Introduction to Algebraic Program Analysis

Zachary Kincaid   Thomas Reps

PRINCETON UNIVERSITY   WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

Algebraic program analysis

- A methodology for designing program analyses based on **algebra**.

Algebraic program analysis

- A methodology for designing program analyses based on **algebra**.
- High-level intuition: define analysis by recursion on program syntax

$$
\begin{aligned}
\mathcal{A}[\![-]\!] &: \textit{Program} \to \textit{Summary} \\
\mathcal{A}[\![S_1; S_2]\!] &= \mathcal{A}[\![S_1]\!] \cdot \mathcal{A}[\![S_2]\!] \\
\mathcal{A}[\![\textbf{if}(*)\{S_1\}\textbf{else}\{S_2\}]\!] &= \mathcal{A}[\![S_1]\!] + \mathcal{A}[\![S_2]\!] \\
\mathcal{A}[\![\textbf{while}(*)\{S\}]\!] &= (\mathcal{A}[\![S]\!])^*
\end{aligned}
$$

Algebraic program analysis

- A methodology for designing program analyses based on **algebra**.
- High-level intuition: define analysis by recursion on program syntax

$$
\begin{aligned}
\mathcal{A}[\![-]\!] &: \textit{Program} \rightarrow \textit{Summary} \\
\mathcal{A}[\![S_1; S_2]\!] &= \mathcal{A}[\![S_1]\!] \cdot \mathcal{A}[\![S_2]\!] \\
\mathcal{A}[\![\textbf{if}(*)\{S_1\}\textbf{else}\{S_2\}]\!] &= \mathcal{A}[\![S_1]\!] + \mathcal{A}[\![S_2]\!] \\
\mathcal{A}[\![\textbf{while}(*)\{S\}]\!] &= (\mathcal{A}[\![S]\!])^*
\end{aligned}
$$

- Framework for understanding **compositionality**

Compositional program analysis

- A program analysis is **compositional** if the result for a composite program is a function of the results for its components

$$
\begin{aligned}
\mathcal{A}[\![-]\!] &: \textit{Program} \rightarrow \textit{Summary} \\
\mathcal{A}[\![S_1; S_2]\!] &= \mathcal{A}[\![S_1]\!] \cdot \mathcal{A}[\![S_2]\!] \\
\mathcal{A}[\![\textbf{if}(*)\{S_1\}\textbf{else}\{S_2\}]\!] &= \mathcal{A}[\![S_1]\!] + \mathcal{A}[\![S_2]\!] \\
\mathcal{A}[\![\textbf{while}(*)\{s\}]\!] &= (\mathcal{A}[\![S]\!])^*
\end{aligned}
$$

Compositional program analysis

- A program analysis is **compositional** if the result for a composite program is a function of the results for its components

$$
\begin{aligned}
\mathcal{A}[\![-]\!] &: \textit{Program} \rightarrow \textit{Summary} \\
\mathcal{A}[\![S_1; S_2]\!] &= \mathcal{A}[\![S_1]\!] \cdot \mathcal{A}[\![S_2]\!] \\
\mathcal{A}[\![\textbf{if}(*)\{S_1\}\textbf{else}\{S_2\}]\!] &= \mathcal{A}[\![S_1]\!] + \mathcal{A}[\![S_2]\!] \\
\mathcal{A}[\![\textbf{while}(*)\{s\}]\!] &= (\mathcal{A}[\![S]\!])^*
\end{aligned}
$$

- Benefits:
  - Potential to scale
  - Easy to parallelize
  - Can be applied to incomplete programs (e.g. libraries)
  - Can respond quickly to program edits
  - **Enables new kinds of analysis techniques**
  - ...

Outline

Regular algebraic program analysis

Semantic foundations of algebraic program analysis

Interprocedural analysis

ω-regular program analysis

Algebraic path problems

- Common structure exhibited by several algorithms:
  [Aho et al. ’74, Backhouse & Carré ’75, Lehmann ’77, Tarjan ’81, ...]
  - Kleene’s (NFA $\rightarrow$ regexp) algorithm
  - Warshall’s transitive closure algorithm
  - Floyd’s shortest path algorithm
  - Gauss-Jordan algorithm for solving system of linear equations
  - . . .

Algebraic path problems

- Common structure exhibited by several algorithms:
  [Aho et al. ’74, Backhouse & Carré ’75, Lehmann ’77, Tarjan ’81, ...]
  - Kleene’s (NFA $\rightarrow$ regexp) algorithm
  - Warshall’s transitive closure algorithm
  - Floyd’s shortest path algorithm
  - Gauss-Jordan algorithm for solving system of linear equations
  - $\ldots$
- Algebraic approach to solving path problems in graphs [Tarjan ’81]:
  1. Compute a regular expression recognizing a set of paths of interest
  2. Interpret the regular expression in a suitable algebraic structure

Path expressions

*A **path expression** for a directed graph $G = (V, E)$: regular expression R over the alphabet of edges E such that each word recognized by R corresponds to a path in G.*

Path expressions

*A **path expression** for a directed graph $G = (V, E)$: regular expression R over the alphabet of edges E such that each word recognized by R corresponds to a path in G.*

Regular expression syntax:

$$R \in \text{RegExp}(\Sigma) ::= a \in \Sigma \mid 0 \mid 1 \mid R_1 + R_2 \mid R_1 R_2 \mid R^*$$

Regular expression semantics:

$$\mathscr{L}[\![0]\!] = \emptyset \qquad \mathscr{L}[\![R_1 \cdot R_2]\!] = \{w_1 w_2 : w_1 \in \mathscr{L}[\![R_1]\!], w_2 \in \mathscr{L}[\![R_2]\!]\}$$

$$\mathscr{L}[\![1]\!] = \{\epsilon\} \qquad \mathscr{L}[\![R_1 + R_2]\!] = \mathscr{L}[\![R_1]\!] \cup \mathscr{L}[\![R_2]\!]$$

$$\mathscr{L}[\![a]\!] = \{a\} \quad \text{For } a \in \Sigma \qquad \mathscr{L}[\![R^*]\!] = \mathscr{L}[\![R]\!]^*$$

Regular expression semantics

- An **interpretation** $\mathscr{I}$ consists of a *regular algebra* and a *semantic function*

Regular expression semantics

- An **interpretation** $\mathscr{I}$ consists of a *regular algebra* and a *semantic function*
- A **regular algebra** $\mathbf{A} = \langle A, 0^A, 1^A, +^A, \cdot^A, *^A \rangle$ consists of
  - A set A (the *universe* or *carrier* of the algebra)
  - Distinguished elements $0^A, 1^A \in A$
  - Two binary operators $\cdot^A, +^A : A \times A \to A$ (*sequencing* and *choice*)
  - A unary operator $*^A : A \to A$ (*iteration*)

Regular expression semantics

- An **interpretation** $\mathscr{I}$ consists of a *regular algebra* and a *semantic function*
- A **regular algebra** $\mathbf{A} = \langle A, 0^A, 1^A, +^A, \cdot^A, *^A \rangle$ consists of
  - A set A (the *universe* or *carrier* of the algebra)
  - Distinguished elements $0^A, 1^A \in A$
  - Two binary operators $\cdot^A, +^A : A \times A \to A$ (*sequencing* and *choice*)
  - A unary operator $*^A : A \to A$ (*iteration*)
- A **semantic function** $f : \Sigma \to A$ maps letters of the alphabet into the algebra

Regular expression semantics

- An **interpretation** $\mathscr{I}$ consists of a *regular algebra* and a *semantic function*
- A **regular algebra** $\mathbf{A} = \langle A, 0^A, 1^A, +^A, \cdot^A, *^A \rangle$ consists of
  - A set A (the *universe* or *carrier* of the algebra)
  - Distinguished elements $0^A, 1^A \in A$
  - Two binary operators $\cdot^A, +^A : A \times A \to A$ (*sequencing* and *choice*)
  - A unary operator $*^A : A \to A$ (*iteration*)
- A **semantic function** $f : \Sigma \to A$ maps letters of the alphabet into the algebra
- Define interpretation $\mathscr{I}[\![-]\!]$: RegExp$(\Sigma) \to A$:

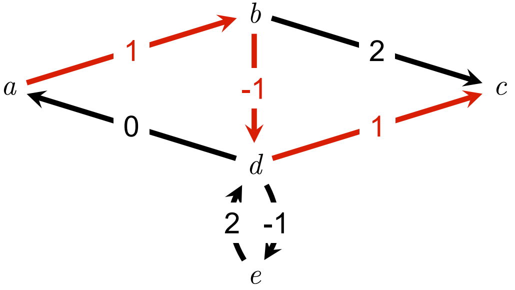
$$\mathscr{I}[\![0]\!] = 0^A \qquad \mathscr{I}[\![R_1 \cdot R_2]\!] = \mathscr{I}[\![R_1]\!] \cdot^A \mathscr{I}[\![R_2]\!]$$

$$\mathscr{I}[\![1]\!] = 1^A \qquad \mathscr{I}[\![R_1 + R_2]\!] = \mathscr{I}[\![R_1]\!] +^A \mathscr{I}[\![R_2]\!]$$

$$\mathscr{I}[\![a]\!] = f(a) \quad \text{For } a \in \Sigma \qquad \mathscr{I}[\![R^*]\!] = \mathscr{I}[\![R]\!]^{*^A}$$

Warm-up: shortest paths

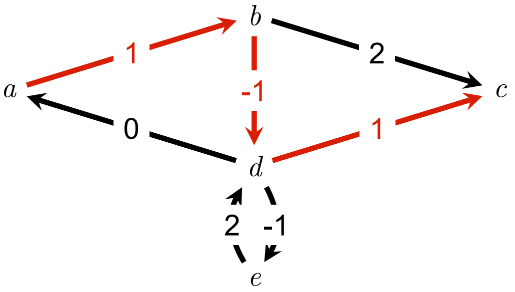
- Consider an edge-weighted graph:



- Suppose we want to compute smallest-weight path from a to c

Warm-up: shortest paths

- Consider an edge-weighted graph:

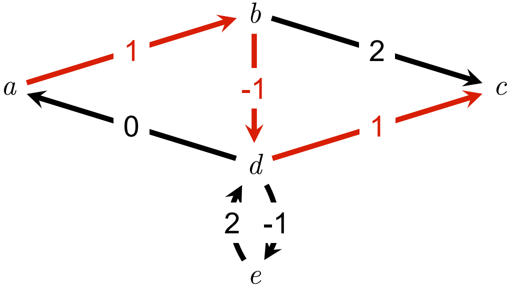


- Suppose we want to compute smallest-weight path from a to c
  1. Compute a path expression recognizing paths from a to c

$$\left(\langle a, b\rangle\langle b, d\rangle \left(\langle d, e\rangle\langle e, d\rangle\right)^* \langle d, a\rangle\right)^* \langle a, b\rangle \left(\langle b, c\rangle + \langle b, d\rangle \left(\langle d, e\rangle\langle e, d\rangle\right)^* \langle d, c\rangle\right)$$

Warm-up: shortest paths

- Consider an edge-weighted graph:



- Suppose we want to compute smallest-weight path from a to c
  1. Compute a path expression recognizing paths from a to c
     $$\left(\langle a, b\rangle\langle b, d\rangle\left(\langle d, e\rangle\langle e, d\rangle\right)^* \langle d, a\rangle\right)^* \langle a, b\rangle \left(\langle b, c\rangle + \langle b, d\rangle\left(\langle d, e\rangle\langle e, d\rangle\right)^* \langle d, c\rangle\right)$$
  2. Interpret the path expression within a *distance algebra*

Algebra of distances

- Distance algebra universe: $\mathbb{Z} \cup \{-\infty, \infty\}$
- Operations:

$$0^D = \infty$$

$$1^D = 0$$

$$d_1 +^D d_2 \triangleq \min(d_1, d_2)$$ Minimum

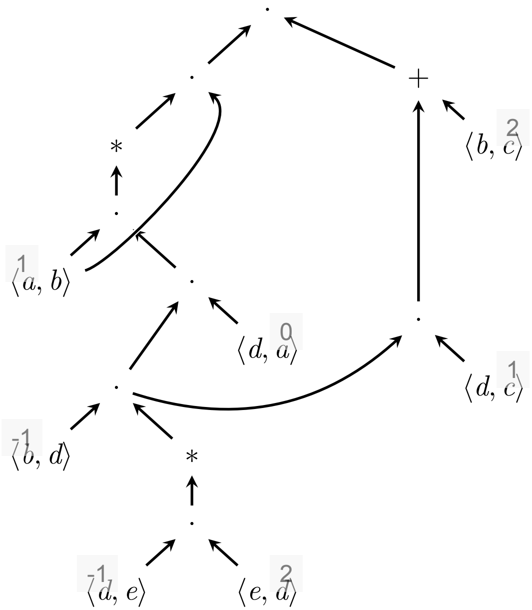
$$d_1 \cdot^D d_2 \triangleq d_1 + d_2$$ Addition

$$d^{*^D} \triangleq \begin{cases} -\infty & \text{if } d < 0 \\ 0 & \text{otherwise} \end{cases}$$ Infimum of $\{nd : n \in \mathbb{N}\}$

Interpreting a path expression DAG

$$(\langle a, b\rangle\langle b, d\rangle\,(\langle d, e\rangle\langle e, d\rangle)^*\,\langle d, a\rangle)^*$$
$$\langle a, b\rangle\,(\langle b, c\rangle + \langle b, d\rangle\,(\langle d, e\rangle\langle e, d\rangle)^*\,\langle d, c\rangle)$$

- Explicit path expressions can be exponential in graph size
- DAG representation to share repeated subexpressions ⇒ polynomial size



Interpreting a path expression DAG

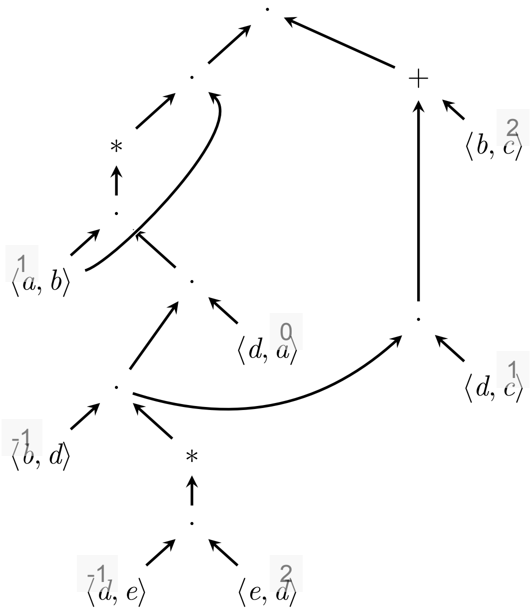
$$0^D = \infty$$

$$1^D = 0$$

$$d_1 +^D d_2 \triangleq \min(d_1, d_2)$$ Minimum

$$d_1 \cdot^D d_2 \triangleq d_1 + d_2$$ Addition

$$d^{*D} \triangleq \begin{cases} -\infty & \text{if } d < 0 \\ 0 & \text{otherwise} \end{cases}$$ $\inf\{nd : n \in \mathbb{N}\}$



Interpreting a path expression DAG

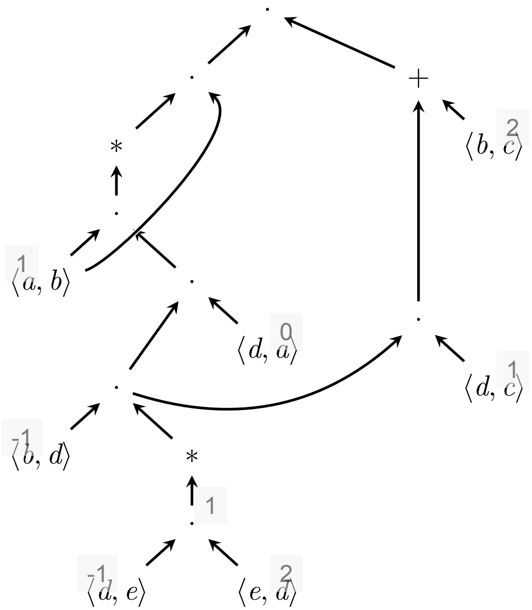
$$0^D = \infty$$

$$1^D = 0$$

$$d_1 +^D d_2 \triangleq \min(d_1, d_2)$$ Minimum

$$d_1 \cdot^D d_2 \triangleq d_1 + d_2$$ Addition

$$d^{*D} \triangleq \begin{cases} -\infty & \text{if } d < 0 \\ 0 & \text{otherwise} \end{cases}$$ $\inf\{nd : n \in \mathbb{N}\}$



Interpreting a path expression DAG

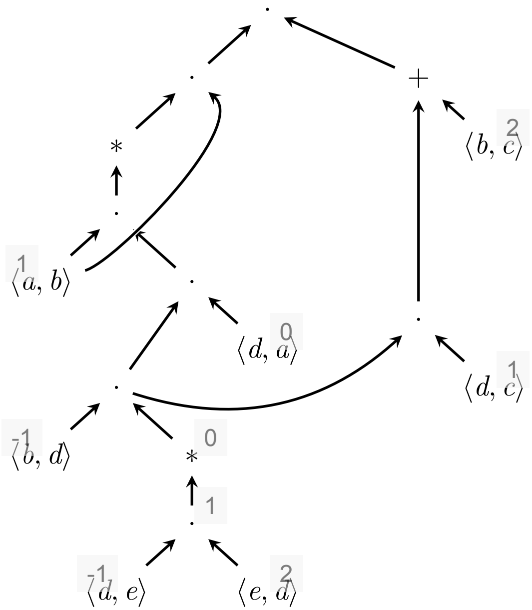
$$0^D = \infty$$

$$1^D = 0$$

$$d_1 +^D d_2 \triangleq \min(d_1, d_2)$$ Minimum

$$d_1 \cdot^D d_2 \triangleq d_1 + d_2$$ Addition

$$d^{*D} \triangleq \begin{cases} -\infty & \text{if } d < 0 \\ 0 & \text{otherwise} \end{cases}$$ $\inf\{nd : n \in \mathbb{N}\}$



Interpreting a path expression DAG

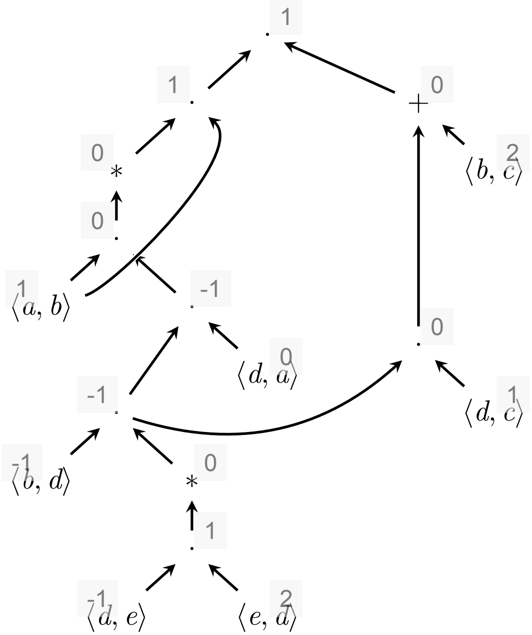
$$0^D = \infty$$

$$1^D = 0$$

$$d_1 +^D d_2 \triangleq \min(d_1, d_2)$$ Minimum

$$d_1 \cdot^D d_2 \triangleq d_1 + d_2$$ Addition

$$d^{*^D} \triangleq \begin{cases} -\infty & \text{if } d < 0 \\ 0 & \text{otherwise} \end{cases}$$ $\inf\{nd : n \in \mathbb{N}\}$



Parallel developments

- **Algebraic path problems**: line of work in algorithms / operations research
- **Elimination-style dataflow analysis**: dataflow analysis using algorithms resembling Gaussian elimination
  [Allen & Cocke '76, Hecht & Ullman '73, Graham & Wegman '76]

Convergence [Tarjan '81]

A Unified Approach to Path Problems

ROBERT ENDRE TARJAN

Stanford University, Stanford, California

ABSTRACT. A general method is described for solving path problems on directed graphs. Such path problems include finding shortest paths, solving sparse systems of linear equations, and carrying out global flow analysis of computer programs The method consists of two steps First, a collection of regular expressions representing sets of paths in the graph is constructed This can be done by using any standard algorithm, such as Gaussian or Gauss–Jordan elimination. Next, a natural mapping from regular

Fast Algorithms for Solving Path Problems

ROBERT ENDRE TARJAN

Stanford University, Stanford, California

ABSTRACT Let $G = (V, E)$ be a directed graph with a distinguished source vertex s. The *single-source path expression problem* is to find, for each vertex v, a regular expression $P(s, v)$ which represents the set of all paths in G from s to v A solution to this problem can be used to solve shortest path problems, solve sparse systems of linear equations, and carry out global flow analysis. A method is described for computing path expressions by dividing G into components, computing path expressions on the components by

- Dataflow analysis as an algebraic path problem
  - Graph: control flow graph
  - Algebra: *transfer functions* $L \to L$ (for some lattice L) [Graham & Wegman '76]

Convergence [Tarjan '81]

A Unified Approach to Path Problems

ROBERT ENDRE TARJAN

Stanford University, Stanford, California

ABSTRACT. A general method is described for solving path problems on directed graphs. Such path problems include finding shortest paths, solving sparse systems of linear equations, and carrying out global flow analysis of computer programs. The method consists of two steps. First, a collection of regular expressions representing sets of paths in the graph is constructed. This can be done by using any standard algorithm, such as Gaussian or Gauss–Jordan elimination. Next, a natural mapping from regular

Fast Algorithms for Solving Path Problems

ROBERT ENDRE TARJAN

Stanford University, Stanford, California

ABSTRACT. Let $G = (V, E)$ be a directed graph with a distinguished source vertex s. The *single-source path expression problem* is to find, for each vertex v, a regular expression $P(s, v)$ which represents the set of all paths in G from s to v. A solution to this problem can be used to solve shortest path problems, solve sparse systems of linear equations, and carry out global flow analysis. A method is described for computing path expressions by dividing G into components, computing path expressions on the components by

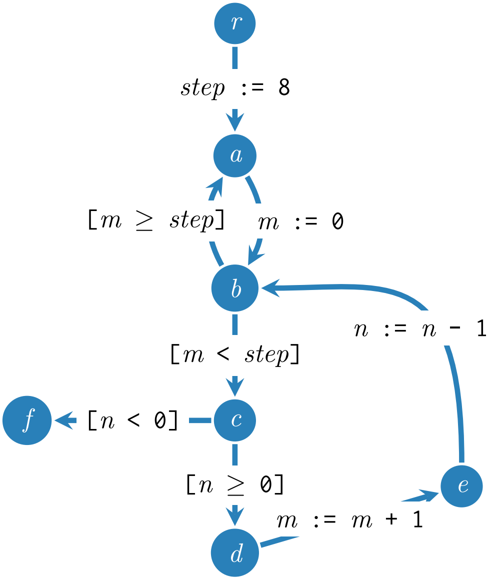
- Dataflow analysis as an algebraic path problem
  - Graph: control flow graph
  - Algebra: *transfer functions* $L \to L$ (for some lattice L) [Graham & Wegman '76]
- Efficient (almost linear time) algorithm for single-source path expression problem
  - *Given:* Graph $G = (V, E)$ and root vertex r
  - *Compute:* For each $v \in V$, a path expression $P(r, v)$ recognizing all paths from r to v in G

Program summarization as an algebraic path problem

```
step = 8
while (true) do
  m := 0
  while (m < step) do
    if (n < 0) then
      halt
    else
      m := m + 1
      n := n - 1
```

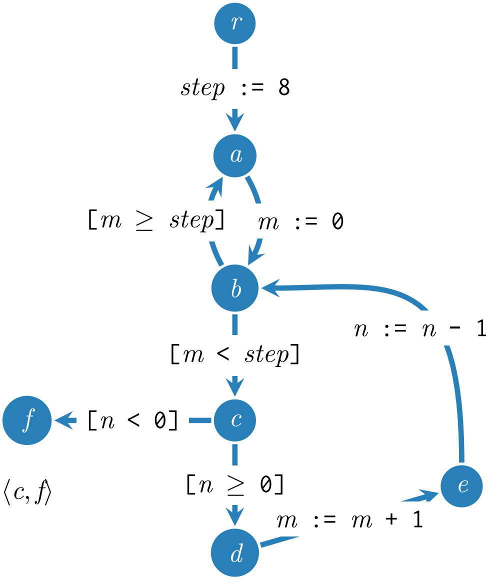
Program summarization as an algebraic path problem

```
step = 8
while (true) do
  m := 0
  while (m < step) do
    if (n < 0) then
      halt
    else
      m := m + 1
      n := n - 1
```



Program summarization as an algebraic path problem

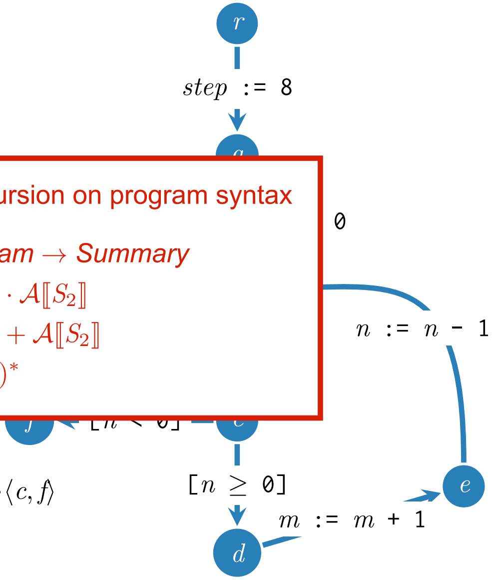
```
step = 8
while (true) do
  m := 0
  while (m < step) do
    if (n < 0) then
      halt
    else
      m := m + 1
      n := n - 1
```



$$\langle r, a\rangle \underbrace{\left(\langle a, b\rangle \overbrace{\left(\langle b, c\rangle\langle c, d\rangle\langle d, e\rangle\langle e, b\rangle\right)^*}^{\text{Inner loop}}\langle b, a\rangle\right)^*}_{\text{Outer loop}}\langle a, b\rangle\langle b, c\rangle\langle c, f\rangle$$

Program summarization as an algebraic path problem

```
step = 8
while (true) do
  m := 0
  while (m < step) do
    if (
      h
    else
      m
      n
```



Recursion on regular expression ∼ recursion on program syntax

$$\mathcal{A}[\![-]\!] : \textit{Program} \to \textit{Summary}$$
$$\mathcal{A}[\![S_1; S_2]\!] = \mathcal{A}[\![S_1]\!] \cdot \mathcal{A}[\![S_2]\!]$$
$$\mathcal{A}[\![\textbf{if}(*)\{S_1\}\textbf{else}\{S_2\}]\!] = \mathcal{A}[\![S_1]\!] + \mathcal{A}[\![S_2]\!]$$
$$\mathcal{A}[\![\textbf{while}(*)\{S\}]\!] = (\mathcal{A}[\![S]\!])^*$$

$$\langle r, a\rangle \underbrace{\left(\langle a, b\rangle \overbrace{\left(\langle b, c\rangle\langle c, d\rangle\langle d, e\rangle\langle e, b\rangle\right)^*}^{\text{Inner loop}} \langle b, a\rangle\right)^*}_{\text{Outer loop}} \langle a, b\rangle\langle b, c\rangle\langle c, f\rangle$$

Transition Formulas

- Transition formula $F(X, X')$: logical formula $\sim$ binary relation on states
  - X: pre-state variables
  - $X' \triangleq \{x' : x \in X\}$: post-state variables

$$\mathit{tf}(\texttt{x := x + 1}) \triangleq \mathsf{x}' = \mathsf{x} + 1 \land \mathsf{y}' = \mathsf{y} \land \mathsf{z}' = \mathsf{z}$$

Transition Formulas

- Transition formula $F(X, X')$: logical formula $\sim$ binary relation on states
  - X: pre-state variables
  - $X' \triangleq \{x' : x \in X\}$: post-state variables

$$\textit{tf}(\texttt{x := x + 1}) \triangleq \mathsf{x}' = \mathsf{x} + 1 \land \mathsf{y}' = \mathsf{y} \land \mathsf{z}' = \mathsf{z}$$

- To verify an assertion:
  1. Compute path expression R from entry to `assert(P)`
  2. Check $\mathbf{TF}[\![R]\!] \land \neg P(X')$
     - UNSAT: assertion verified ✓
     - SAT: No conclusion

Transition Formulas

- Transition formula $F(X, X')$: logical formula $\sim$ binary relation on states
  - X: pre-state variables
  - $X' \triangleq \{x' : x \in X\}$: post-state variables

$$\textit{tf}(\texttt{x := x + 1}) \triangleq \mathsf{x}' = \mathsf{x} + 1 \land \mathsf{y}' = \mathsf{y} \land \mathsf{z}' = \mathsf{z}$$

- To verify an assertion:
  1. Compute path expression R from entry to `assert(P)`
  2. Check $\mathbf{TF}[\![R]\!] \land \neg P(X')$
     - UNSAT: assertion verified ✓
     - SAT: No conclusion
- To bound time usage:
  - Compute path expression R from entry to exit
  - Redefine semantic function *tf'*$(e) \triangleq$ *tf*$(e) \land t' = t + 1$
  - Maximize t' w.r.t. $\mathbf{TF}[\![R]\!]$

Transition Formula Algebras

Universe: set of transition formulas $F(X, X')$ over a fixed set of variables X

$$0^{\mathsf{TF}} \triangleq \textit{false}$$ Empty relation

$$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x$$ Identity relation

Transition Formula Algebras

Universe: set of transition formulas $F(X, X')$ over a fixed set of variables X

$$0^{\mathsf{TF}} \triangleq \textit{false}$$ Empty relation

$$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x$$ Identity relation

$$F +^{\mathsf{TF}} G \triangleq F \vee G$$ Union

Transition Formula Algebras

Universe: set of transition formulas $F(X, X')$ over a fixed set of variables X

$$0^{\mathsf{TF}} \triangleq \textit{false}$$ Empty relation

$$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x$$ Identity relation

$$F +^{\mathsf{TF}} G \triangleq F \vee G$$ Union

$$F \cdot^{\mathsf{TF}} G \triangleq \exists X''. F(X, X'') \wedge G(X'', X')$$ Relational composition

Transition Formula Algebras

Universe: set of transition formulas $F(X, X')$ over a fixed set of variables X

$0^{\mathsf{TF}} \triangleq \textit{false}$	Empty relation
$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x$	Identity relation
$F +^{\mathsf{TF}} G \triangleq F \vee G$	Union
$F \cdot^{\mathsf{TF}} G \triangleq \exists X''. F(X, X'') \wedge G(X'', X')$	Relational composition
$F^{*\mathsf{TF}} \triangleq \ldots$	Approximate transitive closure

Transition Formula Algebras

Universe: set of transition formulas $F(X, X')$ over a fixed set of variables X

$0^{\mathsf{TF}} \triangleq$ *false*	Empty relation
$1^{\mathsf{TF}} \triangleq \bigwedge_{x \in X} x' = x$	Identity relation
$F +^{\mathsf{TF}} G \triangleq F \vee G$	Union
$F \cdot^{\mathsf{TF}} G \triangleq \exists X''. F(X, X'') \wedge G(X'', X')$	Relational composition
$F^{*\mathsf{TF}} \triangleq \ldots$	Approximate transitive closure

Many different implementations

Iteration

$$(-)^* : \mathbf{TF} \to \mathbf{TF}$$

- Input: transition formula summarizing loop body
  - Regardless of structure of inner loop (nested loops, procedure calls, ...)
- Output: transition formula summarizing loop
  - Output language is the same as input language!
- Related work in CAV community: loop summarization / acceleration

Ex. 1: Predicate abstraction

- Houdini [Flanagan & Leino '01]
  - Fix a finite set of predicates P.
  - Infer loop invariants of the form $\left(\bigwedge_{p \in Q \subseteq P} p\right)$ by fixpoint computation

Ex. 1: Predicate abstraction

- Houdini [Flanagan & Leino '01]
  - Fix a finite set of predicates P.
  - Infer loop invariants of the form $\left(\bigwedge_{p \in Q \subseteq P} p\right)$ by fixpoint computation
- *Transition* predicate abstraction [Kroening et al. '08]
  - Fix finite set of *transition* predicates P such that each $p \in P$ is
    - reflexive: $X = X' \models p(X, X')$.
    - transitive: $p(X, X') \wedge p(X', X'') \models p(X, X'')$.
  - Examples: $(x \leq x')$, $(x \geq 0 \Rightarrow x' \geq 0)$, ...
  - Non-examples: $x < x'$ $(x \geq 0 \Rightarrow (x' \leq x))$

Ex. 1: Predicate abstraction

- Houdini [Flanagan & Leino '01]
  - Fix a finite set of predicates P.
  - Infer loop invariants of the form $\left(\bigwedge_{p \in Q \subseteq P} p\right)$ by fixpoint computation
- *Transition* predicate abstraction [Kroening et al. '08]
  - Fix finite set of *transition* predicates P such that each $p \in P$ is
    - reflexive: $X = X' \models p(X, X')$.
    - transitive: $p(X, X') \wedge p(X', X'') \models p(X, X'')$.
  - Examples: $(x \leq x')$, $(x \geq 0 \Rightarrow x' \geq 0)$, ...
  - Non-examples: $x < x'$ $(x \geq 0 \Rightarrow (x' \leq x))$
- Iteration operator: $F^* \triangleq \bigwedge\{p \in P : F \models p\}$
  - No fixpoint computation (max $|P|$ calls to an SMT solver).

Ex 2: Interval analysis

- **Interval invariant** for a loop is an inductive invariant of the form

$$\bigwedge_{x \in X} a_x \leq x \leq b_x$$

Ex 2: Interval analysis

- **Interval invariant** for a loop is an inductive invariant of the form

$$\bigwedge_{x \in X} a_x \leq x \leq b_x$$

```
i = 0;
j = 0;
while (i < 10 ∧ j ≠ 20 ∧ j < 100)  {
  i = i + 1;
  j = j + 1;
}
```

Ex 2: Interval analysis

- **Interval invariant** for a loop is an inductive invariant of the form

$$\bigwedge_{x \in X} a_x \leq x \leq b_x$$

```
i = 0;
j = 0;
while (i < 10 ∧ j ≠ 20 ∧ j < 100)  {
  i = i + 1;
  j = j + 1;
}
```

$0 \leq i \leq 10 \wedge 0 \leq j \leq 100$

Ex 2: Interval analysis

- **Interval invariant** for a loop is an inductive invariant of the form

$$\bigwedge_{x \in X} a_x \leq x \leq b_x$$

$0 \leq i \leq 10 \land 0 \leq j \leq 20$

```
i = 0;
j = 0;
while (i < 10 ∧ j ≠ 20 ∧ j < 100) {
  i = i + 1;
  j = j + 1;
}
```

Ex 2: Interval analysis

- **Interval invariant** for a loop is an inductive invariant of the form

$$\bigwedge_{x \in X} a_x \leq x \leq b_x$$

```
i = 0;
j = 0;
while (i < 10 ∧ j ≠ 20 ∧ j < 100) {
  i = i + 1;
  j = j + 1;
}
```

$0 \leq i \leq 10 \land 0 \leq j \leq 10$ ✗

Ex 2: Interval analysis

- **Interval invariant** for a loop is an inductive invariant of the form

$$\bigwedge_{x \in X} a_x \leq x \leq b_x$$

```
i = 0;
j = 0;
while (i < 10 ∧ j ≠ 20 ∧ j < 100)  {
  i = i + 1;
  j = j + 1;
}
```

- Classical approach to computing interval invariants: iterative, using widening/narrowing [Cousot & Cousot '76]
  - Computes *some* interval invariant; not nessarily *best*

Ex 2: Interval analysis

- Interval invariant for TF $F(X, X')$: for each variable x, a pair a_x, b_x such that

$$\models \underbrace{\forall X, X'. \left(\left(\bigwedge_{x \in X} a_x \leq x \leq b_x \right) \land F(X, X') \right) \Rightarrow \bigwedge_{x \in X} a_x \leq x' \leq b_x}_{Inv(A,B)}$$

Ex 2: Interval analysis

- Interval invariant for TF $F(X, X')$: for each variable x, a pair a_x, b_x such that

$$\models \underbrace{\forall X, X'. \left(\left(\bigwedge_{x \in X} a_x \leq x \leq b_x \right) \wedge F(X, X') \right) \Rightarrow \bigwedge_{x \in X} a_x \leq x' \leq b_x}_{Inv(A,B)}$$

- $F^* \triangleq \forall A, B. \left(\mathit{Inv}(A, B) \wedge \bigwedge_{x \in X} a_x \leq x \leq b_x \right) \Rightarrow \bigwedge_{x \in X} a_x \leq x' \leq b_x$

  [Monniaux '09]

Ex 2: Interval analysis

- Interval invariant for TF $F(X, X')$: for each variable x, a pair a_x, b_x such that

$$\models \underbrace{\forall X, X'. \left(\left(\bigwedge_{x \in X} a_x \leq x \leq b_x \right) \wedge F(X, X') \right) \Rightarrow \bigwedge_{x \in X} a_x \leq x' \leq b_x}_{Inv(A,B)}$$

- $F^* \triangleq \forall A, B. \left(\mathit{Inv}(A, B) \wedge \bigwedge_{x \in X} a_x \leq x \leq b_x \right) \Rightarrow \bigwedge_{x \in X} a_x \leq x' \leq b_x$
  [Monniaux '09]
- F^* entails **all** interval invariants of F.

Ex 3: Recurrence analysis

```
while (x > 0) do
  if (y < 0) then
    x := x + y
    y := y - 1
  else
    x := x - 2
    y := y - 3
```

loop body

$$x > 0 \wedge \begin{pmatrix} (y < 0 \wedge x' = x + y \wedge y' = y - 1) \\ \vee (y \geq 0 \wedge x' = x - 2 \wedge y' = y - 2) \end{pmatrix}$$

Ex 3: Recurrence analysis

```
while (x > 0) do
  if (y < 0) then
    x := x + y
    y := y - 1
  else
    x := x - 2
    y := y - 3
```

loop body

$$x > 0 \wedge \begin{pmatrix} (y < 0 \wedge x' = x + y \wedge y' = y - 1) \\ \vee (y \geq 0 \wedge x' = x - 2 \wedge y' = y - 2) \end{pmatrix}$$

recurrences

$$y^{(k)} \leq y^{(k-1)} - 1$$

$$y^{(k)} \geq y^{(k-1)} - 3$$

$$(2x^{(k)} - y^{(k)}) \leq (2x^{(k-1)} - y^{(k-1)}) - 1$$

Ex 3: Recurrence analysis

```
while (x > 0) do
  if (y < 0) then
    x := x + y
    y := y - 1
  else
    x := x - 2
    y := y - 3
```

loop body

$$\mathsf{x} > 0 \wedge \begin{pmatrix} (y < 0 \wedge \mathsf{x}' = \mathsf{x} + \mathsf{y} \wedge \mathsf{y}' = \mathsf{y} - 1) \\ \vee (\mathsf{y} \geq 0 \wedge \mathsf{x}' = \mathsf{x} - 2 \wedge \mathsf{y}' = \mathsf{y} - 2) \end{pmatrix}$$

recurrences

$$\mathsf{y}^{(k)} \leq \mathsf{y}^{(k-1)} - 1$$

$$\mathsf{y}^{(k)} \geq \mathsf{y}^{(k-1)} - 3$$

$$(2\mathsf{x}^{(k)} - \mathsf{y}^{(k)}) \leq (2\mathsf{x}^{(k-1)} - \mathsf{y}^{(k-1)}) - 1$$

closed forms

$$\mathsf{y}^{(k)} \leq \mathsf{y}^{(0)} - k$$

$$\mathsf{y}^{(k)} \geq \mathsf{y}^{(0)} - 3k$$

$$(2\mathsf{x}^{(k)} - \mathsf{y}^{(k)}) \leq (2\mathsf{x}^{(0)} - \mathsf{y}^{(0)}) - k$$

Ex 3: Recurrence analysis

```
while (x > 0) do
  if (y < 0) then
    x := x + y
    y := y - 1
  else
    x := x - 2
    y := y - 3
```

loop body

$$\mathsf{x} > 0 \wedge \begin{pmatrix} (y < 0 \wedge \mathsf{x}' = \mathsf{x} + \mathsf{y} \wedge \mathsf{y}' = \mathsf{y} - 1) \\ \vee (\mathsf{y} \geq 0 \wedge \mathsf{x}' = \mathsf{x} - 2 \wedge \mathsf{y}' = \mathsf{y} - 2) \end{pmatrix}$$

recurrences

$$\mathsf{y}^{(k)} \leq \mathsf{y}^{(k-1)} - 1$$

$$\mathsf{y}^{(k)} \geq \mathsf{y}^{(k-1)} - 3$$

$$(2\mathsf{x}^{(k)} - \mathsf{y}^{(k)}) \leq (2\mathsf{x}^{(k-1)} - \mathsf{y}^{(k-1)}) - 1$$

closed forms

$$\mathsf{y}^{(k)} \leq \mathsf{y}^{(0)} - k$$

$$\mathsf{y}^{(k)} \geq \mathsf{y}^{(0)} - 3k$$

$$(2\mathsf{x}^{(k)} - \mathsf{y}^{(k)}) \leq (2\mathsf{x}^{(0)} - \mathsf{y}^{(0)}) - k$$

loop abstraction

$$\exists k. k \geq 0 \wedge \mathsf{y}' \leq \mathsf{y} - k \wedge \mathsf{y}' \geq \mathsf{y} - 3k \wedge (2\mathsf{x}' - \mathsf{y}') \leq (2\mathsf{x} - \mathsf{y}) - k$$

Ex 3: Recurrence analysis

```
while (x > 0) do
  if (y < 0) then
    x := x + y
    y := y - 1
  else
    x := x - 2
    y := y - 3
```

loop body

$$x > 0 \wedge \begin{pmatrix} (y < 0 \wedge x' = x + y \wedge y' = y - 1) \\ \vee (y \geq 0 \wedge x' = x - 2 \wedge y' = y - 2) \end{pmatrix}$$

recurrences

$$y^{(k)} \leq y^{(k-1)} - 1$$

$$y^{(k)} \geq y^{(k-1)} - 3$$

$$(2x^{(k)} - y^{(k)}) \leq (2x^{(k-1)} - y^{(k-1)}) - 1$$

closed forms

$$y^{(k)} \leq y^{(0)} - k$$

$$y^{(k)} \geq y^{(0)} - 3k$$

$$(2x^{(k)} - y^{(k)}) \leq (2x^{(0)} - y^{(0)}) - k$$

loop abstraction

$$\exists k. k \geq 0 \wedge y' \leq y - k \wedge y' \geq y - 3k \wedge (2x' - y') \leq (2x - y) - k$$

Ex 3: Recurrence analysis

- A **linear recurrence relation** for a TF $F(X, X')$ is a formula of the form $\mathbf{a}^T\mathbf{x}' \leq \mathbf{a}^T\mathbf{x} + b$ entailed by F
  - $\mathbf{x}/\mathbf{x}'$ denote vectors containing X/X'

Ex 3: Recurrence analysis

- A **linear recurrence relation** for a TF $F(X, X')$ is a formula of the form $\mathbf{a}^T\mathbf{x}' \leq \mathbf{a}^T\mathbf{x} + b$ entailed by F
  - $\mathbf{x}/\mathbf{x}'$ denote vectors containing X/X'
- *Rec*(F) $\triangleq$ convex cone of all linear recurrence relations of F
  - *Rec*$(F) \cong$ valid inequalities of

$$\Delta(F) \triangleq \left(\exists X, X'.F(X, X') \wedge \bigwedge_{x \in X} \delta_x = (x' - x) \right)$$

    That is,

$$F \models \mathbf{a}^T\mathbf{x}' \leq \mathbf{a}^T\mathbf{x} + b \iff \Delta(F) \models \mathbf{a}^T\delta \leq b$$

  - Generators of *Rec*(F) can be computed from convex hull of $\Delta(F)$ [Ancourt et al. '10, Farzan & K '2015]
  - I.e., we can compute **all** implied linear recurrence relations

… and many more

- Polynomial recurrence relations with polynomial / complex exponential closed forms [K et al. ’2018]
- Polynomial recurrence relations with polynomial / rational exponential closed forms [K et al. ’2019]
- Vector addition systems [Silverman & K ’19]
- Octagonal relations [Bozga et al. ’09]
- Combinations thereof
- ...

Ex 4: Affine relation analysis [Karr '76]

- An *affine relation* is a TF of the form $A\mathbf{x}' = B\mathbf{x} + \mathbf{c}$
  - *Sub*universe of transition formulas

Ex 4: Affine relation analysis [Karr '76]

- An *affine relation* is a TF of the form $A\mathbf{x}' = B\mathbf{x} + \mathbf{c}$
  - *Sub*universe of transition formulas
- Closed under relational composition, but not disjunction
  - $F + G \triangleq$ affine hull of $F \vee G$

Ex 4: Affine relation analysis [Karr '76]

- An *affine relation* is a TF of the form $A\mathbf{x}' = B\mathbf{x} + \mathbf{c}$
  - *Sub*universe of transition formulas
- Closed under relational composition, but not disjunction
  - $F + G \triangleq$ affine hull of $F \vee G$
- Lattice of affine relations has no infinite increasing chains
  - $1 \subseteq 1 + F \subseteq 1 + F + (F \circ F) \subseteq \cdots$ reaches limit at some $n \leq 2|X|$
  - $F^* \triangleq \sum_{i=0}^{n} \underbrace{F \circ \cdots \circ F}_{i \text{ times}}$ (*Least* solution to $F^* \circ F + 1 = F^*$)

Designing an algebraic analysis

1. **Define**:
   - Semantic algebra $\mathcal{A} = \langle A, \cdot, +, *, 0, 1 \rangle$
   - Semantic function $f : E \to A$
2. **Apply**: Tarjan's path expression algorithm

Iterative vs. algebraic program analysis

Iterative Framework	Algebraic Framework
Join semi-lattice	Semantic Algebra
Abstract transformers	Semantic function
Chaotic iteration algorithm	Path-expression algorithm

Key differences

- Algebraic analyses are *compositional*
- Loop analysis is *internal* to an algebraic program analysis