Evolving graph structures, non-interference, and complexity

Jean-Yves Marion
Joint work with Daniel Leivant
(ICALP 2013)

Université de Lorraine
Loria

November, 7th 2013

Outline

Outline

Machines on graph structures

Kolmogorov-Uspensky (63) and Schönhage (80):

The "tape" is a finite connected graph with an active node.
Instructions:

1. add a new node together with an edge between the active node and the new one,
2. remove a node and the edges incident to it,
3. add an edge,
4. remove an edge between two existing nodes,
5. halt.

Each instruction creates a next configuration by modifying the input graph structure.

Reference machines

Tarjan (1977)

A reference machine consists of

- a memory
- a finite numbers of registers.
- the memory is a finite set of records. Each record is a finite number of items identified by a name. All records are identical in structure.

Related to linking automaton of Knuth (1968).

Set union

```
while (q ≠ nil) {save := next(q);
                 parent(q) := r;
                 next(q) := next(r);
                 next(r) := q;
                 q := save}
```

Outline

Dynamic data structures

A **Graph Structure** is

- a set $\mathcal{V}$ of vertices,
- a set $\mathcal{D}$ of data,
- a graph vocabulary Σ with 2 interpretations:

  Edge function:

  The interpretation of $\mathbf{f} \in \Sigma$ is a partial $\mathbf{f}_{\mathcal{S}} : \mathcal{V} \to \mathcal{V} \cup \{\mathbf{nil}\}$
  - $dom(\mathbf{f}_{\mathcal{S}}) = \{a \mid \mathbf{f}_{\mathcal{S}}(a) \neq \mathbf{nil}\}$
  - $\mathbf{f}_{\mathcal{S}}$ is a partial function over $\mathcal{V}$

  Data function:

  The interpretation of $\mathbf{f} \in \Sigma$ is $\mathbf{f}_{\mathcal{S}} : \mathcal{V} \to \mathcal{D}$

Examples of Structures

Linked lists:

- An edge function **suc** and a data function **key**
- Two data constants **1**, **0** to represent binary words.

Directed graphs of arity d:

- d edge functions $\mathbf{e}_1, \ldots, \mathbf{e}_d$
- the out-degree of a vertex is $\leq d$
- the in-degree is unbounded

Dynamic data structures in real life

In C

```
typedef struct Cell {
  int key;
  struct Cell *suc;
} TypeCell;
typedef struct Cell *List;
```

also similar to objects . . .

Syntax of simple while imperative language

Expressions:

$$
\begin{array}{lcccl}
V & \in & \text{Vertex} & ::= & X \mid \mathbf{nil} \mid v \mid \mathbf{f}(V) \\
D & \in & \text{Data} & ::= & Y \mid d \mid \mathbf{g}(V) \\
B & \in & \text{Boolean} & ::= & V = V \mid D = D \mid \neg(B) \mid \mathbf{R}(E_1 \ldots E_n)
\end{array}
$$

Syntax of simple while imperative language

Expressions:

$$\begin{array}{lcccl} V & \in & \text{Vertex} & ::= & X \mid \mathbf{nil} \mid v \mid \mathbf{f}(V) \\ D & \in & \text{Data} & ::= & Y \mid d \mid \mathbf{g}(V) \\ B & \in & \text{Boolean} & ::= & V = V \mid D = D \mid \neg(B) \mid \mathbf{R}(E_1 \ldots E_n) \end{array}$$

A skeletal imperative language, which supports pointers:

$$\begin{array}{lcl} P \in \text{Prg} & ::= & X{:=}V \mid Y{:=}D \\ & & \mid \mathbf{f}(X){:=}V \mid \mathbf{g}(X){:=}D \\ & & \mid \mathbf{new}(X) \\ & & \mid \mathbf{skip} \mid P; P \\ & & \mid \mathbf{if}\,(B)\,\{P\}\,\{P\} \\ & & \mid \mathbf{while}\,(B)\,\{P\} \end{array}$$

Outline

Evolving structures

A structure $\mathcal{S}$ has a vertex set $\mathcal{V}$ and a data set $\mathcal{D}$.
Let σ be a store which assigns a vertex of $\mathcal{V}$, a data of $\mathcal{D}$ or **nil** to variables.

- If we run an instruction $\mathbf{f}(X) := V$ on a structure S, it evolves to the structure $\mathcal{S}' = \mathcal{S}[\mathbf{f}(\sigma(X)) \leftarrow v]$

Evolving structures

A structure $\mathcal{S}$ has a vertex set $\mathcal{V}$ and a data set $\mathcal{D}$.
Let σ be a store which assigns a vertex of $\mathcal{V}$, a data of $\mathcal{D}$ or **nil** to variables.

- If we run an instruction $\mathbf{f}(X) := V$ on a structure S, it evolves to the structure $\mathcal{S}' = \mathcal{S}[\mathbf{f}(\sigma(X)) \leftarrow v]$

- If we run an instruction **New**(X) on a structure $\mathcal{S}$, it evolves to $\mathcal{S}'$ s.t. $\mathcal{V}' = \mathcal{V} \cup \{u\}$ and $u \notin \mathcal{V}$.

Copy a list in reverse orders

```
y = nil ;
while (x≠nil)
      { z:=y;
        New(y);
        suc(y):=z;
        x:=suc(x)}
```

When a **New** command is executed,

- We pick up a free vertex u (from a *reserve*)
- The variable X points to this new vertex u.
- The vertex domain increases.

Runtime

A *configuration* is a couple $(\mathcal{S}, \mu)$ consisting of

- a structure $\mathcal{S}$ and of a store μ.

A computation is a sequence of configurations:

$$(\mathcal{S}_0, \sigma_0) \Rightarrow (\mathcal{S}_1, \sigma_1) \Rightarrow \ldots \Rightarrow (\mathcal{S}_n, \sigma_n)$$

Runtime

A *configuration* is a couple $(\mathcal{S}, \mu)$ consisting of

- a structure $\mathcal{S}$ and of a store μ.

A computation is a sequence of configurations:

$$(\mathcal{S}_0, \sigma_0) \Rightarrow (\mathcal{S}_1, \sigma_1) \Rightarrow \ldots \Rightarrow (\mathcal{S}_n, \sigma_n)$$

Runtime:

$$\mathit{Time}_P(\mathcal{S}_0, \mu_0) = n$$

A program P is *running in polynomial time* if

$$\forall(\mathcal{S}, \mu) \qquad \mathit{Time}_P(\mathcal{S}, \mu) \leq k \cdot |\mathcal{S}|^k$$

where $|\mathcal{S}|$ is the cardinal of the vertex-universe $\mathcal{V}$.

Tarjan's register machines

Tarjan (1977)

A reference machine consists of

- a memory
- a finite numbers of registers.
- the memory is a finite set of records. Each record is a finite number of items identified by a name. All records are identical in structure.

Memory is a graph and the control consists in pointers on nodes

Proposition

Pure reference machines compute while programs with update and new commands, and conversely.

Abstract State Machines

Gurevich (1993)

- a vocabulary Σ with static and dynamic names
- a state is a structure (algebra) over Σ on a base set.
- programs are simultaneous structure updates
- a run is a sequence of states (evolving algebras)

- To create a new element, there is a *Reserve*
  - **(1)** The element is there, but not used, e.g. a Turing machine has an infinite tape
  - **(2)** The element is created, e.g. Gandy's formalization of machines

Questions

- What is the domain of computation ?
  - Free algebra, Abstract (Chuch) algebra
  - First order structures like graphs
- Does a domain of computation evolve ?
  - Where does come from new element ?
- What is the address space ?
- How to restrict computational complexity ?
  - By restricting the program syntax
  - By typing programs
  - By analyzing the information flow
- Is the restriction complete for a complexity class ?
- What's about lower bound
- What's about modifying programs

A partial matrix of ICC models

Model	Domain	Evolving	State Space	Expr.	Restrict.
TM	Tapes	No	Infinite	Turing complete	
Pointer machines	Graphs	Yes	Reserve	Turing complete	
ICC	Church algebra	No		Ptime	various
Hoffman Shöpp	Graphs	No	polynomially bounded	$\subseteq$ Logspace	syntactic
Jones Life-cons	Trees	No	polynomially	Logspace	syntactic
*	Graphs	No	polynomially	Logspace	syntactic
*	Graphs	Yes New	polynomially	Ptime	syntactic types
*	Graphs	New update	polynomially	Ptime	syntactic types

Outline

Log-space computation

A program P over graph-structures is a *jumping-program* if it uses no edge update and no new instruction.

Theorem

A language is accepted by a jumping-program iff it is decidable in LOGSPACE.

The proof is based on Jones cons-free imperative programming language WHILE$^{\setminus Cons}$.

Hofmann and Schöpp introduced pure pointer programs over static graph-structures.
See also Jumping automaton on graphs (JAG) of Cook & Rackoff

Outline

Program ramification

Program tiering or *ramification,* has been introduced by Bellantoni & Cook'92 and Leivant'94.

Program ramification

Program tiering or *ramification,* has been introduced by Bellantoni & Cook'92 and Leivant'94.

- Atomic types are elements of a lattice $\mathbb{T} = (T, \preceq, \mathbf{0}, \vee, \wedge)$
- Expression are typed thus:

$$\frac{\Gamma(X) = \alpha}{\Gamma, \Delta \vdash X : \alpha} \qquad \frac{\alpha \to \beta \in \Delta(\mathbf{f}) \qquad \Gamma, \Delta \vdash V : \alpha}{\Gamma, \Delta \vdash \mathbf{f}(V) : \beta}$$

Program ramification

Program tiering or *ramification,* has been introduced by Bellantoni & Cook'92 and Leivant'94.

- Atomic types are elements of a lattice $\mathbb{T} = (T, \preceq, \mathbf{0}, \vee, \wedge)$
- Expression are typed thus:

$$\frac{\Gamma(X) = \alpha}{\Gamma, \Delta \vdash X : \alpha} \qquad \frac{\alpha \to \beta \in \Delta(\mathbf{f}) \qquad \Gamma, \Delta \vdash V : \alpha}{\Gamma, \Delta \vdash \mathbf{f}(V) : \beta}$$

- Types of an edge or a data function **f** satisfies

Stable all types of **f** are of the form $\alpha \to \alpha$,

Reducing all types of **f** are of the form $\alpha \to \beta$, with $\beta \prec \alpha$,

Program ramification

Program tiering or *ramification,* has been introduced by Bellantoni & Cook'92 and Leivant'94.

- Atomic types are elements of a lattice $\mathbb{T} = (T, \preceq, \mathbf{0}, \vee, \wedge)$
- Expression are typed thus:

$$\frac{\Gamma(X) = \alpha}{\Gamma, \Delta \vdash X : \alpha} \qquad \frac{\alpha \to \beta \in \Delta(\mathbf{f}) \qquad \Gamma, \Delta \vdash V : \alpha}{\Gamma, \Delta \vdash \mathbf{f}(V) : \beta}$$

- Types of an edge or a data function **f** satisfies

  Stable all types of **f** are of the form $\alpha \to \alpha$,

  Reducing all types of **f** are of the form $\alpha \to \beta$, with $\beta \prec \alpha$,

- As a result, there is a downward information flow
- and no upward information flow!

Typing rules for explicit information flows

Edge modification

$$\frac{\Gamma, \Delta \vdash X : \alpha \qquad \Gamma, \Delta \vdash V : \alpha}{\Gamma, \Delta \vdash X{:=}V : \alpha}$$

$$\frac{\Gamma, \Delta \vdash \mathbf{f}(X) : \alpha \qquad \Gamma, \Delta \vdash V : \alpha}{\Gamma, \Delta \vdash \mathbf{f}(X){:=}V \ : \alpha}$$

Vertices are created at tier **0**:

$$\frac{\Gamma, \Delta \vdash X : \mathbf{0}}{\Gamma, \Delta \vdash \mathbf{new}(X) \ : \mathbf{0}}$$

Typing rules for implicit information flows

$$\frac{\Gamma, \Delta \vdash B : \alpha \qquad \Gamma, \Delta \vdash P : \alpha}{\Gamma, \Delta \vdash \textbf{while}(B)\{P\} : \alpha}\ \mathbf{0} \prec \alpha$$

$$\frac{\Gamma, \Delta \vdash P : \alpha \qquad \Gamma, \Delta \vdash P' : \beta}{\Gamma, \Delta \vdash P' \,;\, P' : \alpha \vee \beta}$$

$$\frac{\Gamma, \Delta \vdash B : \alpha \quad \Gamma, \Delta \vdash P_i : \alpha}{\Gamma, \Delta \vdash \textbf{if}\ (B)\{P_0\}\{P_1\}\ : \alpha}$$

Copy a list in reverse orders

```
y = nil ;
while (x≠nil)
      { z:=y;
        New(y);
        suc(y):=z;
        x:=suc(x)}
```

Copy a list in reverse orders

```
y = nil ;
while (x≠nil)
      { z:=y;
        New(y);
        suc(y):=z;
        x:=suc(x)}
```

Typing of copy

```
y⁰ = nil : 0;
while (x¹ ≠ nil)
      { z⁰:=y⁰:0;
        New(y⁰):0;
        suc(y⁰):=z⁰:0;
        x¹ := suc(x¹):1 }:1
```

Multiplication

```
v¹ := start¹ : 1;
while (u¹ ≠ nil)
                { v¹ := start¹ : 1;
                  while (v¹ ≠ nil)
                   { v¹ := suc(v¹) : 1;
                     ... } : 1
                     u¹ := suc(u¹) : 1 } : 1
```

Simulate a polynomial Turing Machine.

Secure information flow

Theorem (Non-Interference - informal)

In a computation of a well-typed program, the graph structures accessible by tier α variables does not modify graph structures, which are just accessible by higher tier β variables, if $\alpha < \beta$.

Secure information flow

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

$$\text{Unclassified} < \text{Confidential} < \text{Secret}$$

Volpano, Smith and Irvine (1996)

- A secure flow typing for While programs which is sound wrt a security model.
- A confidential values can not leak to an unclassified variable.

Intuition

Relations between non-interference, information flow and complexity (see Lics'11)

Information flow : two points of view

Security view	Complexity view
A lattice denotes a security policy	A lattice denotes the data ramification
Types for secure flow analysis	Types for complexity flow analysis
Integrity	Downward flows are allowed
Non-interference	Values of tier 0 are obtained by iterating on tier 1 values
Declassification	A value of tier 0 may be upgraded safely to tier 1

Outline

Stationary loops

A loop is stationary of tier α if no **f** of type $\alpha \to \alpha$ is modified therein.

Tree insertion in binary search trees

```
if (x¹ = nil) then {x¹:=T:1;}
  else {
   while ((x¹ ≠ nil) and (key(T¹) ≠ key(x¹)))
     {if (key(T¹) < key(x¹))
          then {p¹:=x¹; x¹:= left(x¹)¹}
          else {p¹:=x¹; x¹:=right(x¹)¹}
        }:1;
if (key(T¹) < key(p¹))
          then {left(p¹) := T¹:1}
          else {right(p¹):= T¹:1}
```

Tightly-modifying loops

A function **f** is probed if it occurs either in some assignment $X := V$ or in the guard of a loop or a branching command.

Set union

```
while (q¹ ≠ nil)¹ {save¹ := next(q¹):1;
                   parent(q¹) := r¹:1;
                   next(q¹) := next(r¹):1;
                   next(r¹) := q¹:1;
                   q¹ := save¹:1}
```

A loop is tightly-modifying if it has modified functions of type $\alpha \to \alpha$, but at most one of those is also probed.

Ptime characterization

A program P is *tightly-ramifiable* if it is well-typed and each loop of P is stationary or tightly-modifying

Theorem

A function over graph-structures is computable in polynomial time iff it is computed by a terminating and tightly-ramifiable program.

It is well typed but val is modified and both suc and val are probed:

An exponential-time program

```
u¹ := head¹ : 1;
while (u¹ ≠ nil )
      { if (val(u) == 1)¹)
  then  { val(u):=0; u:=suc(u)}:1
        else { val(u):=1; u:=head}:1    }:1
```

Proof : Non-Interference

Say $(\mathcal{S}_\Delta, \sigma_\Gamma)$ disregards vertices that are not reachable from some variable of tier **1** using edge functions of type $(\mathbf{1} \to \mathbf{1})$.

Lemma

Suppose $\Gamma, \Delta \vdash P : \alpha$, *and* $\mathcal{S}, \sigma \models P \Rightarrow \mathcal{S}', \sigma' \models P'$.
There is a configuration $(\mathcal{S}'', \sigma'')$ *such that*
1) $\mathcal{S}_\Delta, \sigma_\Gamma \models P \Rightarrow \mathcal{S}'', \sigma'' \models P'$,
2) and $(\mathcal{S}''_\Delta, \sigma''_\Gamma) = (\mathcal{S}'_\Delta, \sigma'_\Gamma)$.

Proof : Number of configurations is polynomially bounded

Lemma

Assume that $\Gamma, \Delta \vdash P : \alpha$*, and* P *is tightly-modifying.*
There is a $k > 0$ *such that if* $\mathcal{S}, \sigma \models P$ $(\Rightarrow)^t$ $\mathcal{S}', \sigma' \models P'$ *then* $t < k + |\mathcal{S}|^k$.

Proof : Number of configurations is polynomially bounded

Lemma

Assume that $\Gamma, \Delta \vdash P : \alpha$*, and* P *is tightly-modifying.*
There is a $k > 0$ *such that if* $\mathcal{S}, \sigma \models P$ $(\Rightarrow)^t$ $\mathcal{S}', \sigma' \models P'$ *then* $t < k + |\mathcal{S}|^k$.

Let $\mathcal{S}$ be a digraph of out-degree 1. We say that a set of vertices C *generates* $\mathcal{S}$ if every vertex in $\mathcal{S}$ is reachable by a path starting at C.

Proof : Number of configurations is polynomially bounded

Lemma

Assume that $\Gamma, \Delta \vdash P : \alpha$*, and* P *is tightly-modifying.*
There is a $k > 0$ *such that if* $\mathcal{S}, \sigma \models P$ $(\Rightarrow)^t$ $\mathcal{S}', \sigma' \models P'$ *then* $t < k + |\mathcal{S}|^k$.

Let $\mathcal{S}$ be a digraph of out-degree 1. We say that a set of vertices C *generates* $\mathcal{S}$ if every vertex in $\mathcal{S}$ is reachable by a path starting at C.

Lemma

The number (up to isomorphism) of digraphs with n vertices, and a generator of size k, is $\leq n^{2k^2}$.

Adding recursion

Augmenting our language with linear recursion

Search of a path in a graph

```
Proc search(v¹,w¹))
      {if (v=w)¹ return true:1;
      visited(v) := true:1;
      forall t¹ in AdjList(v)
      {if (visited(t)¹=false)
       if (search(t,w)¹=true) return true:1;}
      return false:1;}
```

Typing rules where $\mathbf{0} \prec \alpha$:

$$\frac{\Gamma, \Delta \vdash X : \alpha \quad \Gamma, \Delta \vdash E_i : \alpha}{\Gamma, \Delta \vdash X = F(E_1, \ldots, E_i) : \alpha} \qquad \frac{\Gamma, \Delta \vdash E : \alpha}{\Gamma, \Delta \vdash \mathbf{return} E : \alpha}$$

Thanks !