS **2022**: 205-221

  finite-length abstract computations

THE PROGRAMMING LANGUAGE — miniJava

- programs in miniJava are $(\mathbb{D}, \mathtt{P})$
- $\mathbb{D}$ is a finite set of **method name definitions** $A(x_1, \ldots, x_n) = P_A$
- P is the **main process**
- the syntax of P and P_A is

$$P ::= 0 \mid (\nu\ x)\ P \mid (\nu\ P)\ P \mid \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P \mid A(e_1, \ldots, e_n) \mid \mathtt{sync}(x)\{\ P\ \}$$

$$e ::= x \mid v \mid e\ \mathit{op}\ e$$

in Java this is

```
Thread t = new Thread(){ P } ;
t.start() ;
```

in Java this is

```
synchronized(x){ . . . }
```

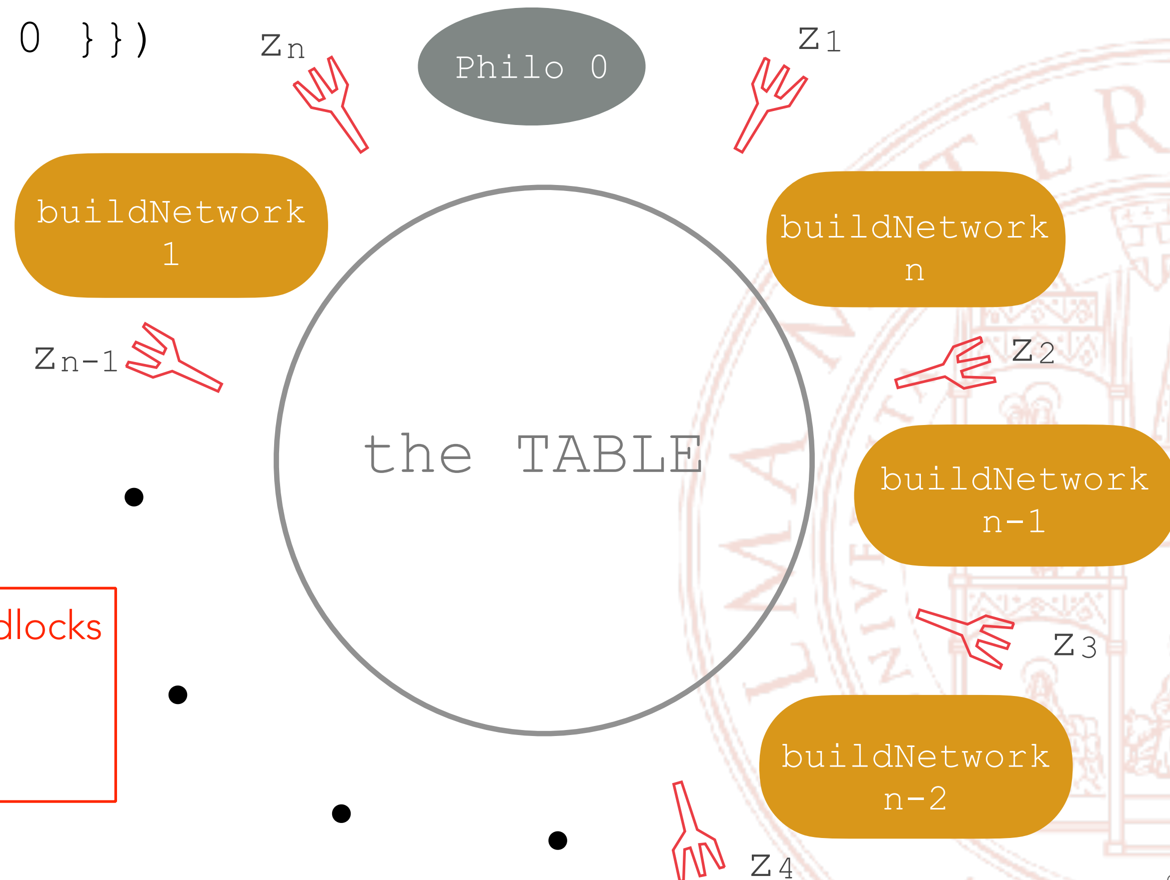
- you may write complex programs

THE DINING PHILOSOPHERS IN miniJava

- you may write complex programs in miniJava:

```
buildNetwork(x,y,n) = (ν z)(
   if (n==1) then sync(x){sync(y){ 0 }}
   else (ν sync(x){sync(z){ 0 }}) buildNetwork(z,y,n-1)
   )
```

```
(ν sync(zₙ){sync(z₁){ 0 }})
buildNetwork(z₁,zₙ,n)
```



- **symmetric strategy**: possible deadlocks
- at least one Philo with a **different strategy**: no deadlocks

THE OPERATIONAL SEMANTICS OF `miniJava`

- program $(\mathbb{D}, \mathtt{P})$
- the initial state is $\mathtt{P} \bullet \varepsilon$, generic states are $\mathtt{P}_1 \bullet \sigma_1 \mid \ldots \mid \mathtt{P}_n \bullet \sigma_n$
- the operational semantics:

sequence of locks that have been acquired by $\mathtt{P}_1$

parallel threads

(Zero)
$$0 \bullet \sigma \longrightarrow$$

(NewO)
$$\frac{z = \mathit{fresh}(\;)}{(\nu\ x)\, P \bullet \sigma \mid \mathbb{P} \longrightarrow P\{^z/_x\} \bullet \sigma \mid \mathbb{P}}$$

(NewT)
$$(\nu\ P)\ Q \bullet \sigma \longrightarrow P \bullet \varepsilon \mid Q \bullet \sigma$$

(IfT)
$$\frac{[\![e]\!] \neq 0}{\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \bullet \sigma \longrightarrow P \bullet \sigma}$$

(IfF)
$$\frac{[\![e]\!] = 0}{\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \bullet \sigma \longrightarrow Q \bullet \sigma}$$

(Call)
$$\frac{[\![\overline{e}]\!] = \overline{v} \quad A(\overline{z}) = P}{A(\overline{e}) \bullet \sigma \longrightarrow P\{^{\overline{v}}/_{\overline{z}}\} \bullet \sigma}$$

(Sync)
$$\frac{x \notin \mathbb{P}}{\mathtt{sync}(x)\{\ P\ \} \bullet \sigma \mid \mathbb{P} \longrightarrow P \bullet \sigma \cdot x \mid \mathbb{P}}$$

reentrancy

DEADLOCKS IN `miniJava`

a program ($\mathbb{D}$, `P`) is **deadlock-free** if, whenever

$$\texttt{P} \bullet \varepsilon \longrightarrow^* \mathbb{P} \quad \text{and} \quad \mathbb{P} = \texttt{sync(x)\{ Q \}} \bullet \sigma \mid \mathbb{P}'$$

then there is $\mathbb{P}''$ such that $\mathbb{P} \to \mathbb{P}''$

e.g.

- $(\nu$ `sync(x){sync(y){ 0 }})` `sync(y){sync(x){ 0 }}` deadlocks
- $(\nu$ `sync(`z_n`){sync(`z_1`){0}})` *buildNetwork*`(`z_1`,`z_n`,n)` deadlocks for every `n`
- $(\nu$ `sync(`z_1`){sync(`z_n`){0}})` *buildNetwork*`(`z_1`,`z_n`,n)` never deadlocks

our behavioural types are **lams**

lams define sets of dependencies

* dependencies are relations containig pairs (x,y)

* the syntax is:

$$\ell \;=\; 0 \;\mid\; (x,y) \;\mid\; (\nu\; x)\ell \;\mid\; \ell \;\&\; \ell' \;\mid\; \ell + \ell' \;\mid\; \mathbf{A}(x_1,\ldots,x_n)$$

* a program is $(\mathcal{L},\ \ell)$ where $\mathcal{L}$ are lam-function definitions

$$\mathbf{A}(x_1,\ldots,x_n) = (\nu\; z_1,\ldots,z_m)\,\ell$$

* example of a lam-function:

$$\mathbf{buildNetwork}(x,y) \;=\; (\nu\; z)((x,y) \;+\; (x,z) \;\&\; \mathbf{buildNetwork}(z,y))$$

LAMS — THE RELATION $\sqsupseteq$

$$\ell \sqsupseteq \ell'$$

- if ℓ contains a function invocation $\mathtt{A}(u_1, ..., u_n)$
- $\mathtt{A}(x_1, ..., x_n) = (\nu\ z_1, ..., z_m)\ \ell_A$ is in $\mathscr{L}$
- ℓ' is ℓ where the invocation $\mathtt{A}(u_1, ..., u_n)$ has been replaced by $\ell_A\{^{z_1', ..., z_m'}/_{z_1, ..., z_m}\}\{^{u_1, ..., u_n}/_{x_1, ..., x_n}\}$ with $z_1', ..., z_m'$ fresh variables

example of $\sqsupseteq$:

$(y,x) \,\&\, \mathtt{buildNetwork}(x,y)$

$\sqsupseteq (y,x)\&(x,y) + (y,x)\&(x,z_1)\& \mathtt{buildNetwork}(z_1,y))$

$\sqsupseteq (y,x)\&(x,y) + (y,x)\&(x,z_1)\&[(z_1,y) + (z_1,z_2)\& \mathtt{buildNetwork}(z_2,y)]$

$= (y,x)\&(x,y) + (y,x)\&(x,z_1)\&(z_1,y) + (y,x)\&(x,z_1)\&(z_1,z_2)\& \mathtt{buildNetwork}(z_2,y)$

THE STATIC SEMANTICS OF `miniJava`

Processes – $\wr x_1 \cdot x_2 \cdots x_n \int = (x_1, x_2) \& \cdots \& (x_{n-1}, x_n)$:

(T-Zero)

$$\Gamma \;;\; \sigma \vdash \mathsf{0} : \wr \sigma \int$$

(T-New)

$$\frac{\Gamma,\ x : \mathtt{Obj} \;;\; \sigma \vdash P : \ell \qquad x \notin \sigma}{\Gamma ; \sigma \vdash (\nu\ x)\ P : (\nu\ x)\ \ell}$$

(T-Sync)

$$\frac{\Gamma \;;\; \sigma \cdot x \vdash P : \ell}{\Gamma \;;\; \sigma \vdash \mathtt{sync}(x)\{\ P\ \} : \ell}$$

(T-If)

$$\frac{\Gamma \vdash e : \mathtt{int} \qquad \Gamma \;;\; \sigma \vdash P : \ell \qquad \Gamma \;;\; \sigma \vdash P' : \ell'}{\Gamma \;;\; \sigma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' : \ell + \ell'}$$

(T-Par)

$$\frac{\Gamma \;;\; \sigma \vdash P : \ell \qquad \Gamma, u : \mathtt{Obj} \;;\; u \vdash Q : \ell'}{\Gamma \;;\; \sigma \vdash (\nu\ Q)\ P : \ell \,\&\, (\nu\ u)\ \ell'}$$

(T-Call)

$$\frac{\Gamma \vdash \overline{u} : \overline{\mathtt{Obj}} \qquad \Gamma \vdash \overline{e} : \overline{\mathtt{int}}}{\Gamma \;;\; \sigma \vdash A(\overline{u}, \overline{e}) : \mathsf{A}(\overline{u}) \,\&\, \wr \sigma \int}$$

Programs:

(T-Prog)

$$\frac{\Big(u : \mathtt{Obj} \;;\; u \vdash P_A : \ell_A\Big)^{A(\overline{x},\overline{y}) = P_A \in \mathbb{D}} \qquad \mathscr{L} = \bigcup_{A(\overline{x},\overline{y}) = P_A \in \mathbb{D}} \{\mathsf{A}(u, \overline{x}) = \ell_A\} \qquad \Gamma, u' : \mathtt{Obj} \;;\; u' \vdash P : \ell}{\Gamma \vdash (\mathbb{D}, P) : \big(\mathscr{L}, \ell\big)}$$

THE SUBJECT REDUCTION THEOREM

judgments for states

if $\Gamma \vdash \mathbb{P} : \ell$ and $\mathbb{P} \rightarrow \mathbb{P}'$ then there are Γ', ℓ' such that $\Gamma' \vdash \mathbb{P}' : \ell'$ and $\ell \sqsupseteq \ell'$

THE SECOND HALF: CIRCULAR DEPENDENCIES IN LAMS

example of $\sqsupseteq$:

this is a **circular dependency**

this is also a **circular dependency** (by transitive closure)

$(y,x)\,\&$ **buildNetwork**(x,y)

$\sqsupseteq\ (y,x)\&(x,y)\ +\ (y,x)\&(x,z_1)\&$ **buildNetwork**$(z_1,y))$

$\sqsupseteq\ (y,x)\&(x,y)\ +\ (y,x)\&(x,z_1)\&[\,(z_1,y)\ +\ (z_1,z_2)\&$ **buildNetwork**$(z_2,y)\,]$

$=\ (y,x)\&(x,y)\ +\ (y,x)\&(x,z_1)\&(z_1,y)\ +\ (y,x)\&(x,z_1)\&(z_1,z_2)\&$ **buildNetwork**(z_2,y)

a lam ℓ has a **circularity** if there is ℓ' such that

$\ell \sqsupseteq \ell'$ and ℓ' has a sumand with a circular dependency

THEOREM: it is decidable whether a lam ℓ has a **circularity** or not

THE SECOND HALF: CIRCULAR DEPENDENCIES IN LAMS

absence of circularities in lams corresponds to deadlock freedom in `miniJava` processes

THEOREM: if $\Gamma \vdash \mathbb{P} : \ell$ and ℓ has **no circularity** then $\mathbb{P}$ is **deadlock-free**

CONCLUSIONS

1. type inference relates programs and behavioural types

2. behavioural type analysis allows us to demonstrate sensible properties on programs