(Optimal) Program Analysis of Sequential and Parallel Programs

Markus Müller-Olm
Westfälische Wilhelms-Universität Münster, Germany

3rd Summer School on Verification Technology, Systems, and Applications

Luxemburg, September 6-10, 2010
Dream of Automatic Analysis

program analyzer result

\[ G(\Phi \rightarrow F\Psi) \]

specification of property

main()
{ x=17;
  if (x>63)
  { y=17; x=10; x=x+1; }
  else
  { x=42;
    while (y<99)
    { y=x+y; x=x+1; }
    y=11; }
  x=y+1;
  printf(x);
}
Fundamental Problem

Rice's Theorem (informal version):

All non-trivial semantic properties of programs from a Turing-complete programming language are undecidable.

Consequence:

For Turing-complete programming languages:

Automatic analyzers of semantic properties, which are both correct and complete are impossible.

😊
What can we do about it?

- Give up „automatic“: interactive approaches:
  - proof calculi, theorem provers, …

- Give up „sound“: ???

- Give up „complete“: approximative approaches:
  - Approximate analyses:
    - data flow analysis, abstract interpretation, type checking, …
  - Analyse weaker formalism:
    - model checking, reachability analysis, equivalence- or preorder-checking, …
What can we do about it?

- Give up „automatic“: interactive approaches:
  - proof calculi, theorem provers, …

- Give up „sound“: ???

- Give up „complete“: approximative approaches:
  - Approximate analyses:
    - data flow analysis, abstract interpretation, type checking, …
  - Analyse weaker formalism:
    - model checking, reachability analysis, equivalence- or preorder-checking, …
Overview

- Introduction
- Fundamentals of Program Analysis
  Excursion 1
- Interprocedural Analysis
  Excursion 2
- Analysis of Parallel Programs
  Excursion 3
  Appendix
- Conclusion

Apology for not giving proper credit in these lectures!
Overview

- Introduction
- **Fundamentals of Program Analysis**
  - Excursion 1
- Interprocedural Analysis
  - Excursion 2
- Analysis of Parallel Programs
  - Excursion 3
  - Appendix
- Conclusion

Apology for not giving proper credit in these lectures!
main() {
  x=17;
  if (x>63) {
    y=17; x=10; x=x+1;
  } else {
    x=x+42;
    while (y<99) {
      y=x+y; x=y+1;
    }
    y=11;
  }
  x=y+1;
}
Dead Code Elimination

Goal:
find and eliminate assignments that compute values which are never used

Fundamental problem:
undecidability
→ use approximate algorithm:
e.g.: ignore that guards prohibit certain execution paths

Technique:
1) perform live variables analyses:
   variable x is live at program point u iff
   there is a path from u on which x is used before it is modified

2) eliminate assignments to variables that are not live at the target point
Live Variables Analysis

0: x=17
1: y>63, ¬(y>63)
2: y:=17
3: x:=10
4: ¬(y<99), x:=x+1
5: ¬(y<99), x:=y+1
6: y<99, x:=y+1
7: y=x+y
8: ¬(y<99), y:=11
9: y:=11
10: x:=y+1
11:
Interpretation of Partial Orders in Approximate Program Analysis

$x \sqsubseteq y$:
- $x$ is more precise information than $y$.
- $y$ is a correct approximation of $x$.

$\square X$ for $X \subseteq L$, where $(L, \sqsubseteq)$ is the partial order:
the most precise information consistent with all informations $x \in X$.

Example:
order for live variables analysis:
- $(P(\text{Var}), \sqsubseteq)$ with $\text{Var} =$ set of variables in the program

Remark:
often dual interpretation in the literature!
Complete Lattice

Complete lattice \((L, \sqsubseteq)\):
- a partial order \((L, \sqsubseteq)\) for which the least upper bound, \(\sqcup X\), exists for all \(X \subseteq L\).

In a complete lattice \((L, \sqsubseteq)\):
- \(\sqcap X\) exists for all \(X \subseteq L\): \(\sqcap X = \bigcup \{ x \in L \mid x \sqsubseteq X \}\)
- least element \(\perp\) exists: \(\perp = \bigcup L = \bigcap \emptyset\)
- greatest element \(\top\) exists: \(\top = \bigcup \emptyset = \bigcap L\)

Example:
- for any set \(A\) let \(P(A) = \{X \mid X \subseteq A\}\) (power set of \(A\)).
- \((P(A), \subseteq)\) is a complete lattice.
- \((P(A), \supseteq)\) is a complete lattice.
Specifying Live Variables Analysis by a Constraint System

Compute (smallest) solution over \((L, \sqsubseteq) = (P(\text{Var}), \subseteq)\) of:

\[
\begin{align*}
A[\text{fin}] & \sqsupseteq \text{init}, \quad \text{for fin, the termination node} \\
A[u] & \sqsupseteq f_e(A[v]), \quad \text{for each edge } e = (u,s,v)
\end{align*}
\]

where \(\text{init} = \text{Var},\)

\[f_e : P(\text{Var}) \rightarrow P(\text{Var}), \quad f_e(x) = x \setminus \text{kill}_e \cup \text{gen}_e, \quad \text{with}\]
- \(\text{kill}_e = \text{variables assigned at } e\)
- \(\text{gen}_e = \text{variables used in an expression evaluated at } e\)
Specifying Live Variables Analysis by a Constraint System

Remarks:

1. Every solution is „correct“ (whatever this means).

2. The smallest solution is called MFP-solution; it comprises a value MFP[u] ∈ L for each program point u.

3. MFP abbreviates „maximal fixpoint“ for traditional reasons.

4. The MFP-solution is the most precise one.
Backwards vs. Forward Analyses

Live Variables Analysis is a **Backwards Analysis**, i.e.:

- analysis info flows *from target node to source node* of an edge
- the initial inequality is for the termination node of the flow graph

\[ A[te] \supseteq init, \text{ for } te, \text{ the termination point} \]
\[ A[u] \supseteq f_e(A[v]), \text{ for each edge } e = (u, s, v) \in E \]

Dually, there are **Forward Analyses** i.e.:

- analysis info flows *from source node to target node* of an edge.
- the initial inequality is for the start node of the flow graph

\[ A[st] \supseteq init, \text{ for } st, \text{ the start node} \]
\[ A[v] \supseteq f_e(A[u]), \text{ for each edge } e = (u, s, v) \in E \]

**Examples**: reaching definitions, available expressions, constant propagation, ...
Data-Flow Frameworks

Correctness

- generic properties of frameworks can be studied and proved

Implementation

- efficient, generic implementations can be constructed
Three Questions

- Do (smallest) solutions always exist?
- How to compute the (smallest) solution?
- How to justify that a solution is what we want?
Three Questions

- Do (smallest) solutions always exist?
  - How to compute the (smallest) solution?
  - How to justify that a solution is what we want?
Knaster-Tarski Fixpoint Theorem

Definitions:
Let \((L, \sqsubseteq)\) be a partial order.
- \(f : L \to L\) is \textit{monotonic} iff \(\forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)\).
- \(x \in L\) is a \textit{fixpoint} of \(f\) iff \(f(x) = x\).

Fixpoint Theorem of Knaster-Tarski:
Every monotonic function \(f\) on a complete lattice \(L\) has a least fixpoint \(\text{lfp}(f)\) and a greatest fixpoint \(\text{gfp}(f)\).

More precisely,
\[
\text{lfp}(f) = \bigcap \{ x \in L \mid f(x) \sqsubseteq x \} \quad \text{least pre-fixpoint}
\]
\[
\text{gfp}(f) = \bigcup \{ x \in L \mid x \sqsubseteq f(x) \} \quad \text{greatest post-fixpoint}
\]
Knaster-Tarski Fixpoint Theorem

L:

- $\top$ (pre-fixpoints of $f$)
- $\text{gfp}(f)$
- $\text{fixpoints of } f$
- $\text{ifp}(f)$
- $\bot$ (post-fixpoints of $f$)

Picture from: Nielson/Nielson/Hankin, *Principles of Program Analysis*
Smallest Solutions Always Exist

- Define functional $F : L^n \rightarrow L^n$ from right hand sides of constraints such that:
  - $\sigma$ solution of constraint system iff $\sigma$ pre-fixpoint of $F$

- Functional $F$ is monotonic.

- By Knaster-Tarski Fixpoint Theorem:
  - $F$ has a least fixpoint which equals its least pre-fixpoint.
Three Questions

- Do (smallest) solutions always exist?
- How to compute the (smallest) solution?
- How to justify that a solution is what we want?
Workset-Algorithm

\[ W = \emptyset; \]
\[ \text{forall (program points } v) \{ \hspace{1cm} A[v] = \bot; \hspace{1cm} W = W \cup \{v\}; \} \]
\[ A[\text{fin}] = \text{init}; \]
\[ \text{while } W \neq \emptyset \{ \]
\[ \hspace{1cm} v = \text{Extract}(W); \]
\[ \hspace{1cm} \text{forall (u, s with edge } e = (u, s, v) \{ \]
\[ \hspace{2cm} t = f_e(A[v]); \]
\[ \hspace{2cm} \text{if } \neg(t \sqsubseteq A[u]) \{ \]
\[ \hspace{3cm} A[u] = A[u] \cup t; \]
\[ \hspace{3cm} W = W \cup \{u\}; \]
\[ \hspace{2cm} \} \]
\[ \hspace{1cm} \} \]
\[ \} \]
Invariants of the Main Loop

a) \( A[u] \sqsubseteq MFP[u] \) f.a. prg. points \( u \)

b1) \( A[\text{fin}] \sqsupseteq init \)

b2) \( v \notin W \Rightarrow A[u] \sqsupseteq f_e(A[v]) \) f.a. edges \( e = (u, s, v) \)

If and when workset algorithm terminates:

\( A \) is a solution of the constraint system by b1)&b2)

\[ \Rightarrow A[u] \sqsubseteq MFP[u] \quad \text{f.a. } u \]

Hence, with a): \( A[u] = MFP[u] \quad \text{f.a. } u \) 😊
How to Guarantee Termination

- Lattice \((L, \sqsubseteq)\) has finite heights
  \[ \Rightarrow \text{algorithm terminates after at most } \#\text{prg points} \cdot (\text{heights}(L)+1) \text{ iterations of main loop} \]

- Lattice \((L, \sqsubseteq)\) has no infinite ascending chains
  \[ \Rightarrow \text{algorithm terminates} \]

- Lattice \((L, \sqsubseteq)\) has infinite ascending chains:
  \[ \Rightarrow \text{algorithm may not terminate;} \]
  use \textit{widening operators} in order to enforce termination
\( \nabla : L \times L \rightarrow L \) is called a *widening operator* iff

1) \( \forall x, y \in L : x \sqcup y \sqsubseteq x \nabla y \)

2) for all sequences \( (l_n)_n \), the (ascending) chain \( (w_n)_n \)

\[
w_0 = l_0, \quad w_{i+1} = w_i \nabla l_{i+1} \text{ for } i > 0
\]

stabilizes eventually.
Workset-Algorithm with Widening

\[ W = \emptyset; \]
\[ \text{forall (program points } v) \{ A[v] = \perp; W = W \cup \{v\}; \} \]
\[ A[\text{fin}] = \text{init}; \]
\[ \text{while } W \neq \emptyset \{ \]
\[ v = \text{Extract}(W); \]
\[ \text{forall (u, s with } e = (u, s, v) \text{ edge)} \{ \]
\[ t = f_e(A[v]); \]
\[ \text{if } \neg(t \subseteq A[u]) \{ \]
\[ A[u] = A[u] \triangledown t; \]
\[ W = W \cup \{u\}; \]
\[ \} \]
\[ \} \]
Invariants of the Main Loop

a) \[ A[u] \subseteq \text{MFP}[u] \] f.a. prg. points \( u \)

b1) \[ A[\text{fin}] \supseteq \text{init} \]

b2) \[ \nu \notin W \implies A[u] \supseteq f_e(A[\nu]) \] f.a. edges \( e = (u,s,\nu) \)

With a widening operator we enforce termination but we lose invariant a).

Upon termination, we have:

\[ A \text{ is a solution of the constraint system by b1)\&b2) } \]
\[ \implies A[u] \supseteq \text{MFP}[u] \] f.a. \( u \)

Compute a sound upper approximation (only)!
Example of a Widening Operator: Interval Analysis

The goal

Find save interval for the values of program variables, e.g. of \( i \) in:

\[
\text{for } (i=0; \ i<42; \ i++) \\
\quad \text{if } (0<=i \ \text{and} \ i<42) \\
\quad \{ \\
\quad \quad A1 = A+i; \\
\quad \quad M[A1] = i; \\
\quad \}
\]

..., e.g., in order to remove the redundant array range check.
Example of a Widening Operator: Interval Analysis

The lattice...
\[(L, \sqsubseteq) = \left( \{ [l,u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u \} \cup \{\emptyset\}, \sqsubseteq \right)\]

... has infinite ascending chains, e.g.:
\[ [0,0] \subset [0,1] \subset [0,2] \subset \ldots \]

A widening operator:
\[ [l_0,u_0] \uplus [l_1,u_1] = [l_2,u_2], \text{ where} \]
\[ l_2 = \begin{cases} l_0 & \text{if } l_0 \leq l_1 \\ -\infty & \text{otherwise} \end{cases} \quad \text{and} \quad u_2 = \begin{cases} u_0 & \text{if } u_0 \geq u_1 \\ +\infty & \text{otherwise} \end{cases} \]

A chain of maximal length arising with this widening operator:
\[ \emptyset \subset [3,7] \subset [3,\infty] \subset [-\infty,\infty] \]
Analyzing the Program with the Widening Operator

⇒ Result is far too imprecise!

Example taken from: H. Seidl, Vorlesung „Programmoptimierung“
Remedy 1: Loop Separators

- Apply the widening operator only at a "loop separator" (a set of program points that cuts each loop).
- We use the loop separator \{1\} here.

\[\Rightarrow\] Identify condition at edge from 2 to 3 as redundant! 😊
Remedy 2: Narrowing

- Iterate again from the result obtained by widening
  --- Iteration from a prefix-point stays above the least fixpoint! ---

⇒ We get the exact result in this example (but not guaranteed)!

![Diagram and table showing the process and result.]
Remarks

- Can use a work-list instead of a work-set
- Special iteration strategies in special situations
- Semi-naive iteration
Recall: Specifying Live Variables Analysis by a Constraint System

Compute (smallest) solution over \((L, \sqsubseteq) = (P(\text{Var}), \subseteq)\) of:

\[
\begin{align*}
A[\text{fin}] & \sqsubseteq \text{init}, & \text{for } \text{fin}, \text{ the termination node} \\
A[u] & \sqsubseteq f_e(A[v]), & \text{for each edge } e = (u, s, v)
\end{align*}
\]

where \(\text{init} = \text{Var},\)

\[f_e : P(\text{Var}) \to P(\text{Var}), \quad f_e(x) = x \setminus \text{kill}_e \cup \text{gen}_e,\]

with

- \(\text{kill}_e = \text{variables assigned at } e\)
- \(\text{gen}_e = \text{variables used in an expression evaluated at } e\)
Recall: Questions

- Do (smallest) solutions always exist?
- How to compute the (smallest) solution?
- How to justify that a solution is what we want?
Three Questions

- Do (smallest) solutions always exist?
- How to compute the (smallest) solution?
- How to justify that a solution is what we want?
  - MOP vs MFP-solution
  - Abstract interpretation
Three Questions

- Do (smallest) solutions always exist?

- How to compute the (smallest) solution?

- How to justify that a solution is what we want?
  - MOP vs MFP-solution
  - Abstract interpretation
Assessing Data Flow Frameworks

Execution Semantics → Abstraction → MOP-solution

sound? how precise?

MFP-solution

sound? precise?
Live Variables

\[ \text{MOP}[y] = \emptyset \cup \{ y \} = \{ y \} \]
Meet-Over-All-Paths Solution (MOP)

- Forward Analysis

\[ \text{MOP}[u] := \bigsqcup_{p \in \text{Paths}[\text{entry}, u]} F_p(\text{init}) \]

- Backward Analysis

\[ \text{MOP}[u] := \bigsqcup_{p \in \text{Paths}[u, \text{exit}]} F_p(\text{init}) \]

- Here: „Join-over-all-paths“; MOP traditional name
Coincidence Theorem

Definition:
A framework is positively-distributive if
\[ f(\sqcup X) = \sqcup \{ f(x) \mid x \in X \} \] for all \( \emptyset \neq X \subseteq L \), \( f \in F \).

Theorem:
For any instance of a positively-distributive framework:
\[ MOP[u] = MFP[u] \] for all program points \( u \)
(if all program points reachable).

Remark:
A framework is positively-distributive if a) and b) hold:
(a) it is distributive:
\[ f(x \sqcup y) = f(x) \sqcup f(y) \] f.a. \( f \in F \), \( x,y \in L \).
(b) it is effective:
\( L \) does not have infinite ascending chains.

Remark: All bitvector frameworks are distributive and effective.
Lattice for Constant Propagation

Lattice $L$: \[ \{ \rho \mid \rho : \text{Var} \rightarrow (\mathbb{Z} \cup \{\top\}) \} \cup \{\bot\} \]

\[ \rho \sqsubseteq \rho' : \iff \rho = \bot \lor \]

\[ (\rho, \rho' \neq \bot \land \forall x : \rho(x) \sqsubseteq \rho'(x)) \]
\begin{equation}
\begin{align*}
\text{MOP}[v] &= (\top, \top, 5) \\
(x, y, z) &= (2, 3, 5) \\
(x, y) &= (3, 2, 5)
\end{align*}
\end{equation}
\begin{equation}
\begin{aligned}
(\rho(x), \rho(y), \rho(z)) \\
(\top, \top, \top) \\
(\top, \top, \top) \\
(\top, \top, \top) \\
\end{aligned}
\end{equation}

\begin{equation}
\begin{aligned}
x &:= 2 \\
\text{y} &:= 3 \\
z &:= x+y \\
\text{out}(x) \\
\end{aligned}
\end{equation}

\begin{equation}
\begin{aligned}
x &:= 3 \\
y &:= 2 \\
z &:= x+y \\
\end{aligned}
\end{equation}

\begin{equation}
\begin{aligned}
\text{MFP}[v] &=(\top, \top, \top) \\
\text{MOP}[v] &=(\top, \top, 5) \\
\end{aligned}
\end{equation}
Correctness Theorem

Definition:
A framework is monotone if for all $f \in F$, $x, y \in L$:
\[ x \sqsubseteq y \implies f(x) \sqsubseteq f(y) \, . \]

Theorem:
In any monotone framework:
\[ \text{MOP}[u] \sqsubseteq \text{MFP}[u] \] for all program points $u$.

Remark:
Any "reasonable" framework is monotone. 🙂
Assessing Data Flow Frameworks

- Execution Semantics
- Abstraction
- MOP-solution
- MFP-solution

sound

sound

precise, if distrib.
Where Flow Analysis Looses Precision

Potential loss of precision
Three Questions

- Do (smallest) solutions always exist?
- How to compute the (smallest) solution?
- How to justify that a solution is what we want?
  - MOP vs MFP-solution
  - Abstract interpretation
Abstract Interpretation

Often used as reference semantics:

- sets of reaching runs:
  \[(D, \sqsubseteq) = (P(\text{Edges}^*), \subseteq)\] or \[(D, \sqsubseteq) = (P(\text{Stmt}^*), \subseteq)\]

- sets of reaching states ("collecting semantics"):
  \[(D, \sqsubseteq) = (P(\Sigma^*), \subseteq)\] with \[\Sigma = \text{Var} \rightarrow \text{Val}\]
Assume a universally-disjunctive abstraction function $\alpha : D \to D^\#$.

**Correct abstract interpretation:**

Show $\alpha(o(x_1, \ldots, x_k)) \sqsubseteq^\# o^\#(\alpha(x_1), \ldots, \alpha(x_k))$ f.a. $x_1, \ldots, x_k \in L$, operators $o$

Then $\alpha(MFP[u]) \sqsubseteq^\# MFP^\#[u]$ f.a. $u$

**Correct and precise abstract interpretation:**

Show $\alpha(o(x_1, \ldots, x_k)) = o^\#(\alpha(x_1), \ldots, \alpha(x_k))$ f.a. $x_1, \ldots, x_k \in L$, operators $o$

Then $\alpha(MFP[u]) = MFP^\#[u]$ f.a. $u$

Use this as a guideline for designing correct (and precise) analyses!
Abstract Interpretation

Constraint system for reaching runs:

\[ R[st] \supseteq \{ \varepsilon \}, \quad \text{for st, the start node} \]

\[ R[v] \supseteq R[u] \cdot \{ e \}, \quad \text{for each edge } e = (u, s, v) \]

Operational justification:

Let \( R[u] \) be components of smallest solution over \( P(Edges^*) \). Then

\[ R[u] = R^{op}[u] =_{\text{def}} \{ r \in Edges^* | st \xrightarrow{r} u \} \quad \text{for all } u \]

Prove:

a) \( R^{op}[u] \) satisfies all constraints (direct)

\[ \Rightarrow R[u] \subseteq R^{op}[u] \quad \text{f.a. } u \]

b) \( w \in R^{op}[u] \Rightarrow w \in R[u] \) (by induction on \( |w| \))

\[ \Rightarrow R^{op}[u] \subseteq R[u] \quad \text{f.a. } u \]
Abstract Interpretation

Constraint system for reaching runs:

\[
R[st] \supseteq \{\varepsilon\}, \quad \text{for } st, \text{ the start node}
\]

\[
R[v] \supseteq R[u] \cdot \{\langle e \rangle\}, \quad \text{for each edge } e = (u, s, v)
\]

Derive the analysis:

Replace

\[
\{\varepsilon\} \quad \text{by } \text{init}
\]

\[
(\bullet) \cdot \{\langle e \rangle\} \quad \text{by } f_e
\]

Obtain abstracted constraint system:

\[
R^*[st] \supseteq \text{init}, \quad \text{for } st, \text{ the start node}
\]

\[
R^*[v] \supseteq f_e(R^*[u]), \quad \text{for each edge } e = (u, s, v)
\]
Abstract Interpretation

MOP-Abstraction:
Define $\alpha_{\text{MOP}} : \mathcal{P}(\text{Edges}^*) \rightarrow \mathcal{L}$ by

$$\alpha_{\text{MOP}}(R) = \bigsqcup \{ f_r(\text{init}) \mid r \in R \} \quad \text{where} \quad f_\varepsilon = \text{Id}, \quad f_{s, \langle e \rangle} = f_e \circ f_s$$

Remark:
For all transfer functions $f_e$ are monotone, the abstraction is correct:
$$\alpha_{\text{MOP}}(R[u]) \subseteq R^#[u] \quad \text{f.a. prg. points } u$$

If all transfer function $f_e$ are universally-distributive, the abstraction is correct and precise:
$$\alpha_{\text{MOP}}(R[u]) = R^#[u] \quad \text{f.a. prg. points } u$$

Justifies MOP vs. MFP theorems (*cum grano salis*). 😊
Overview

- Introduction
- Fundamentals of Program Analysis

---

Excursion 1

- Interprocedural Analysis
  - Excursion 2

- Analysis of Parallel Programs
  - Excursion 3
  - Appendix

- Conclusion
Challenges for Automatic Analysis

- Data aspects:
  - infinite number domains
  - dynamic data structures (e.g. lists of unbounded length)
  - pointers
  - ...

- Control aspects:
  - recursion
  - concurrency
  - creation of processes / threads
  - synchronization primitives (locks, monitors, communication stmts ...)
  - ...

⇒ infinite/unbounded state spaces
Classifying Analysis Approaches

control aspects

data aspects

analysis techniques
(My) Main Interests of Recent Years

Data aspects:
- algebraic invariants over $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{Z}_m$ ($m = 2^n$) in sequential programs, partly with recursive procedures
- invariant generation relative to Herbrand interpretation

Control aspects:
- recursion
- concurrency with process creation / threads
- synchronization primitives, in particular locks/monitors

Technics:
- fixpoint-based
- automata-based
- (linear) algebra
- syntactic substitution-based techniques
- ...
Overview

- Introduction
- Fundamentals of Program Analysis
  - Excursion 1
- Interprocedural Analysis
  - Excursion 2
- Analysis of Parallel Programs
  - Excursion 3
  - Appendix
- Conclusion
A Note on Karr’s Algorithm

Markus Müller-Olm
FernUniversität Hagen
(on leave from Universität Dortmund)

Joint work with
Helmut Seidl (TU München)

ICALP 2004, Turku, July 12-16, 2004
What this Excursion is About...

0

\[ x_1 := 1 \]
\[ x_2 := 1 \]
\[ x_3 := 1 \]

1

\[ x_1 := x_1 + 1 \]
\[ x_2 := 2x_2 - 2x_1 + 5 \]
\[ x_3 := x_3 + x_2 \]

2

\[ x_3 = x_1^2 \]
\[ x_2 = 2x_1 - 1 \]
Affine Programs

- Basic Statements:
  - affine assignments: \( x_1 := x_1 - 2x_3 + 7 \)
  - unknown assignments: \( x_i := ? \)
    \[ \rightarrow \text{abstract too complex statements} \]

- Affine Programs:
  - control flow graph \( G = (N, E, st) \), where
    - \( N \) finite set of program points
    - \( E \subset N \times \text{Stmt} \times N \) set of edges
    - \( st \in N \) start node

- Note: non-deterministic instead of guarded branching
The Goal: Precise Analysis

Given an affine program, determine for each program point

- all valid affine relations:
  \[ a_0 + \sum a_i x_i = 0 \quad a_i \in \mathbb{Q} \]

More ambitious goal:
- determine all valid polynomial relations (of degree \( \leq d \)):
  \[ p(x_1, \ldots, x_k) = 0 \quad p \in \mathbb{Q}[x_1, \ldots, x_n] \]
Applications of Affine (and Polynomial) Relations

- Data-flow analysis:
  - definite equalities: \( x = y \)
  - constant detection: \( x = 42 \)
  - discovery of symbolic constants: \( x = 5yz + 17 \)
  - complex common subexpressions: \( xy + 42 = y^2 + 5 \)
  - loop induction variables

- Program verification
  - strongest valid affine (or polynomial) assertions
    (cf. Petri Net invariants)
Karr’s Algorithm \[\text{[Karr, 1976]}\]

- Determines valid affine relations in programs.

- **Idea**: Perform a data-flow analysis maintaining for each program point a set of affine relations, i.e., a linear equation system.

- **Fact**: Set of valid affine relations forms a vector space of dimension at most \(k+1\), where \(k = \#\text{program variables}\).

  \[\Rightarrow\] can be represented by a basis.

  \[\Rightarrow\] forms a complete lattice of height \(k+1\).
Deficiencies of Karr’s Algorithm

- Basic operations are complex
  - „non-invertible“ assignments
  - union of affine spaces

- $O(n \cdot k^4)$ arithmetic operations
  - $n$ size of the program
  - $k$ number of variables

- Numbers may have exponential length
Our Contribution

- Reformulation of Karr’s algorithm:
  - basic operations are simple
  - $O(n \cdot k^3)$ arithmetic operations
  - numbers stay of polynomial length: $O(n \cdot k^2)$
  Moreover:
  - generalization to polynomial relations of bounded degree
  - show, algorithm finds all affine relations in „affine programs“

- Ideas:
  - represent affine spaces by affine bases instead of lin. eq. syst.
  - use semi-naive fixpoint iteration
  - keep a reduced affine basis for each program point during fixpoint iteration
Affine Basis
Concrete Collecting Semantics

Smallest solution over subsets of $\mathbb{Q}^k$ of:

$V[st] \supseteq \mathbb{Q}^k$

$V[v] \supseteq f_s(V[u])$, for each edge $(u,s,v)$

where

$f_{x_i:=t}(X) = \{x[x_i \mapsto t(x)] \mid x \in X\}$

$f_{x_i:=?}(X) = \{x[x_i \mapsto c] \mid x \in X, c \in \mathbb{Q}\}$

First goal: compute affine hull of $V[u]$ for each $u$. 
Abstraction

Affine hull:

\[ \text{aff}(X) = \{ \sum \lambda_i x_i \mid x_i \in X, \lambda_i \in \mathbb{Q}, \sum \lambda_i = 1 \} \]

The affine hull operator is a closure operator:

\[ \text{aff}(X) \supseteq X, \text{aff}(\text{aff}(X)) = X, \ X \subseteq Y \Rightarrow \text{aff}(X) \subseteq \text{aff}(Y) \]

\[ \Rightarrow \text{Affine subspaces of } \mathbb{Q}^k \text{ ordered by set inclusion form a complete lattice:} \]

\[ (D, \subseteq) = \left( \{ X \subseteq \mathbb{Q}^k \mid \text{aff}(X) = X \}, \subseteq \right). \]

Affine hull is even a precise abstraction:

Lemma: \( f_s(\text{aff}(X)) = \text{aff}(f_s(X)). \)
Abstract Semantics

Smallest solution over \((D, \sqsubseteq)\) of:

\[
V^\#[st] \supseteq \mathbb{Q}^k
\]

\[
V^\#[v] \supseteq f_s(V^\#[u]), \text{ for each edge } (u, s, v)
\]

Lemma: \(V^\#[u] = \text{aff}(V[u])\) for all program points \(u\).
Basic Semi-naive Fixpoint Algorithm

\[
\text{forall } (v \in N) \quad G[v] = \emptyset ; \\
G[st] = \{0, e_1, \ldots, e_k\}; \\
W = \{(st,0),(st,e_1),\ldots,(st,e_k)\}; \\
\textbf{while } W \neq \emptyset \quad \{ \\
\quad (u, x) = \text{Extract}(W); \\
\quad \text{forall } (s, v \text{ with } (u, s, v) \in E) \quad \{ \\
\quad \quad t = [s] x; \\
\quad \quad \textbf{if } (t \notin \text{aff}(G[v])) \quad \{ \\
\quad \quad \quad G[v] = G[v] \cup \{t\}; \\
\quad \quad \quad W = W \cup \{(v, t)\}; \\
\quad \quad \} \\
\quad \} \\
\}
\]
Example

\[\begin{align*}
    x_1 &:= 1 \\
    x_2 &:= 1 \\
    x_3 &:= 1 \\
    x_1 &:= x_1 + 1 \\
    x_2 &:= 2x_2 - 2x_1 + 5 \\
    x_3 &:= x_3 + x_2
\end{align*}\]

\[
\begin{bmatrix}
    0 \\
    0 \\
    0
\end{bmatrix},
\begin{bmatrix}
    1 \\
    1 \\
    1
\end{bmatrix},
\begin{bmatrix}
    1 \\
    1 \\
    1
\end{bmatrix}
\]

\[
\begin{bmatrix}
    1 \\
    1 \\
    1
\end{bmatrix},
\begin{bmatrix}
    2 \\
    3 \\
    4
\end{bmatrix},
\begin{bmatrix}
    3 \\
    4 \\
    5
\end{bmatrix},
\begin{bmatrix}
    4 \\
    7 \\
    16
\end{bmatrix}
\]

\[
\begin{bmatrix}
    0 \\
    0 \\
    0
\end{bmatrix},
\begin{bmatrix}
    1 \\
    0 \\
    0
\end{bmatrix},
\begin{bmatrix}
    0 \\
    1 \\
    0
\end{bmatrix},
\begin{bmatrix}
    0 \\
    0 \\
    1
\end{bmatrix}
\]

\[
\begin{bmatrix}
    1 \\
    1 \\
    1
\end{bmatrix},
\begin{bmatrix}
    2 \\
    3 \\
    4
\end{bmatrix},
\begin{bmatrix}
    3 \\
    4 \\
    5
\end{bmatrix},
\begin{bmatrix}
    4 \\
    7 \\
    16
\end{bmatrix}
\]

\[
\begin{bmatrix}
    1 \\
    1 \\
    1
\end{bmatrix},
\begin{bmatrix}
    1 \\
    3 \\
    4
\end{bmatrix},
\begin{bmatrix}
    1 \\
    4 \\
    5
\end{bmatrix}
\]

\[x \in \text{aff} \left\{ \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix}, \begin{bmatrix} 3 \\ 4 \\ 9 \end{bmatrix} \right\}\]
Correctness

Theorem:
   a) Algorithm terminates after at most $nk + n$ iterations of the loop, where $n = |\mathcal{N}|$ and $k$ is the number of variables.
   b) For all $v \in \mathcal{N}$, we have $\text{aff}(G_{\text{fin}}[v]) = V^\#[v]$.

Invariants for b)
   I1: $\forall v \in \mathcal{N}: G[v] \subseteq V[v]$ and $\forall (u, x) \in W : x \in V[u]$.
   I2: $\forall (u, s, v) \in E: \text{aff}\left(G[v] \cup \{[s] \mid (u, x) \in W\}\right) \supseteq f_s(\text{aff}(G[u]))$. 
Complexity

Theorem:

a) The affine hulls $V^\#[u] = \text{aff}(V[u])$ can be computed in time $O(n \cdot k^3)$, where $n = |N| + |E|$.

b) In this computation only arithmetic operations on numbers with $O(n \cdot k^2)$ bits are used.

Store diagonal basis for membership tests.
Propagate original vectors.
Point + Linear Basis
Example

\[ x_1 := 1 \]
\[ x_2 := 1 \]
\[ x_3 := 1 \]

\[ x_1 := x_1 + 1 \]
\[ x_2 := 2x_2 - 2x_1 + 5 \]
\[ x_3 := x_3 + x_2 \]
Determining Affine Relations

Lemma: $a$ is valid for $X$ $\iff$ $a$ is valid for $\text{aff}(X)$.

$\Rightarrow$ suffices to determine the affine relations valid for affine bases; can be done with a linear equation system!

Theorem:

a) The vector spaces of all affine relations valid at the program points of an affine program can be computed in time $O(n \cdot k^3)$.
b) This computation performs arithmetic operations on integers with $O(n \cdot k^2)$ bits only.
Example

\[ a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 = 0 \] is valid at 2

\[ a_0 + 2a_1 + 3a_2 + 4a_3 = 0 \]

\[ 1a_1 + 2a_2 \]

\[ 2a_3 = 0 \]

\[ a_0 = a_2, \quad a_1 = -2a_2, \quad a_3 = 0 \]

\[ 2x_1 - x_2 - 1 \] is valid at 2

\[ a_0 \]

\[ a_1 \]

\[ a_2 \]

\[ a_3 \]
Also in the Paper

- Non-deterministic assignments
- Bit length estimation
- Polynomial relations
- Affine programs + affine equality guards
  - validity of affine relations undecidable
End of Excursion 1
(Optimal) Program Analysis of Sequential and Parallel Programs

Markus Müller-Olm
Westfälische Wilhelms-Universität Münster, Germany

3rd Summer School on Verification Technology, Systems, and Applications

Luxemburg, September 6-10, 2010
Overview

- Introduction
- Fundamentals of Program Analysis
  Excursion 1

- Interprocedural Analysis
  Excursion 2

- Analysis of Parallel Programs
  Excursion 3
  Appendix

- Conclusion
Interprocedural Analysis

Main:

P(): c:=a+b

Q():

R():

c:=a+b

P(): c:=a+b

Q:

a:=7

c:=a+b

R:

a:=7

c:=a+b

R:

Q:

P():

call edges

procedures

recursion
Running Example:
(Definite) Availability of the single expression $a+b$

The lattice:

```
false  a+b not available
|      |
true  a+b available
```

Initial value: false
Intra-Procedural-Like Analysis

Conservative assumption: procedure destroys all information; information flows from call node to entry point of procedure

The lattice:

\[ P(\lambda x. \text{false}) \]

\[ \text{false} \]

\[ \text{true} \]

\[ \text{true} \]

\[ \text{false} \]

\[ \text{false} \]

\[ \text{false} \]

\[ \text{false} \]

\[ \text{true} \]

\[ \text{false} \]

\[ \text{false} \]

\[ \text{false} \]

\[ \text{false} \]

\[ \text{true} \]

\[ \text{false} \]

\[ \text{true} \]
Context-Insensitive Analysis

Conservative assumption: Information flows from each call node to entry of procedure and from exit of procedure back to return point.

The lattice:
Conservative assumption: Information flows from each call node to entry of procedure and from exit of procedure back to return point.
Assume a universally-disjunctive abstraction function $\alpha : D \rightarrow D^\#$. 

**Correct abstract interpretation:**

Show $\alpha(o(x_1,\ldots,x_k)) \sqsubseteq^# o^\#(\alpha(x_1),\ldots,\alpha(x_k))$ f.a. $x_1,\ldots,x_k \in L$, operators $o$

Then $\alpha(MFP[u]) \sqsubseteq^# MFP^#[u]$ f.a. $u$

**Correct and precise abstract interpretation:**

Show $\alpha(o(x_1,\ldots,x_k)) = o^\#(\alpha(x_1),\ldots,\alpha(x_k))$ f.a. $x_1,\ldots,x_k \in L$, operators $o$

Then $\alpha(MFP[u]) = MFP^#[u]$ f.a. $u$

Use this as a guideline for designing correct (and precise) analyses!
Main:

$\text{e}_0: \ c := a + b$

$\text{e}_1: \ P()$

$\text{e}_2: \ a := 7$

$\text{e}_3: \ P()$

$\text{r}_M$

The lattice:

false

true

$\text{P:}$

$\text{e}_4: \ c := a + b$

$\text{r}_P$
Let's Apply Our Abstract Interpretation Recipe: Constraint System for Feasible Paths

Operational justification:

\[
S(u) = \left\{ r \in \text{Edges}^* \mid \text{st}_p \xrightarrow{r} u \right\} \quad \text{for all } u \text{ in procedure } p
\]

\[
S(p) = \left\{ r \in \text{Edges}^* \mid \text{st}_p \xrightarrow{r} \epsilon \right\} \quad \text{for all procedures } p
\]

\[
R(u) = \left\{ r \in \text{Edges}^* \mid \exists w \in \text{Nodes}^* : \text{st}_{\text{Main}} \xrightarrow{r} uw \right\} \quad \text{for all } u
\]

Same-level runs:

\[
S(p) \supseteq S(r_p) \quad \text{r}_p \text{ return point of } p
\]

\[
S(st_p) \supseteq \{\epsilon\} \quad \text{st}_p \text{ entry point of } p
\]

\[
S(v) \supseteq S(u) \cdot \{\langle e \rangle\} \quad e = (u,s,v) \text{ base edge}
\]

\[
S(v) \supseteq S(u) \cdot S(p) \quad e = (u,p,v) \text{ call edge}
\]

Reaching runs:

\[
R(st_{\text{Main}}) \supseteq \{\epsilon\} \quad \text{st}_{\text{Main}} \text{ entry point of } \text{Main}
\]

\[
R(v) \supseteq R(u) \cdot \{\langle e \rangle\} \quad e = (u,s,v) \text{ basic edge}
\]

\[
R(v) \supseteq R(u) \cdot S(p) \quad e = (u,p,v) \text{ call edge}
\]

\[
R(st_p) \supseteq R(u) \quad e = (u,p,v) \text{ call edge, } st_p \text{ entry point of } p
\]
Context-Sensitive Analysis

Idea:

Phase 1: Compute summary information for each procedure...
... as an abstraction of same-level runs

Phase 2: Use summary information as transfer functions for procedure calls...
... in an abstraction of reaching runs

Classic approaches for summary informations:

1) Functional approach: [Sharir/Pnueli 81, Knoop/Steﬀen: CC´92]
   Use (monotonic) functions on data ﬂow informations!

2) Relational approach: [Cousot/Cousot: POPL´77]
   Use relations (of a representable class) on data ﬂow informations!

3) Call string approach: [Sharir/Pnueli 81], [Khedker/Karkare: CC´08]
   Analyse relative to ﬁnite portion of call stack!
Formalization of Functional Approach

Abstractions:

Abstract same-level runs with $\alpha_{\text{Funct}} : \text{Edges}^* \rightarrow (L \rightarrow L)$:

$$\alpha_{\text{Funct}}(R) = \square \{ f_r \mid r \in R \} \quad \text{for } R \subseteq \text{Edges}^*$$

Abstract reaching runs with $\alpha_{\text{MOP}} : \text{Edges}^* \rightarrow L$:

$$\alpha_{\text{MOP}}(R) = \square \{ f_r(\text{init}) \mid r \in R \} \quad \text{for } R \subseteq \text{Edges}^*$$

1. Phase: Compute summary informations, i.e., functions:

$$S^#(p) \sqsubseteq S^#(r_p) \quad r_p \text{ return point of } p$$
$$S^#(st_p) \sqsubseteq \text{id} \quad st_p \text{ entry point of } p$$
$$S^#(v) \sqsubseteq f_e^# \circ S^#(u) \quad e = (u,s,v) \text{ base edge}$$
$$S^#(v) \sqsubseteq S^#(p) \circ S^#(u) \quad e = (u,p,v) \text{ call edge}$$

2. Phase: Use summary informations; compute on data flow informations:

$$R^#(st_{\text{Main}}) \sqsubseteq \text{init} \quad st_{\text{Main}} \text{ entry point of } \text{Main}$$
$$R^#(v) \sqsubseteq f_e^#(R^#(u)) \quad e = (u,s,v) \text{ basic edge}$$
$$R^#(v) \sqsubseteq S^#(p)(R^#(u)) \quad e = (u,p,v) \text{ call edge}$$
$$R^#(st_p) \sqsubseteq R^#(u) \quad e = (u,p,v) \text{ call edge, } st_p \text{ entry point of } p$$
Functional Approach

Theorem:

Correctness: For any monotone framework:
\[ \alpha_{\text{MOP}}(R[u]) \subseteq R^#u \quad \text{f.a. } u \]

Completeness: For any universally-distributive framework:
\[ \alpha_{\text{MOP}}(R[u]) = R^#u \quad \text{f.a. } u \]

Alternative condition:
framework positively-distributive & all prog. point dyn. reachable

Remark:

a) Functional approach is effective, if \( L \) is finite...

b) ... but may lead to chains of length up to \( |L| \cdot \text{height}(L) \) at each program point (in general).
Observations:

Just three monotone functions on lattice $L$:

- $\lambda x. \text{false}$
- $\lambda x. x$
- $\lambda x. \text{true}$

$\text{false}$

$\text{true}$

Functional composition of two such functions $f, g : L \rightarrow L$:

$$h \circ f = \begin{cases} f & \text{if } h = i \\ h & \text{if } h \in \{ g, k \} \end{cases}$$

Analogous: precise interprocedural analysis for all (separable) bitvector problems in time linear in program size.
Context-Sensitive Analysis, 1. Phase
Context-Sensitive Analysis, 2. Phase

The lattice:

Main:
- P(): true
- Q(): true
- R(): false

P():
- true
- false

Q():
- true
- false

R():
- true
- false

P: i
- true
- false

Q: k
- true
- false

R: g
- true
- false

the lattice:
- false
- true
**Functional Approach**

**Theorem:**

**Correctness:** For any monotone framework:
\[ \alpha_{MOP}(R[u]) \subseteq R^\#[u] \quad \text{f.a. } u \]

**Completeness:** For any universally-distributive framework:
\[ \alpha_{MOP}(R[u]) = R^\#[u] \quad \text{f.a. } u \]

*Alternative condition:* framework positively-distributive & all prog. point dyn. reachable

**Remark:**

a) Functional approach is **effective**, if \( L \) is finite ...

b) ... but may lead to **chains of length up to** \(|L| \cdot \text{height}(L)\) at each program point.
Overview

- Introduction
- Fundamentals of Program Analysis
  Excursion 1

- Interprocedural Analysis
  Excursion 2

- Analysis of Parallel Programs
  Excursion 3
  Appendix

- Conclusion
Precise Interprocedural Analysis through Linear Algebra

Markus Müller-Olm
FernUniversität Hagen
(on leave from Universität Dortmund)

Joint work with
Helmut Seidl (TU München)

POPL 2004, Venice, January 14-16, 2004
Finding Invariants...

Main:

0

1

x₁ := x₂

2

x₃ := 0

3

x₁ - x₂ - x₃ = 0

P()

4

x₁ := x₁ - x₂ - x₃

P()

5

x₁ - x₂ - x₃ - x₂x₃ = 0

P:

6

x₃ := x₃ + 1

7

x₁ := x₁ + x₂ + 1

8

P()

9

P()

x₁ := x₁ - x₂

x₁ := x₁ - x₂ - x₃ - x₂x₃ = 0
... through Linear Algebra

- Linear Algebra
  - vectors
  - vector spaces, sub-spaces, bases
  - linear maps, matrices
  - vector spaces of matrices
  - Gaussian elimination
  - ...

Applications

- definite equalities: \( x = y \)
- constant propagation: \( x = 42 \)
- discovery of symbolic constants: \( x = 5yz+17 \)
- complex common subexpressions: \( xy+42 = y^2+5 \)
- loop induction variables
- program verification
- ...
A Program Abstraction

Affine programs:

- affine assignments: \( x_1 := x_1 - 2x_3 + 7 \)
- unknown assignments: \( x_i := ? \)  
  \( \rightarrow \) abstract too complex statements!
- non-deterministic instead of guarded branching
The Challenge

Given an affine program
(with procedures, parameters, local and global variables, ...)
over $R$:
($R$ the field $\mathbb{Q}$ or $\mathbb{Z}_p$, a modular ring $\mathbb{Z}_m$, the ring of integers $\mathbb{Z}$, an effective PIR,...)

- determine all valid affine relations:
  \[ a_0 + \sum a_i x_i = 0 \quad a_i \in R \]
  \[ 5x+7y-42=0 \]

- determine all valid polynomial relations (of degree $\leq d$):
  \[ p(x_1,\ldots,x_k) = 0 \quad p \in R[x_1,\ldots,x_n] \]
  \[ 5xy^2+7z^3-42=0 \]

... and all this in polynomial time (unit cost measure) !!!
Infinity Dimensions

push-down

arithmetic
Use a Standard Approach for Interprocedural Generalization of Karr?

**Functional approach**  [Sharir/Pnueli, 1981], [Knoop/Steffen, 1992]
- Idea: summarize each procedure by function on data flow facts
- Problem: not applicable

**Call-string approach**  [Sharir/Pnueli, 1981], [Khedker/Karkare: CC´08]
- Idea: take just a finite piece of run-time stack into account
- Problem: not exact

**Relational approach**  [Cousot/Cousot, 1977]
- Idea: summarize each procedure by approximation of I/O relation
- Problem: not exact
Towards the Algorithm ...
Concrete Semantics of an Execution Path

Every execution path \( \pi \) induces an **affine transformation** of the program state:

\[
\begin{bmatrix}
  x_1 := x_1 + x_2 + 1; \\
  x_3 := x_3 + 1
\end{bmatrix}(\nu)
\]

\[
= \begin{bmatrix}
  x_3 := x_3 + 1
\end{bmatrix}(\begin{bmatrix}
  x_1 := x_1 + x_2 + 1
\end{bmatrix}(\nu))
\]

\[
= \begin{bmatrix}
  x_3 := x_3 + 1
\end{bmatrix}\begin{bmatrix}
  1 & 1 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
  \nu_1 \\
  \nu_2 \\
  \nu_3
\end{bmatrix} + \begin{bmatrix}
  1 \\
  0 \\
  0
\end{bmatrix}
\]

\[
= \begin{bmatrix}
  1 & 1 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
  \nu_1 \\
  \nu_2 \\
  \nu_3
\end{bmatrix} + \begin{bmatrix}
  1 \\
  0 \\
  0
\end{bmatrix}
\]
Affine Relations

- An affine relation can be viewed as a vector:

\[ x_1 - 3x_2 + 5 = 0 \] corresponds to \[ a = \begin{pmatrix} 5 \\ 1 \\ 3 \\ 0 \end{pmatrix} \]
Affine Assignments induce linear wp-Transformations on Affine Relations

\[ \{x_2 + x_3 + 5 = 0\} \quad x_1 := 4x_2 + x_3 + 3 \quad \{x_1 - 3x_2 + 2 = 0\} \]

\[ \begin{align*}
\begin{bmatrix}
1 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 4 & 1 & 0 \\
0 & 1 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
2 \\
1 \\
-3 \\
0 \\
\end{bmatrix}
= 
\begin{bmatrix}
5 \\
0 \\
1 \\
1 \\
\end{bmatrix}
\end{align*} \]

A linear transformation:

weakest precondition!
WP of Affine Relations

- Every execution path $\pi$ induces a linear transformation of affine post-conditions into their weakest pre-conditions:

\[
\left[ x_1 := x_1 + x_2 + 1; x_3 := x_3 + 1 \right] (a)
\]

\[= \left[ x_1 := x_1 + x_2 + 1 \right] \left( \left[ x_3 := x_3 + 1 \right] (a) \right)\]

\[= \left[ x_1 := x_1 + x_2 + 1 \right] \begin{pmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3
\end{pmatrix}
\]

\[= \begin{pmatrix}
1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3
\end{pmatrix}
\]
Observations

- Only the zero relation is valid at program start:
  \[ 0 : 0+0x_1+\ldots+0x_k = 0 \]

- Thus, relation \( a_0+a_1x_1+\ldots+a_kx_k=0 \) is valid at program point \( v \)
  iff
  \[ M a = 0 \quad \text{for all } M \in \{[[\pi]^T \mid \pi \text{ reaches } v\} \]
  iff
  \[ M a = 0 \quad \text{for all } M \in \text{Span } \{[[\pi]^T \mid \pi \text{ reaches } v\} \]
  iff
  \[ M a = 0 \quad \text{for all } M \text{ in a basis of } \text{Span } \{[[\pi]^T \mid \pi \text{ reaches } v\} \]

- Matrices \( M \) form a vector space of dimension \((k+1) \times (k+1)\)

- Sub-spaces form a complete lattice of height \( O(k^2) \).
Let's Apply Our Abstract Interpretation Recipe: Constraint System for Feasible Paths

Operational justification:

\[ S(u) = \{ r \in \text{Edges}^* \mid st_p \xrightarrow{r} u \} \quad \text{for all } u \text{ in procedure } p \]
\[ S(p) = \{ r \in \text{Edges}^* \mid st_p \xrightarrow{r} \varepsilon \} \quad \text{for all procedures } p \]
\[ R(u) = \{ r \in \text{Edges}^* \mid \exists \omega \in \text{Nodes}^* : st_{\text{Main}} \xrightarrow{r} u\omega \} \quad \text{for all } u \]

Same-level runs:

\[ S(p) \supseteq S(r_p) \quad r_p \text{ return point of } p \]
\[ S(st_p) \supseteq \{\varepsilon\} \quad st_p \text{ entry point of } p \]
\[ S(v) \supseteq S(u) \cdot \{\langle e\rangle\} \quad e = (u,s,v) \text{ base edge} \]
\[ S(v) \supseteq S(u) \cdot S(p) \quad e = (u,p,v) \text{ call edge} \]

Reaching runs:

\[ R(st_{\text{Main}}) \supseteq \{\varepsilon\} \quad st_{\text{Main}} \text{ entry point of } \text{Main} \]
\[ R(v) \supseteq R(u) \cdot \{\langle e\rangle\} \quad e = (u,s,v) \text{ basic edge} \]
\[ R(v) \supseteq R(u) \cdot S(p) \quad e = (u,p,v) \text{ call edge} \]
\[ R(st_p) \supseteq R(u) \quad e = (u,p,v) \text{ call edge, } st_p \text{ entry point of } p \]
Algorithm for Computing Affine Relations

1) Compute a basis $B$ with:
   \[ \text{Span } B = \text{Span } \{ [[\pi]]^T \mid \pi \text{ reaches } v \} \]
   for each program point by a precise abstract interpretation:

   Lattice: Subspaces of $\text{IF}^{(k+1) \times (k+1)}$

   Replace:

   \[
   \begin{align*}
   \{ \varepsilon \} & \quad \text{by} \quad \{ I \} \quad (I \text{ identity matrix}) \\
   \text{concatenation} & \quad \text{by} \quad \text{matrix product} \quad (\text{lifted to subspaces}) \\
   \{ \langle e \rangle \} & \quad \text{by} \quad \langle A_e \rangle \text{ for affine assignment edge } e = (u,s,v)
   \end{align*}
   \]

2) Solve the linear equation system:
   \[ M a = 0 \quad \text{for all } M \in B \]
Theorem

In an affine program:

- The following vector spaces of matrices can be computed precisely:
  \[ \alpha(R(v)) = \text{Span} \{ [\pi]^T \mid \pi \in R(v) \} \] for each prg. point \( v \).

- The vector spaces
  \[ \{ a \in \mathbb{F}^{k+1} \mid \text{affine relation } a \text{ is valid at } v \} \]
  can be computed precisely for all prg. points \( v \).

- The time complexity is \textbf{linear} in the program size and \textbf{polynomial} in the number of variables: \( O(n \cdot k^3) \)
  \( (n \text{ size of the program, } k \text{ number of variables}) \)
An Example

Main:

0

$x_1 := x_2$

1

$x_3 := 0$

2

$P()$

3

$x_1 := x_1 - x_2 - x_3$

4

P:

0

$x_3 := x_3 + 1$

1

$x_1 := x_1 + x_2 + 1$

2

$P()$

3

$x_1 := x_1 - x_2$

4

$⇒$ stable!

P:

$\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}$

$\begin{bmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}$

$\begin{bmatrix}
1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}$

$\begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}$

$\begin{bmatrix}
1 & 2 & 0 & 2 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}$

$\begin{bmatrix}
1 & 2 & 0 & 2 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}$
An Example

\[ a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 = 0 \] is valid at 3

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 
\end{bmatrix} = 0 \quad \text{and} \quad
\begin{bmatrix}
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 
\end{bmatrix} = 0
\]

\[ a_0 = 0 \land a_2 = a_3 = -a_1 \]

Just the affine relations of the form

\[ a_1 x_1 - a_1 x_2 - a_1 x_3 = 0 \quad (a_1 \in \mathbb{F}) \]

are valid at 3
Extensions

- Also in the paper:
  - Local variables, value parameters, return values
  - Computing polynomial relations of bounded degree
  - Affine pre-conditions
  - Formalization as an abstract interpretation

- In follow-up papers (see webpage):
  - Computing over modular rings (e.g. modulo $2^w$) or PIRs
  - Forward algorithm
End of Excursion 2
Overview

- Introduction
- Fundamentals of Program Analysis
  Excursion 1

- Interprocedural Analysis
  Excursion 2

- Analysis of Parallel Programs
  Excursion 3
  Appendix

- Conclusion
Interprocedural Analysis of Parallel Programs

Main:
- \( \text{c:=a+b} \)

P: \( \text{c:=a+b} \)

Q: \( \text{a:=7} \) \( \text{c:=a+b} \)

R: \( \text{a:=7} \) \( \text{c:=a+b} \)

parallel call edge
Interleaving- Operator $\otimes$
(Shuffle-Operator)

Example:

$$\langle a, b \rangle \otimes \langle x, y \rangle = \left\{ \begin{array}{l}
\langle a, b, x, y \rangle \\
\langle a, x, b, y \rangle, \langle a, x, y, b \rangle \\
\langle x, a, b, y \rangle, \langle x, a, y, b \rangle, \langle x, y, a, b \rangle \\
\end{array} \right. $$
Constraint System for Same-Level Runs

Operational justification:

\[
\begin{align*}
S(u) &= \{ r \in \text{Edges}^* \mid st_p \xrightarrow{r} u \} \quad \text{for all } u \text{ in procedure } p \\
S(p) &= \{ r \in \text{Edges}^* \mid st_p \xrightarrow{r} \varepsilon \} \quad \text{for all procedures } p
\end{align*}
\]

Same-level runs:

\[
\begin{align*}
S(p) &\supseteq S(r_p) & r_p &\text{ return point of } p \\
S(st_p) &\supseteq \{ \varepsilon \} & st_p &\text{ entry point of } p \\
S(v) &\supseteq S(u) \cdot \langle \{ e \} \rangle & e = (u, s, v) &\text{ base edge} \\
S(v) &\supseteq S(u) \cdot S(p) & e = (u, p, v) &\text{ call edge} \\
S(v) &\supseteq S(u) \cdot (S(p_0) \otimes S(p_1)) & e = (u, p_0 \parallel p_1, v) &\text{ parallel call edge}
\end{align*}
\]

[Seidl/Steffen: ESOP 2000]
Constraint System for a Variant of Reaching Runs

Operational justification:

\[ R(u,q) = \{ r \in \text{Edges}^* \mid \exists c \in \text{Config} : st_q \xrightarrow{r} c, \text{At}_u(c) \} \]

for program point \( u \) and procedure \( q \)

\[ P(q) = \{ r \in \text{Edges}^* \mid \exists c \in \text{Config} : st_q \xrightarrow{r} c \} \]

Reaching runs:

\[ R(u,q) \supseteq S(u) \quad u \text{ program point in procedure } q \]
\[ R(u,q) \supseteq S(v) \cdot R(u,p) \quad e = (v,p,\_\_\_) \text{ call edge in proc. } q \]
\[ R(u,q) \supseteq S(v) \cdot (R(u,p_i) \otimes P(p_{i-1})) \quad e = (v,p_0 \parallel p_{i-1} \_\_\_) \text{ parallel call edge in proc. } q, \ i = 0,1 \]

Interleaving potential:

\[ P(p) \supseteq R(u,p) \quad u \text{ program point and } p \text{ procedure} \]
Interleaving- Operator $\otimes$
(Shuffle-Operator)

Example:

$$\langle a, b \rangle \otimes \langle x, y \rangle = \left\{ \begin{array}{l}
\langle a, b, x, y \rangle \\
\langle a, x, b, y \rangle, \langle a, x, y, b \rangle \\
\langle x, a, b, y \rangle, \langle x, a, y, b \rangle, \langle x, y, a, b \rangle \\
\end{array} \right.$$

The only new ingredient:

interleaving operator $\otimes$ must be abstracted!
Case: Availability of Single Expression

![Diagram showing abstract shuffle operator and lattice]

Abstract shuffle operator:

<table>
<thead>
<tr>
<th>⊗#</th>
<th>i</th>
<th>g</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>i</td>
<td>g</td>
<td>k</td>
</tr>
<tr>
<td>g</td>
<td>g</td>
<td>g</td>
<td>k</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
</tr>
</tbody>
</table>

Main lemma:

∀ f_j ∈ {g, k, i} :  
\[
\forall j \in \{i\} : \left( f_n \circ \ldots \circ f_{j+1} \circ f_j \right) = f_j \\
\forall j \not\in \{i\} : f_1 = f_j
\]

Treat other (separable) bitvector problems analogously...

⇒ precise interprocedural analyses for all bitvector problems !
Overview

- Introduction
- Fundamentals of Program Analysis
  Excursion 1
- Interprocedural Analysis
  Excursion 2
- Analysis of Parallel Programs
  Excursion 3
  Appendix
- Conclusion
Precise Fixpoint-Based Analysis of Programs with Thread-Creation and Procedures

Markus Müller-Olm
Westfälische Wilhelms-Universität Münster

Joint work with:
Peter Lammich
[same place]

CONCUR 2007
(My) Main Interests of Recent Years

Data aspects
- algebraic invariants over $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{Z}_m$ ($m = 2^n$) in sequential programs, partly with recursive procedures
- invariant generation relative to Herbrand interpretation

Control aspects
- recursion
- concurrency with process creation / threads
- synchronization primitives, in particular locks/monitors

Technics used
- fixpoint-based
- automata-based
- (linear) algebra
- syntactic substitution-based techniques
- ...
Another Program Model

Procedures

Recursive procedure calls

Basic actions

Spawn commands

P: A
B

call P

spawn Q

Q:
C
D
E

Entry point, \( e_q \), of Q

Return point, \( x_q \), of Q

0
1
2
3
4
5
6
7
Spawns are Fundamentally Different

P induces trace language: \( L = \bigcup \{ A^n \cdot (B^m \otimes (C^i \cdot D^j)) \mid n \geq m \geq 0, i \geq j \geq 0 \} \)

Cannot characterize \( L \) by constraint system with "⋅" and "⊗".

[Bouajjani, MO, Touili: CONCUR 2005]
Gen/Kill-Problems

- Class of simple but important DFA problems

- Assumptions:
  - Lattice \((L,\sqsubseteq)\) is distributive
  - Transfer functions have form \(f_e(l) = (l \sqcap \text{kill}_e) \sqcup \text{gen}_e\) with \(\text{kill}, \text{gen} \in L\)

- Examples:
  - bitvector problems, e.g.
  - available expressions, live variables, very busy expressions, ...
Data Flow Analysis

Goal:
Compute, for each program point u:
- Forward analysis: $\text{MOP}^F[u] = \alpha^F(\text{Reach}[u])$, where $\alpha^F(X) = \bigcup \{ f_w(x_0) \mid w \in X \}$
- Backward analysis: $\text{MOP}^B[u] = \alpha^B(\text{Leave}[u])$, where $\alpha^B(X) = \bigcup \{ f_w(\perp) \mid w^R \in X \}$

\[
\begin{align*}
\text{Reach}[u] &= \{ w \mid \exists c : \{ [e_{Main}] \xrightarrow{w} c \land at_u(c) \} \\
\text{Leave}[u] &= \{ w \mid \exists c : \{ [e_{Main}] \xrightarrow{*} c \xrightarrow{w} \perp \land at_u(c) \} \\
at_u(c) &\iff \exists w : (uw) \in c \\
f_w &= f_{e_n} \circ \cdots \circ f_{e_1}, \text{ for } w = e_1 \cdots e_n
\end{align*}
\]
Data Flow Analysis

Goal:
Compute, for each program point \( u \):
- Forward analysis: \( \text{MOP}^F[u] = \alpha^F(\text{Reach}[u]) \), where \( \alpha^F(X) = \bigcup \{ f_w(x_0) \mid w \in X \} \)
- Backward analysis: \( \text{MOP}^B[u] = \alpha^B(\text{Leave}[u]) \), where \( \alpha^B(X) = \bigcup \{ f_w(\bot) \mid w^R \in X \} \)

Problem for programs with threads and procedures:
We cannot characterize \( \text{Reach}[u] \) and \( \text{Leave}[u] \) by a constraint system with operators „concatenation“ and „interleaving“.
One Way Out

- Derive alternative characterization of MOP-solution:
  - reason on level of execution paths
  - exploit properties of gen/kill-problems

- Characterize the path sets occurring as least solutions of constraint systems

- Perform analysis by abstract interpretation of these constraint systems

[Lammich/MO: CONCUR 2007]
Forward Analysis
Directly Reaching Paths and Potential Interleaving

**Reaching path:** a suitable interleaving of the red and blue paths

**Directly reaching path:** the red path

**Potential interference:** set of edges in the blue paths (note: no order information!)

Formalization by augmented operational semantics with markers (see paper)
**Forward MOP-solution**

**Theorem:** For gen/kill problems:

\[ \text{MOP}^F[u] = \alpha^F(\text{DReach}[u]) \sqcup \alpha^\text{Pl}(\text{Pl}[u]), \]

where \( \alpha^\text{Pl}(X) = \sqcup \{ \text{gen}_e \mid e \in X \} \).

**Remark**

- \( \text{DReach}[u] \) and \( \text{Pl}[u] \) can be characterized by constraint systems (see paper)

- \( \alpha^F(\text{DReach}[u]) \) and \( \alpha^\text{Pl}(\text{Pl}[u]) \) can be computed by an abstract interpretation of these constraint systems
Characterizing Directly Reaching Paths

Same level paths:

\[
\text{[init]} \quad S[e_q] \supseteq \{\varepsilon\} \quad \text{for } q \in P \\
\text{[base]} \quad S[v] \supseteq S[u] ; e \quad \text{for } e = (u, \text{base } _, v) \in E \\
\text{[call]} \quad S[v] \supseteq S[u] ; e ; S[r_q] ; \text{ret} \quad \text{for } e = (u, \text{call } q, v) \in E \\
\text{[spawn]} \quad S[v] \supseteq S[u] ; e \quad \text{for } e = (u, \text{spawn } q, v) \in E
\]

Directly reaching paths:

\[
\text{[init]} \quad R[e_{\text{main}}] \supseteq \{\varepsilon\} \\
\text{[reach]} \quad R[u] \supseteq R[e_p] ; S[u] \quad \text{for } u \in N_p \\
\text{[callp]} \quad R[e_q] \supseteq R[u] ; e \quad \text{for } e = (u, \text{call } q, _) \in E \\
\text{[spawnp]} \quad R[e_q] \supseteq R[u] ; e \quad \text{for } e = (u, \text{spawn } q, _) \in E
\]
Backwards Analysis
Directly Leaving Paths and Potential Interleaving

Leaving path: a suitable interleaving of orange, black and parts of blue paths

Directly leaving path: a suitable interleaving of orange and black paths

Potential interference: the edges in the blue paths

Formalization by augmented operational semantics with markers (see paper)
Interleaving from Threads created in the Past

Theorem: For gen/kill problems:

\[ \text{MOP}^B[u] = \alpha^B(\text{DLeave}[u]) \sqcup \alpha^{\text{PI}}(\text{PI}[u]), \]

where \( \alpha^{\text{PI}}(E) = \sqcup \{ \text{gen}_e | e \in E \} \).

Remark

- We know no simple characterization of DLeave[u] by a constraint system.

- Main problem: Threads generated in a procedure instance survive that instance.
Representative Directly Leaving Paths

A representative directly leaving path:

1 2 3 4 5

at u
Interleaving from Threads created in the Future

Lemma
\[ \alpha^B(D\text{Leave}[u]) = \alpha^B(RD\text{Leave}[u]) \] (for gen/kill problems).

Corollary
\[ \text{MOP}^B[u] = \alpha^B(RD\text{Leave}[u]) \sqcup \alpha^\text{PI}(P\text{I}[u]) \] (for gen/kill problems).

Remark
- RD\text{Leave}[u] and P\text{I}[u] can be characterized by constraint systems (see paper)
- \[ \alpha^B(RD\text{Leave}[u]) \] and \[ \alpha^\text{PI}(P\text{I}[u]) \] can be computed by an abstract interpretation of these constraint systems
Also in the Paper

- Formalization of these ideas
  - constraint systems for path sets
  - validation with respect to operational semantics

- Parallel calls in combination with threads
  - threads become trees instead of stacks ...

- Analysis of running time:
  - global information in time linear in the program size
Summary

- Forward- and backward gen/kill-analysis for programs with threads and procedures
- More efficient than automata-based approach
- More general than known fixpoint-based approach
- **Current work**: Precise analysis in presence of locks/monitors (see papers at SAS 2008, CAV 2009 for first results)
End of Excursion 3
Appendix

Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems
DPNs: Dynamic Pushdown-Networks

A *dynamic pushdown-network* (over a finite set of actions Act) consists of:

- $P$, a finite set of control symbols
- $\Gamma$, a finite set of stack symbols
- $\Delta$, a finite set of rules of the following form

$$
\begin{align*}
p\gamma & \xrightarrow{a} p_1 w_1 \\
p\gamma & \xrightarrow{a} p_1 w_1 \triangleright p_2 w_2 
\end{align*}
$$

(with $p, p_1, p_2 \in P$, $\gamma \in \Gamma$, $w_1, w_2 \in \Gamma^*$, $a \in \text{Act}$).
DPNs: Dynamic Pushdown-Networks

A **State** of a DPN is a word in \((P\Gamma^*)^+\):

\[ p_1w_1p_2w_2\ldots p_kw_k \quad \text{(with } p_i \in P, w_i \in \Gamma^*, k > 0) \]

... an infinite state space

The transition relation of a DPN:

\[
\begin{align*}
(p, \gamma \xrightarrow{a} p_1w_1) &\in \Delta: & u p \gamma v \xrightarrow{a} u p_1 w_1 v \\
(p, \gamma \xrightarrow{a} p_1w_1 \triangleright p_2w_2) &\in \Delta: & u p \gamma v \xrightarrow{a} u p_2 w_2 p_1 w_1 v
\end{align*}
\]
Example

Consider the following DPN with a single rule

\[ p\gamma \xrightarrow{a} p\gamma \delta q\gamma \]

Transitions:

\[ p\gamma \]
\[ q\gamma d\gamma \]
\[ q\gamma q\gamma d\gamma \]
\[ q\gamma q\gamma q\gamma d\gamma \]
\[ q\gamma q\gamma q\gamma q\gamma d\gamma \]
\[ \vdots \]
\[ \vdots \]
Reachability Analysis

Given:
- Model of a system: M
- Set of system states: Bad

Reachability analysis:
- Can a state from Bad be reached from an initial states of the system?
  \[ \exists \sigma_0, ..., \sigma_k : \text{Init} \ni \sigma_0 \rightarrow \cdots \rightarrow \sigma_k \in \text{Bad} \]

Applications:
- Check safety properties:
  Bad is a set of states to be avoided
- More applications by iterated computation of reachability sets for sub-models of the system model, e.g. data-flow analysis...
Reachability Analysis

Given:

- Model of a system: M
- Set of system states: Bad

Reachability analysis:

- Can a state from Bad be reached from an initial state of the system?
  \( \exists \sigma_0, \ldots, \sigma_k : \text{Init} \ni \sigma_0 \rightarrow \cdots \rightarrow \sigma_k \in \text{Bad} \)

Def.: - \( \text{pre}^*(X) =_{\text{df}} \{ \sigma | \exists \sigma' \in X : \sigma \rightarrow^* \sigma' \} \)
  - \( \text{post}^*(X) =_{\text{df}} \{ \sigma | \exists \sigma' \in X : \sigma' \rightarrow^* \sigma \} \)

Equivalent formulations of reachability analysis:

- \( \text{pre}^*(\text{Bad}) \cap \text{Init} \neq \emptyset \)
- \( \text{post}^*(\text{Init}) \cap \text{Bad} \neq \emptyset \)

\( \Rightarrow \) Computation of \( \text{pre}^* \) or \( \text{post}^* \) is key to reachability analysis
Reachability Analysis of Finite State Systems

\[ \varphi_0 = \text{Init} \]
\[ \varphi_{i+1} \overset{\text{def}}{=} \varphi_i \cup \text{post}(\varphi_i) \]
\[ \text{post}(X) \overset{\text{def}}{=} \{ \sigma \mid \exists \sigma' \in X : \sigma \rightarrow \sigma' \} \]

\[ \Rightarrow \quad \text{Bad reachable from initial state} \]
Reachability Analysis of Finite State Systems

\[ \varphi_0 = \text{Init} \]

\[ \varphi_{i+1} = \varphi_i \cup \text{post} (\varphi_i) \]

\[ \text{post}(X) = \{ \sigma | \exists \sigma' \in X : \sigma \rightarrow \sigma' \} \]

\( \Rightarrow \) Bad not reachable from initial state
Problems with Infinite-State Systems

- State sets $\varphi_i$ can be infinite

  $\Rightarrow$ symbolic representation of (certain) infinite state sets

  Here: by finite automata
Example: Representation of an Infinite State Set of a DPN by a Word Automaton

An automaton A:

The *regular* set of states represented by A:

\[ L(A) = (q\gamma q\gamma \gamma^*)^* \]

... an infinite set of states.
Problems with Infinite-State Systems

- State sets $\varphi_i$ can be infinite
  - symbolic representation of (certain) infinite state sets
    - Here: by finite (word) automata

- Iterated computation of reachability sets does not terminate in general
  - Methods for *acceleration* of the computation
    - Here: by computing with finite automata
Computing pre* for DPNs with Finite Automata

Theorem [Bouajjani, MO, Touili, 2005]

For every DPN and every regular state set R,
pre*(R) is regular and can be computed in polynomial time.

Proof:

Generalization of a known technique for single pushdown systems:
saturation of an automaton for R.

⇒ Reachability analysis is effective for regular sets Bad of states!
Example: Reachability Analysis for DPNs

Consider again DPN with the rule

\[ pγ \xrightarrow{a} pγγ \triangleright qγ \]

and the infinite set of states

\[ \text{Bad} = (qγqγpγ^*)^* = L(A) \]

Analysis problem: can \( \text{Bad} \) be reached from \( pγ \)?
Example: Reachability Analysis for DPNs

1. **Step**: Saturate automaton for Bad with the DPN rule:

   \[ p\gamma \rightarrow^a p\gamma > q\gamma \]

   Resulting automaton \( A_{\text{pre}^*} \) represents \( \text{pre}^*(\text{Bad}) \)!

2. **Step**: Check, whether \( p\gamma \) is accepted by \( A_{\text{pre}^*} \) or not

   **Result**: Bad is reachable from \( p\gamma \), as \( A_{\text{pre}^*} \) accepts \( p\gamma \).
Modelling Programs with Procedures and Threads by DPNs

Main:

1. $x := x + 1$
2. call Main
3. $y := 0$
4. spawn Q

Q:

1. $y := x * y$
2. call Q
3. $x := y + 1$

Transition Rules:

- $N_1 \xrightarrow{x := x + 1} N_2$
- $N_2 \xrightarrow{\text{call}_p} N_1 N_3$
- $N_3 \xrightarrow{y := 0} N_4$
- $N_1 \xrightarrow{\text{spawn}_Q} N_4 \triangleright M_1$
- $M_1 \xrightarrow{y := x * y} M_2$
- $M_2 \xrightarrow{\text{call}_Q} M_1 M_3$
- $M_3 \xrightarrow{x := y + 1} M_4$
- $N_1 \xrightarrow{\text{skip}} M_4$
Live Variables Analysis
via
Iterated $\text{pre}[^*]$-computation

Observation

Variable $x$ is *live* at $u$

iff

$$e_{\text{Main}} \in \text{pre}^*(A_t_u \cap \text{pre}_{\Delta_{\text{non-def}}}^* (\text{pre}_{\Delta_{\text{use}}} (\text{Conf})))$$

Remark

This condition can be checked by computing with automata

Esparza, Knoop
Steffen, Schmidt
A Non-Representability Result

- P induces trace language: \( L = \bigcup \{ A^n \cdot (B^m \otimes (C^i \cdot D^j)) | n \geq m \geq 0, i \geq j \geq 0 \} \)
- L cannot be characterized by constraint system with operators „concatenation“ and „interleaving“
Observation [Bouajjani, MO, Touili, 2005]
In general, post*(R) is not regular, not even if R is finite.

Example:
Consider DPN with the rule \( p\gamma \xrightarrow{a} p\gamma \gamma > q\gamma \)
Recall:

\[
\begin{align*}
p\gamma & \\
q\gamma & \\
p\gamma p\gamma & \\
p\gamma p\gamma p\gamma & \\
p\gamma p\gamma p\gamma p\gamma & \\
\vdots & \\
p\gamma p\gamma p\gamma p\gamma p\gamma p\gamma & \\
\end{align*}
\]
\[\vdots\]
post*({p\gamma}) = \{ (q\gamma)^kp\gamma^{k+1} \mid k \geq 0 \} \text{ is not regular.}

Theorem [Bouajjani, MO, Touili, 2005]
For every DPN, post*(R) is contextfree if R is contextfree.
It can be computed in polynomial time.
A Little Bit of Synchronization ...

- CDPNs – Constrained Dynamic Pushdown Networks

- Idea: Threads can observe (stable regular patterns of) their children, but not vice versa

- States are represented by trees in order to mirror father/child relationship

- Use tree automata techniques for
  - representation of state sets and
  - symbolic computation of pre* (under certain conditions)

- See the CONCUR 2005 paper

- More recent papers: lock and monitor-sensitive analysis
Comparison of Fixpoint-based and Automata-based Algorithm

Fixpoint-based algorithm: [Lammich/MO: CONCUR 2007]
- computes information for all program points at once in linear time
- can use bitvector operations for computing multiple bits at once

Automata-based algorithm: [Bouajjani/MO/Touili: CONCUR 2005]
- based on pre*-computations of regular sets of configurations
- needs linear time for each program point:
  thus: overall running time is quadractic
- must be iterated for each bit
- more generic w.r.t. sets of configurations
End of Appendix
Conclusion

- Program analysis very broad topic
- Provides generic analysis techniques for (software) systems
- Here just one path through the forest
- Many interesting topics not covered
Thank you!