Abstractions from Proofs

T. A. Henzinger R. Jhala R. Majumdar K. L. McMillan

Talk by Arnaud Fietzke

Software Model Checking Seminar SS05

30th June 2005

Outline

1. Introduction
   - Motivation
   - Craig Interpolation

Outline

1. Introduction
   - Motivation
   - Craig Interpolation
2. Interpolants from Proofs
   - Interpolated Sequents
   - Completeness

Outline

1. Introduction
   - Motivation
   - Craig Interpolation
2. Interpolants from Proofs
   - Interpolated Sequents
   - Completeness
3. Language and Abstraction
   - Program representation
   - Strongest Postcondition and Predicate Abstraction

Outline

1. Introduction
   - Motivation
   - Craig Interpolation
2. Interpolants from Proofs
   - Interpolated Sequents
   - Completeness
3. Language and Abstraction
   - Program representation
   - Strongest Postcondition and Predicate Abstraction
4. Putting Things Together
   - Basic Definitions
   - Pointer-free Programs
   - Handling Function Calls
   - Handling Pointers

Motivation

- Counterexample-guided abstraction refinement continously adds new predicates.

Motivation

- Counterexample-guided abstraction refinement continously adds new predicates.
- However, most predicates are only locally useful.

Motivation

- Counterexample-guided abstraction refinement continously adds new predicates.
- However, most predicates are only locally useful.
- We need a way to
  - learn a minimal set of predicates from a spurious counterexample
  - use these predicates only where they are useful

Motivation

Example

```
      while(*) {
1:      if(p1) lock();
        if(p1) unlock();
        ...
2:      if(p2) lock();
        if(p2) unlock();
        ...
n:      if(pn) lock();
        if(pn) unlock();
```

Motivation

Example

```
    while(*) {
1:    if(p1) lock();
      if(p1) unlock();
      ...
2:    if(p2) lock();
      if(p2) unlock();
      ...
n:    if(pn) lock();
      if(pn) unlock();
```

assume $p1$;
lock();
assume $\neg p1$;
assume $p2$;
lock();

Each pi is useful only between i and $i + 1$.

Craig Interpolation

Definition

Given a pair of formulas (φ^-, φ^+),
an **interpolant** for (φ^-, φ^+) is a formula ψ such that

Craig Interpolation

Definition

Given a pair of formulas (φ^-, φ^+),
an **interpolant** for (φ^-, φ^+) is a formula ψ such that

- φ^- implies ψ

Craig Interpolation

Definition

Given a pair of formulas (φ^-, φ^+),
an **interpolant** for (φ^-, φ^+) is a formula ψ such that

- φ^- implies ψ
- $\psi \wedge \varphi^+$ is unsatisfiable

Craig Interpolation

Definition

Given a pair of formulas (φ^-, φ^+),
an **interpolant** for (φ^-, φ^+) is a formula ψ such that

- φ^- implies ψ
- $\psi \wedge \varphi^+$ is unsatisfiable
- the variables of ψ are common to both φ^- and φ^+

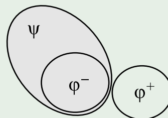
Craig Interpolation

Definition

Given a pair of formulas (φ^-, φ^+), an **interpolant** for (φ^-, φ^+) is a formula ψ such that

- φ^- implies ψ
- $\psi \wedge \varphi^+$ is unsatisfiable
- the variables of ψ are common to both φ^- and φ^+

Example



Craig Interpolation

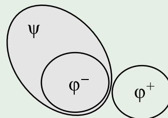
Definition

Given a pair of formulas (φ^-, φ^+),
an **interpolant** for (φ^-, φ^+) is a formula ψ such that

- φ^- implies ψ
- $\psi \wedge \varphi^+$ is unsatisfiable
- the variables of ψ are common to both φ^- and φ^+

ψ always exists if $\varphi^- \wedge \varphi^+$ unsatisfiable.

Example



FOL

We consider formulas in quantifier-free first-order logic of linear equality:

- a **term** is of the form $c_0 + c_1 x_1 + ... + c_n x_n$ with variables $x_1, ..., x_n$ and integer constants $c_1, ..., c_n$

FOL

We consider formulas in quantifier-free first-order logic of linear equality:

- a **term** is of the form $c_0 + c_1 x_1 + ... + c_n x_n$ with variables $x_1, ..., x_n$ and integer constants $c_1, ..., c_n$
- an **atomic predicate** is either
  - a propositional variable
  - or of the form $0 \leq x$ where x is a term

FOL

We consider formulas in quantifier-free first-order logic of linear equality:

- a **term** is of the form $c_0 + c_1 x_1 + ... + c_n x_n$ with variables $x_1, ..., x_n$ and integer constants $c_1, ..., c_n$
- an **atomic predicate** is either
  - a propositional variable
  - or of the form $0 \leq x$ where x is a term
- a **literal** is a (possibly negated) atomic predicate
- a **clause** is a disjunction of literals

FOL

We consider formulas in quantifier-free first-order logic of linear equality:

- a **term** is of the form $c_0 + c_1 x_1 + ... + c_n x_n$ with variables $x_1, ..., x_n$ and integer constants $c_1, ..., c_n$
- an **atomic predicate** is either
  - a propositional variable
  - or of the form $0 \leq x$ where x is a term
- a **literal** is a (possibly negated) atomic predicate
- a **clause** is a disjunction of literals
- a **sequent** is of the form $\Gamma \vdash \Delta$ where Γ and Δ are sets of formulas.

FOL

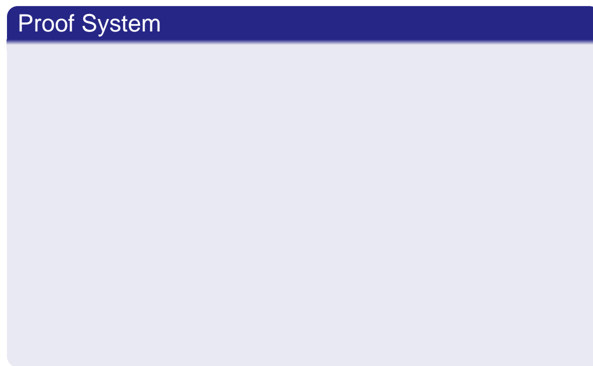
We consider formulas in quantifier-free first-order logic of linear equality:

- a **term** is of the form $c_0 + c_1 x_1 + ... + c_n x_n$ with variables $x_1, ..., x_n$ and integer constants $c_1, ..., c_n$
- an **atomic predicate** is either
  - a propositional variable
  - or of the form $0 \leq x$ where x is a term
- a **literal** is a (possibly negated) atomic predicate
- a **clause** is a disjunction of literals
- a **sequent** is of the form $\Gamma \vdash \Delta$ where Γ and Δ are sets of formulas.

Interpretation of $\Gamma \vdash \Delta$: $\bigwedge_{\phi \in \Gamma} \phi \Rightarrow \bigvee_{\phi \in \Delta} \phi$

Refutation Proof System

Assume we have a theorem prover that generates refutations for sets of clauses using the following



Refutation Proof System

Assume we have a theorem prover that generates refutations for sets of clauses using the following

Proof System

$$\textsc{Hyp}\ \frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma$$

Refutation Proof System

Assume we have a theorem prover that generates refutations for sets of clauses using the following

Proof System

$$\text{HYP}\ \frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma$$

$$\text{COMB}\ \frac{\Gamma \vdash 0 \leq x \qquad \Gamma \vdash 0 \leq y}{\Gamma \vdash 0 \leq c_1 x + c_2 y}\ c_{1,2} > 0$$

Refutation Proof System

Assume we have a theorem prover that generates refutations for sets of clauses using the following

Proof System

$$\textsc{Hyp}\ \frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma$$

$$\textsc{Comb}\ \frac{\Gamma \vdash 0 \leq x \qquad \Gamma \vdash 0 \leq y}{\Gamma \vdash 0 \leq c_1 x + c_2 y}\ c_{1,2} > 0$$

$$\textsc{Contra}\ \frac{\{\phi_1, \ldots, \phi_n\} \vdash 0 \leq c}{\Gamma \vdash \neg\phi_1, \ldots, \neg\phi_n}\ c < 0$$

Refutation Proof System

Assume we have a theorem prover that generates refutations for sets of clauses using the following

Proof System

$$\textsc{Hyp}\ \frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma$$

$$\textsc{Comb}\ \frac{\Gamma \vdash 0 \leq x \qquad \Gamma \vdash 0 \leq y}{\Gamma \vdash 0 \leq c_1 x + c_2 y}\ c_{1,2} > 0$$

$$\textsc{Contra}\ \frac{\{\phi_1, \ldots, \phi_n\} \vdash 0 \leq c}{\Gamma \vdash \neg\phi_1, \ldots, \neg\phi_n}\ c < 0$$

$$\textsc{Res}\ \frac{\Gamma \vdash \{\phi\} \cup \Theta \qquad \Gamma \vdash \{\neg\phi\} \cup \Theta'}{\Gamma \vdash \Theta \cup \Theta'}$$

Interpolated sequents

We want to extract interpolants from an infeasibility proof of a spurious error trace.

Interpolated sequents

We want to extract interpolants from an infeasibility proof of a spurious error trace.

Definition

- An **interpolated sequent** is of the form $(\varphi^-, \varphi^+) \vdash \Delta\ [\psi]$ with
  - φ^- and φ^+ clause sets,
  - Δ a set of formulas,
  - ψ a formula.

Interpolated sequents

We want to extract interpolants from an infeasibility proof of a spurious error trace.

Definition

- An **interpolated sequent** is of the form $(\varphi^-, \varphi^+) \vdash \Delta\ [\psi]$ with
  - φ^- and φ^+ clause sets,
  - Δ a set of formulas,
  - ψ a formula.

Interpretation:

(1) $\varphi^- \vdash \psi$

Interpolated sequents

We want to extract interpolants from an infeasibility proof of a spurious error trace.

Definition

- An **interpolated sequent** is of the form $(\varphi^-, \varphi^+) \vdash \Delta\ [\psi]$ with
  - φ^- and φ^+ clause sets,
  - Δ a set of formulas,
  - ψ a formula.

Interpretation:

(1) $\varphi^- \vdash \psi$

(2) $\psi, \varphi^+ \vdash \Delta$

Interpolated sequents

We want to extract interpolants from an infeasibility proof of a spurious error trace.

Definition

- An **interpolated sequent** is of the form $(\varphi^-, \varphi^+) \vdash \Delta \, [\psi]$ with
  - φ^- and φ^+ clause sets,
  - Δ a set of formulas,
  - ψ a formula.

Interpretation:

(1) $\varphi^- \vdash \psi$

(2) $\psi, \varphi^+ \vdash \Delta$

(3) $\psi \preceq (\varphi^- \cap \varphi^+) \cup \Delta$

Interpolated sequents

We want to extract interpolants from an infeasibility proof of a spurious error trace.

Definition

- An **interpolated sequent** is of the form $(\varphi^-, \varphi^+) \vdash \Delta\ [\psi]$ with
  - φ^- and φ^+ clause sets,
  - Δ a set of formulas,
  - ψ a formula.

Interpretation:

(1) $\varphi^- \vdash \psi$

(2) $\psi, \varphi^+ \vdash \Delta$

(3) $\psi \preceq (\varphi^- \cap \varphi^+) \cup \Delta$

If $(\varphi^-, \varphi^+) \vdash \bot\ [\psi]$ then ψ is an interpolant for (φ^-, φ^+).

Derivation rules

Proof rule

$$\textsc{Hyp}\ \frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma$$

Derivation rules

$$\textsc{Hyp-A}\ \frac{}{(\varphi^-, \varphi^+) \vdash \phi\ [\phi]}\ \phi \in \varphi^-$$

$$\textsc{Hyp-B}\ \frac{}{(\varphi^-, \varphi^+) \vdash \phi\ [\top]}\ \phi \in \varphi^+$$

$\top$ stands for $0 \leq 0$

Derivation rules

Proof rule

$$\textsc{Comb}\ \frac{\Gamma \vdash 0 \le x \quad \Gamma \vdash 0 \le y}{\Gamma \vdash 0 \le c_1 x + c_2 y}\ c_{1,2} > 0$$

Derivation rule

$$\textsc{Comb}\ \frac{\begin{array}{c}(\varphi^-, \varphi^+) \vdash 0 \le x \ \ [0 \le x'] \\ (\varphi^-, \varphi^+) \vdash 0 \le y \ \ [0 \le y']\end{array}}{(\varphi^-, \varphi^+) \vdash 0 \le c_1 x + c_2 y \ \ [0 \le c_1 x' + c_2 y']}\ c_{1,2} > 0$$

Derivation rules

Proof rule

$$\textsc{Contra}\ \frac{\{\phi_1, ..., \phi_n\} \vdash 0 \leq c}{\Gamma \vdash \neg\phi_1, ..., \neg\phi_n}\ c < 0$$

Derivation rule

$$\textsc{Contra}\ \frac{(\{a_1, \ldots, a_k\}, \{b_1, \ldots, b_m\}) \vdash \bot \ \ [\psi]}{(\varphi^-, \varphi^+) \vdash \neg a_1, \ldots, \neg a_k, \neg b_1, \ldots, \neg b_m \ \ [(\neg a_1 \vee \cdots \vee \neg a_k) \vee \psi]}$$

Derivation rules

Proof rule

$$\textsc{Res}\ \frac{\Gamma \vdash \{\phi\} \cup \Theta \qquad \Gamma \vdash \{\neg\phi\} \cup \Theta'}{\Gamma \vdash \Theta \cup \Theta'}$$

Derivation rules

$$\textsc{Res-A}\ \frac{\begin{array}{c}(\varphi^-, \varphi^+) \vdash \phi, \Theta \ \ [(\phi \vee \rho) \vee \psi] \\ (\varphi^-, \varphi^+) \vdash \neg\phi, \Theta' \ \ [(\neg\phi \vee \rho') \vee \psi']\end{array}}{(\varphi^-, \varphi^+) \vdash \Theta, \Theta' \ \ [(\rho \vee \rho') \vee (\psi \vee \psi')]}$$

Derivation rules

Proof rule

$$\textsc{Res}\ \frac{\Gamma \vdash \{\phi\} \cup \Theta \qquad \Gamma \vdash \{\neg\phi\} \cup \Theta'}{\Gamma \vdash \Theta \cup \Theta'}$$

Derivation rules

$$\textsc{Res-A}\ \frac{\begin{array}{c}(\varphi^-, \varphi^+) \vdash \phi, \Theta \ \ [(\phi \vee \rho) \vee \psi] \\ (\varphi^-, \varphi^+) \vdash \neg\phi, \Theta' \ \ [(\neg\phi \vee \rho') \vee \psi']\end{array}}{(\varphi^-, \varphi^+) \vdash \Theta, \Theta' \ \ [(\rho \vee \rho') \vee (\psi \vee \psi')]}$$

$$\textsc{Res-B}\ \frac{\begin{array}{c}(\varphi^-, \varphi^+) \vdash \phi, \Theta \ \ [\rho \vee \psi] \\ (\varphi^-, \varphi^+) \vdash \neg\phi, \Theta' \ \ [\rho' \vee \psi']\end{array}}{(\varphi^-, \varphi^+) \vdash \Theta, \Theta' \ \ [(\rho \vee \rho') \vee (\psi \wedge \psi')]}$$

Derivation rules

- The derivation rules are complete w.r.t. the proof system:

Derivation rules

- The derivation rules are complete w.r.t. the proof system:

  For every derivation $\mathcal{P}$ of $(\varphi^-, \varphi^+) \vdash \phi$, exists a corresponding derivation $\mathcal{P}'$ of $(\varphi^-, \varphi^+) \vdash \phi\ [\psi]$

Derivation rules

- The derivation rules are complete w.r.t. the proof system:

  For every derivation $\mathcal{P}$ of $(\varphi^-, \varphi^+) \vdash \phi$, exists a corresponding derivation $\mathcal{P}'$ of $(\varphi^-, \varphi^+) \vdash \phi\ [\psi]$

- We refer to ψ as $\text{ITP}.(\varphi^-, \varphi^+).\mathcal{P}$

Derivation rules

Example

Let $\varphi := \varphi^- \cup \varphi^+$ with

- $\varphi^- := (0 \leq y - x)(0 \leq z - y)$
- $\varphi^+ := (0 \leq x - z - 1)$

$$\textsc{Comb}\ \dfrac{\textsc{Comb}\ \dfrac{\textsc{Hyp}\ \dfrac{}{\varphi \vdash (0 \leq y - x)} \quad \textsc{Hyp}\ \dfrac{}{\varphi \vdash (0 \leq z - y)}}{\varphi \vdash (0 \leq z - x)} \quad \textsc{Hyp}\ \dfrac{}{\varphi \vdash (0 \leq x - z - 1)}}{\varphi \vdash (0 \leq -1)}$$

Derivation rules

Example

Let $\varphi := \varphi^- \cup \varphi^+$ with

- $\varphi^- := (0 \leq y - x)(0 \leq z - y)$
- $\varphi^+ := (0 \leq x - z - 1)$

$$\dfrac{\dfrac{}{\varphi \vdash (0 \leq y - x)\ [0 \leq y - x]}\,\textsc{Hyp-A} \quad \dfrac{}{\varphi \vdash (0 \leq z - y)\ [0 \leq z - y]}\,\textsc{Hyp-A}}{\varphi \vdash (0 \leq z - x)}\,\textsc{Comb}$$

$$\dfrac{\varphi \vdash (0 \leq z - x) \quad \dfrac{}{\varphi \vdash (0 \leq x - z - 1)\ [\top]}\,\textsc{Hyp-B}}{\varphi \vdash (0 \leq -1)}\,\textsc{Comb}$$

Derivation rules

Example

Let $\varphi := \varphi^- \cup \varphi^+$ with

- $\varphi^- := (0 \leq y - x)(0 \leq z - y)$
- $\varphi^+ := (0 \leq x - z - 1)$

$$\dfrac{\dfrac{\dfrac{}{\varphi \vdash (0 \leq y - x)\ [0 \leq y - x]}\textsc{Hyp-A} \quad \dfrac{}{\varphi \vdash (0 \leq z - y)\ [0 \leq z - y]}\textsc{Hyp-A}}{\varphi \vdash (0 \leq z - x)\ [0 \leq z - x]}\textsc{Comb} \quad \dfrac{}{\varphi \vdash (0 \leq x - z - 1)\ [\top]}\textsc{Hyp-B}}{\varphi \vdash (0 \leq -1)}\textsc{Comb}$$

Derivation rules

Example

Let $\varphi := \varphi^- \cup \varphi^+$ with

- $\varphi^- := (0 \le y - x)(0 \le z - y)$
- $\varphi^+ := (0 \le x - z - 1)$

$$\dfrac{\dfrac{\text{HYP-A}\;\dfrac{}{\varphi \vdash (0 \le y - x)\ [0 \le y - x]} \qquad \text{HYP-A}\;\dfrac{}{\varphi \vdash (0 \le z - y)\ [0 \le z - y]}}{\varphi \vdash (0 \le z - x)\ [0 \le z - x]}\text{COMB} \qquad \text{HYP-B}\;\dfrac{}{\varphi \vdash (0 \le x - z - 1)\ [\top]}}{\varphi \vdash \bot\ [0 \le z - x]}\text{COMB}$$

The Interpolant is generated by a linear scan of the proof.

Program representation

Syntax

- We consider a language with
  - integer variables,
  - pointers,
  - call-by-value functions
- and the following operations (set *Ops*):

Program representation

Syntax

- We consider a language with
  - integer variables,
  - pointers,
  - call-by-value functions
- and the following operations (set *Ops*):
  - $l := e$, assigns value of e to location $l \in Lvals$

Program representation

Syntax

- We consider a language with
  - integer variables,
  - pointers,
  - call-by-value functions
- and the following operations (set *Ops*):
  - $l := e$, assigns value of e to location $l \in Lvals$
  - `assume`(p), succeeds iff p evaluates to *true*

Program representation

Syntax

- We consider a language with
  - integer variables,
  - pointers,
  - call-by-value functions
- and the following operations (set *Ops*):
  - $l := e$, assigns value of e to location $l \in Lvals$
  - `assume`(p), succeeds iff p evaluates to *true*
  - $f(x_1, \ldots, x_n)$, function call with actual parameters $x_1, \ldots, x_n$
  - and `return` to the caller

Program representation

Syntax

- Each function f is represented as a control flow automaton C_f

Program representation

Syntax

- Each function f is represented as a control flow automaton C_f

Definition

A **CFA** is defined by $C_f = (L_f, E_f, l_f^0, Op_f, V_f)$ with

- $L_f \subseteq PC, \quad E_f \subseteq L_f \times L_f, \quad l_f^0 \in L_f$
- $Op_f : E_f \rightarrow Ops$
- local variables $V_f \subseteq Lvals$ with
  - $X_f \subseteq V_f$ set of formal parameters
  - $r_f \in V_f$ variable storing return value

Program representation

Syntax

- Each function f is represented as a control flow automaton C_f

Definition

A **CFA** is defined by $C_f = (L_f, E_f, l_f^0, Op_f, V_f)$ with

- $L_f \subseteq PC, \quad E_f \subseteq L_f \times L_f, \quad l_f^0 \in L_f$
- $Op_f : E_f \rightarrow Ops$
- local variables $V_f \subseteq Lvals$ with
  - $X_f \subseteq V_f$ set of formal parameters
  - $r_f \in V_f$ variable storing return value

- A program is a set of CFAs $\mathbb{P} = \{C_{f_0}, \ldots, C_{f_k}\} \ni C_{main}$

Program representation

Syntax

- Each function f is represented as a control flow automaton C_f

Definition

A **CFA** is defined by $C_f = (L_f, E_f, l_f^0, Op_f, V_f)$ with

- $L_f \subseteq PC, \quad E_f \subseteq L_f \times L_f, \quad l_f^0 \in L_f$
- $Op_f : E_f \to Ops$
- local variables $V_f \subseteq Lvals$ with
  - $X_f \subseteq V_f$ set of formal parameters
  - $r_f \in V_f$ variable storing return value

- A program is a set of CFAs $\mathbb{P} = \{C_{f_0}, \ldots, C_{f_k}\} \ni C_{main}$
- $PC = \bigcup\{L_f \mid C_f \in \mathbb{P}\}$

Traces

- a **command** is a pair $(op, pc) \in Ops \times PC$, written $(op_i : pc_i)$

Traces

- a **command** is a pair $(op, pc) \in Ops \times PC$, written $(op_i : pc_i)$
- a **trace** of $\mathbb{P}$ is a sequence $(op_1 : pc_1); \ldots ; (op_n : pc_n)$ such that
  - $Op(l^0_{main}, pc_1) = op_1$
  - function calls and returns are properly matched
  - operations remaining within a function body respect paths in the corresponding CFA

Strongest Postcondition and Predicate Abstraction

- Given
  - a set of states represented by a formula φ in *FOL*
  - an operation *op*
  - a set of atomic predicates $P \subseteq FOL$

Strongest Postcondition and Predicate Abstraction

- Given
  - a set of states represented by a formula φ in *FOL*
  - an operation *op*
  - a set of atomic predicates $P \subseteq FOL$
- the **strongest postcondition** of φ w.r.t. to *op*, written $SP.\varphi.op$ is the set of states reachable from states in φ after executing *op*. It gives the concrete semantics of a trace.

Strongest Postcondition and Predicate Abstraction

- Given
  - a set of states represented by a formula φ in *FOL*
  - an operation *op*
  - a set of atomic predicates $P \subseteq FOL$
- the **strongest postcondition** of φ w.r.t. to *op*, written $SP.\varphi.op$ is the set of states reachable from states in φ after executing *op*. It gives the concrete semantics of a trace.
- The **predicate abstraction** of φ w.r.t. P is the strongest formula ψ such that
  - ψ is a boolean combination of predicates in P and
  - φ implies ψ

Strongest Postcondition and Predicate Abstraction

- Let $\Pi : PC \rightarrow 2^{FOL}$ be a mapping of program locations to sets of atomic predicates

Strongest Postcondition and Predicate Abstraction

- Let $\Pi : PC \rightarrow 2^{FOL}$ be a mapping of program locations to sets of atomic predicates
- SP_Π is the abstraction of of SP w.r.t Π:
  - if φ denotes a set of states,
  - $(op : pc)$ is a command,
  - then $SP_\Pi.\varphi.(op : pc)$ is the predicate abstraction of $SP.\varphi.op$ w.r.t $\Pi.pc$

Strongest Postcondition and Predicate Abstraction

- Let $\Pi : PC \rightarrow 2^{FOL}$ be a mapping of program locations to sets of atomic predicates
- SP_Π is the abstraction of of SP w.r.t Π:
  - if φ denotes a set of states,
  - $(op : pc)$ is a command,
  - then $SP_\Pi.\varphi.(op : pc)$ is the predicate abstraction of $SP.\varphi.op$ w.r.t $\Pi.pc$
- a trace t is SP-feasible if $SP.true.t$ is satisfiable

Strongest Postcondition and Predicate Abstraction

- Let $\Pi : PC \rightarrow 2^{FOL}$ be a mapping of program locations to sets of atomic predicates
- SP_{Π} is the abstraction of of SP w.r.t Π:
  - if φ denotes a set of states,
  - $(op : pc)$ is a command,
  - then $SP_{\Pi}.\varphi.(op : pc)$ is the predicate abstraction of $SP.\varphi.op$ w.r.t $\Pi.pc$
- a trace t is SP-feasible if $SP.true.t$ is satisfiable
- a trace t is SP_{Π}-feasible if $SP_{\Pi}.true.t$ is satisfiable

Overview

- Goal: an algorithm *Extract* that takes a trace t and returns a mapping Π such that:
  - t is SP-infeasible iff t is SP_{Π}-infeasible for $\Pi = Extract.t$

Overview

- Goal: an algorithm *Extract* that takes a trace t and returns a mapping Π such that:
  - t is *SP*-infeasible iff t is SP_{Π}-infeasible for $\Pi = Extract.t$
- We will consider classes of programs of increasing complexity:
  - **PI**: flat and pointer-free
  - **PII**: pointer-free programs
  - **PIII**: flat programs with pointers
  - **PIV**: general programs (pointers and function calls)

Basic Definitions

We need the following notions:

- an **lvalue map** is a function $\theta \in \mathit{Lvm} = \mathit{Lvals} \to \mathbb{N}$

Basic Definitions

We need the following notions:

- an **lvalue map** is a function $\theta \in Lvm = Lvals \rightarrow \mathbb{N}$
  - the operator **Upd** : $Lvm \rightarrow 2^{Lvals} \rightarrow Lvm$ is defined as follows:

$$(Upd.\theta.L).l = \begin{cases} \theta.l & \text{if } l \notin L, \\ i_l & \text{otherwise.} \end{cases}$$

with fresh integer i_l

Basic Definitions

We need the following notions:

- an **lvalue map** is a function $\theta \in Lvm = Lvals \rightarrow \mathbb{N}$
  - the operator **Upd** : $Lvm \rightarrow 2^{Lvals} \rightarrow Lvm$ is defined as follows:

    $$(Upd.\theta.L).l = \begin{cases} \theta.l & \text{if } l \notin L, \\ i_l & \text{otherwise.} \end{cases}$$

    with fresh integer i_l
- **Sub**$.\theta.l = \langle l, \theta.l \rangle$
- **Clean**$.\langle l, i \rangle = l$

Basic Definitions

We need the following notions:

- an **lvalue map** is a function $\theta \in Lvm = Lvals \to \mathbb{N}$
  - the operator **Upd** : $Lvm \to 2^{Lvals} \to Lvm$ is defined as follows:

    $$(Upd.\theta.L).l = \begin{cases} \theta.l & \text{if } l \notin L, \\ i_l & \text{otherwise.} \end{cases}$$

    with fresh integer i_l
- **Sub**$.\theta.l = \langle l, \theta.l \rangle$
- **Clean**$.\langle l, i \rangle = l$
- a **constraint map** is a function $\Gamma : \mathbb{N} \to FOL$ that maps each point i of trace t to a constraint $\Gamma.i$

Basic Definitions

We need the following notions:

- an **lvalue map** is a function $\theta \in Lvm = Lvals \rightarrow \mathbb{N}$
  - the operator **Upd** : $Lvm \rightarrow 2^{Lvals} \rightarrow Lvm$ is defined as follows:

    $$(Upd.\theta.L).l = \begin{cases} \theta.l & \text{if } l \notin L, \\ i_l & \text{otherwise.} \end{cases}$$

    with fresh integer i_l
- **Sub**$.\theta.l = \langle l, \theta.l \rangle$
- **Clean**$.\langle l, i \rangle = l$
- a **constraint map** is a function $\Gamma : \mathbb{N} \rightarrow FOL$ that maps each point i of trace t to a constraint $\Gamma.i$
- $\bigwedge_{1 \leq i \leq n} \Gamma.i$ is the **trace formula** (TF) of trace $t = (op_1 : pc_1); \ldots; (op_n : pc_n)$
  The trace is feasible iff the TF is satisfiable.

Basic Definitions

For each class of programs, we will define

- an operator **Con**, with $\mathit{Con}.(\theta,\Gamma).t = (\theta',\Gamma')$ that updates the lvalue map θ and computes a new constraint map Γ' given a trace t

Algorithm *Extract*

Idea: t is *SP*-infeasible iff t is SP_{Π}-infeasible for $\Pi = \mathit{Extract}.t$

Algorithm *Extract*

Idea: t is SP-infeasible iff t is SP_{Π}-infeasible for $\Pi = \mathit{Extract}.t$
Sketch of algorithm:

Algorithm 1

Input: infeasible trace $t = (op_1 : pc_1); \ldots; (op_n : pc_n)$
Output: a map $\Pi : PC \to FOL$

Algorithm *Extract*

Idea: t is SP-infeasible iff t is SP_Π-infeasible for $\Pi = Extract.t$
Sketch of algorithm:

Algorithm 1

Input: infeasible trace $t = (op_1 : pc_1); \ldots; (op_n : pc_n)$
Output: a map $\Pi : PC \to FOL$
$\Pi.pc_i := \emptyset$ *for* $1 \leq i \leq n$
$(\cdot, \Gamma) := Con.(\theta_0, \Gamma_0).t$
$\mathcal{P} :=$ *derivation of* $\bigwedge_{1 \leq i \leq n} \Gamma.i \vdash \bot$

Algorithm *Extract*

Idea: t is SP-infeasible iff t is SP_{Π}-infeasible for $\Pi = Extract.t$
Sketch of algorithm:

Algorithm 1

Input: infeasible trace $t = (op_1 : pc_1); \ldots ; (op_n : pc_n)$
Output: a map $\Pi : PC \rightarrow FOL$

$\Pi.pc_i := \emptyset$ *for* $1 \leq i \leq n$
$(\cdot, \Gamma) := Con.(\theta_0, \Gamma_0).t$
$\mathcal{P} :=$ *derivation of* $\bigwedge_{1 \leq i \leq n} \Gamma.i \vdash \bot$
for $i := 1$ **to** n **do**
 $\varphi^- := \bigwedge_{1 \leq j \leq i} \Gamma.j$
 $\varphi^+ := \bigwedge_{i+1 \leq j \leq n} \Gamma.j$
 $\psi := \text{ITP}.(\varphi^-, \varphi^+).\mathcal{P}$
 $\Pi.pc_i := \Pi.pc_i \cup Atoms.(Clean.\psi)$

Algorithm *Extract*

Idea: t is SP-infeasible iff t is SP_Π-infeasible for $\Pi = Extract.t$
Sketch of algorithm:

Algorithm 1

Input: infeasible trace $t = (op_1 : pc_1);\ldots;(op_n : pc_n)$
Output: a map $\Pi : PC \rightarrow FOL$

$\Pi.pc_i := \emptyset$ *for* $1 \leq i \leq n$
$(\cdot, \Gamma) := Con.(\theta_0, \Gamma_0).t$
$\mathcal{P} :=$ *derivation of* $\bigwedge_{1\leq i\leq n} \Gamma.i \vdash \bot$
for $i := 1$ **to** n **do**
  $\varphi^- := \bigwedge_{1\leq j\leq i} \Gamma.j$
  $\varphi^+ := \bigwedge_{i+1\leq j\leq n} \Gamma.j$
  $\psi := \text{ITP}.(\varphi^-, \varphi^+).\mathcal{P}$
  $\Pi.pc_i := \Pi.pc_i \cup Atoms.(Clean.\psi)$
return Π.

Algorithm *Extract*

Example

trace t	$Con.(\theta_0, \Gamma_0).t$	
$(\text{assume}(b > 0) : pc_1);$	$\langle b, 0\rangle > 0$	φ^-
$(c := 2 * b : pc_2);$	$\langle c, 1\rangle = 2 * \langle b, 0\rangle$	
$(a := b : pc_3);$	$\langle a, 2\rangle = \langle b, 0\rangle$	
$(a := a - 1 : pc_4);$	$\langle a, 3\rangle = \langle a, 2\rangle - 1$	
$(\text{assume}(a < b) : pc_5);$	$\langle a, 3\rangle < \langle b, 0\rangle$	φ^+
$(\text{assume}(a = c) : pc_6)$	$\langle a, 3\rangle = \langle c, 1\rangle$	

Algorithm *Extract*

Example

trace t	$Con.(\theta_0, \Gamma_0).t$	
$(\text{assume}(b > 0) : pc_1);$	$\langle b, 0\rangle > 0$	φ^-
$(c := 2 * b : pc_2);$	$\langle c, 1\rangle = 2 * \langle b, 0\rangle$	
$(a := b : pc_3);$	$\langle a, 2\rangle = \langle b, 0\rangle$	
$(a := a - 1 : pc_4);$	$\langle a, 3\rangle = \langle a, 2\rangle - 1$	
$(\text{assume}(a < b) : pc_5);$	$\langle a, 3\rangle < \langle b, 0\rangle$	φ^+
$(\text{assume}(a = c) : pc_6)$	$\langle a, 3\rangle = \langle c, 1\rangle$	

- $\psi = \langle a, 3\rangle \leq \langle c, 1\rangle - 2$

Algorithm *Extract*

Example

trace t	$Con.(\theta_0, \Gamma_0).t$	
$(\text{assume}(b > 0) : pc_1);$	$\langle b, 0\rangle > 0$	φ^-
$(c := 2 * b : pc_2);$	$\langle c, 1\rangle = 2 * \langle b, 0\rangle$	
$(a := b : pc_3);$	$\langle a, 2\rangle = \langle b, 0\rangle$	
$(a := a - 1 : pc_4);$	$\langle a, 3\rangle = \langle a, 2\rangle - 1$	
$(\text{assume}(a < b) : pc_5);$	$\langle a, 3\rangle < \langle b, 0\rangle$	φ^+
$(\text{assume}(a = c) : pc_6)$	$\langle a, 3\rangle = \langle c, 1\rangle$	

- $\psi = \langle a, 3\rangle \leq \langle c, 1\rangle - 2$
- cleaned: $a \leq c - 2$

PI programs

Trace

$(x := e : pc_i)$

$SP.\varphi.t$:

$SP_\Pi.\varphi.t$:

$Con.(\theta, \Gamma).t$:

PI programs

Trace

$(x := e : pc_i)$

$SP.\varphi.t$: $\exists x'.(\varphi[x'/x] \wedge x = e[x'/x])$
where x' is a fresh variable

$SP_{\Pi}.\varphi.t$:

$Con.(\theta, \Gamma).t$:

PI programs

Trace

$(x := e : pc_i)$

$SP.\varphi.t$: $\exists x'.(\varphi[x'/x] \wedge x = e[x'/x])$
where x' is a fresh variable

$SP_\Pi.\varphi.t$: $\alpha(\Pi.pc).(SP.\varphi.t)$

$Con.(\theta, \Gamma).t$:

PI programs

Trace

$(x := e : pc_i)$

$SP.\varphi.t$: $\exists x'.(\varphi[x'/x] \wedge x = e[x'/x])$
where x' is a fresh variable

$SP_{\Pi}.\varphi.t$: $\alpha(\Pi.pc).(SP.\varphi.t)$

$Con.(\theta, \Gamma).t$: $(\theta', \Gamma[i \mapsto (Sub.\theta'.x = Sub.\theta.e)])$
where $\theta' = Upd.\theta.\{x\}$

PI programs

Trace

$(\text{assume}(p) : pc_i)$

$SP.\varphi.t$:

$SP_\Pi.\varphi.t$:

$Con.(\theta, \Gamma).t$:

PI programs

Trace

(assume(p) : pc_i)

$SP.\varphi.t$: $\varphi \wedge p$

$SP_{\Pi}.\varphi.t$:

$Con.(\theta, \Gamma).t$:

PI programs

Trace

(assume(p) : pc_i)

$SP.\varphi.t$: $\varphi \wedge p$

$SP_\Pi.\varphi.t$: $\alpha(\Pi.pc).(SP.\varphi.t)$

$Con.(\theta, \Gamma).t$:

PI programs

Trace

(assume(p) : pc_i)

$SP.\varphi.t$: $\varphi \wedge p$

$SP_\Pi.\varphi.t$: $\alpha(\Pi.pc).(SP.\varphi.t)$

$Con.(\theta, \Gamma).t$: $(\theta, \Gamma[i \mapsto Sub.\theta.p])$

PI programs

Trace

$t_1 ; t_2$

$SP.\varphi.t$:

$SP_\Pi.\varphi.t$:

$Con.(\theta, \Gamma).t$:

PI programs

Trace

$t_1\,;\,t_2$

$SP.\varphi.t$: $SP.(SP.\varphi.t_1).t_2$

$SP_\Pi.\varphi.t$:

$Con.(\theta,\Gamma).t$:

PI programs

Trace

$t_1 ; t_2$

$SP.\varphi.t$:	$SP.(SP.\varphi.t_1).t_2$
$SP_\Pi.\varphi.t$:	$SP_\Pi.(SP_\Pi.\varphi.t_1).t_2$
$Con.(\theta, \Gamma).t$:	

PI programs

Trace

$t_1 \,;\, t_2$

$SP.\varphi.t$:	$SP.(SP.\varphi.t_1).t_2$
$SP_\Pi.\varphi.t$:	$SP_\Pi.(SP_\Pi.\varphi.t_1).t_2$
$Con.(\theta,\Gamma).t$:	$Con.(Con.(\theta,\Gamma).t_1).t_2$

Handling Function Calls

We want to abstract functions independently of their calling contexts.
We use **polymorphic abstraction**

Handling Function Calls

We want to abstract functions independently of their calling contexts.
We use **polymorphic abstraction**

Polymorphic abstraction

- for every formal parameter x, a function has a new local symbolic variable ϕ_x

Handling Function Calls

We want to abstract functions independently of their calling contexts.
We use **polymorphic abstraction**

Polymorphic abstraction

- for every formal parameter x, a function has a new local symbolic variable ϕ_x
- ϕ_x holds the value of actual parameter for x

Handling Function Calls

We want to abstract functions independently of their calling contexts.
We use **polymorphic abstraction**

Polymorphic abstraction

- for every formal parameter x, a function has a new local symbolic variable ϕ_x
- ϕ_x holds the value of actual parameter for x
- ϕ_x is never written to

Handling Function Calls

We want to abstract functions independently of their calling contexts.
We use **polymorphic abstraction**

Polymorphic abstraction

- for every formal parameter x, a function has a new local symbolic variable ϕ_x
- ϕ_x holds the value of actual parameter for x
- ϕ_x is never written to
- sym_f denotes the set of all symbolic variables of f

Handling Function Calls

We want to abstract functions independently of their calling contexts.
We use **polymorphic abstraction**

Polymorphic abstraction

- for every formal parameter x, a function has a new local symbolic variable ϕ_x
- ϕ_x holds the value of actual parameter for x
- ϕ_x is never written to
- sym_f denotes the set of all symbolic variables of f

This allows us to express the effect of f in terms of its arguments, at the call site.

PII programs

Trace

$$(y := f(\vec{e}) : pc_i);$$
$$t_1;$$
$$(\text{return} : pc_j)$$

$SP.\varphi.t$:

PII programs

Trace

$$(y := f(\vec{e}) : pc_i);$$
$$t_1;$$
$$(\text{return} : pc_j)$$

$$\begin{aligned} SP.\varphi.t: \quad & \exists y', \vec{\phi}.\varphi[y'/y] \wedge \\ & \exists r.\vec{\phi} = \vec{e}[y'/y] \wedge \\ & \exists V_f^-.SP.true.t_1 \wedge \\ & y = r \end{aligned}$$

with fresh y', formals $\vec{x}$, return var r, $V_f^- = V_f \backslash (sym_f \cup \{r\})$

PII programs

Trace

$$(y := f(\vec{e}) : pc_i);$$
$$t_1;$$
$$(\text{return} : pc_j)$$

$$SP_\Pi.\varphi.t: \quad \alpha.(\Pi.pc_j).\\ (\exists y', \vec{\phi}.\varphi[y'/y] \wedge\\ \exists r.\vec{\phi} = \vec{e}[y'/y] \wedge\\ \exists V_f^-.Sp_\Pi.true.t_1 \wedge\\ y = r)$$

PII programs

Trace

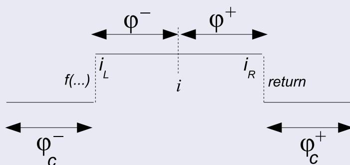
$$(y := f(\vec{e}) : pc_i);$$
$$t_1;$$
$$(\text{return} : pc_j)$$

$Con.(\theta, \Gamma).t$: (θ', Γ')
where

$$\theta' = Upd.\theta.\{y\}$$
$$\theta_I = Upd.\theta'.V_f$$
$$\Gamma_I = \Gamma[i \mapsto Sub.\theta_I.\vec{\phi} = Sub.\theta_I.\vec{e}]$$
$$(\theta_O, \Gamma_O) = Con.(\theta_I, \Gamma_I).t_1$$
$$\Gamma' = \Gamma_O[j \mapsto Sub.\theta'.y = Sub.\theta_O.r]$$

Getting Well-Scoped Predicates

Scope of symbols

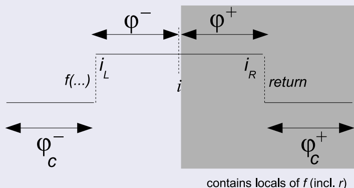


Getting Well-Scoped Predicates

Scope of symbols

Getting Well-Scoped Predicates

Scope of symbols



To get predicates that contain only symbols that are in scope at location i, we construct the interpolant of $(\varphi^-, \varphi^+ \wedge \varphi_C^- \wedge \varphi_C^+)$

Getting Well-Scoped Predicates

We need to modify our algorithm *Extract*:

Getting Well-Scoped Predicates

We need to modify our algorithm *Extract*:

Algorithm 2

Input: infeasible trace $t = (op_1 : pc_1); \ldots ; (op_n : pc_n)$
Output: a map $\Pi : PC \rightarrow FOL$

Getting Well-Scoped Predicates

We need to modify our algorithm *Extract*:

Algorithm 2

Input: infeasible trace $t = (op_1 : pc_1); \ldots ; (op_n : pc_n)$
Output: a map $\Pi : PC \to FOL$

$\Pi.pc_i := \emptyset$ *for* $1 \leq i \leq n$
$(\cdot, \Gamma) := Con.(\theta_0, \Gamma_0).t$
$\mathcal{P} :=$ *derivation of* $\bigwedge_{1 \leq i \leq n} \Gamma.i \vdash \bot$

Getting Well-Scoped Predicates

We need to modify our algorithm *Extract*:

Algorithm 2

Input: infeasible trace $t = (op_1 : pc_1); \ldots ; (op_n : pc_n)$
Output: a map $\Pi : PC \rightarrow FOL$

$\Pi.pc_i := \emptyset$ *for* $1 \leq i \leq n$
$(\cdot, \Gamma) := Con.(\theta_0, \Gamma_0).t$
$\mathcal{P} :=$ *derivation of* $\bigwedge_{1 \leq i \leq n} \Gamma.i \vdash \bot$
for $i := 1$ **to** n **do**
$\varphi^- := \bigwedge_{i_L+1 \leq j \leq i} \Gamma.j$
$\varphi^+ := \bigwedge_{i+1 \leq j \leq i_R-1} \Gamma.j$
$\varphi_C := (\bigwedge_{1 \leq j \leq i_L} \Gamma.j) \wedge (\bigwedge_{i_R \leq j \leq n} \Gamma.j)$
$\psi := \text{ITP}.(\varphi^-, \varphi^+ \wedge \varphi_C).\mathcal{P}$
$\Pi.pc_i := \Pi.pc_i \cup Atoms.(Clean.\psi)$

Getting Well-Scoped Predicates

We need to modify our algorithm *Extract*:

Algorithm 2

Input: infeasible trace $t = (op_1 : pc_1); \ldots; (op_n : pc_n)$
Output: a map $\Pi : PC \rightarrow FOL$

$\Pi.pc_i := \emptyset$ *for* $1 \leq i \leq n$
$(\cdot, \Gamma) := Con.(\theta_0, \Gamma_0).t$
$\mathcal{P} :=$ *derivation of* $\bigwedge_{1 \leq i \leq n} \Gamma.i \vdash \bot$
for $i := 1$ **to** n **do**
$\varphi^- := \bigwedge_{i_L+1 \leq j \leq i} \Gamma.j$
$\varphi^+ := \bigwedge_{i+1 \leq j \leq i_R-1} \Gamma.j$
$\varphi_C := (\bigwedge_{1 \leq j \leq i_L} \Gamma.j) \wedge (\bigwedge_{i_R \leq j \leq n} \Gamma.j)$
$\psi := \text{ITP}.(\varphi^-, \varphi^+ \wedge \varphi_C).\mathcal{P}$
$\Pi.pc_i := \Pi.pc_i \cup Atoms.(Clean.\psi)$
return Π.

Handling Pointers

- Theory of arrays defines functions **sel**, **upd** and the axiom

$$sel(upd(M, a, v), b) = \begin{cases} v & \text{if } a = b, \\ sel(M, b) & \text{otherwise.} \end{cases}$$

Handling Pointers

- Theory of arrays defines functions **sel**, **upd** and the axiom

$$sel(upd(M, a, v), b) = \begin{cases} v & \text{if } a = b, \\ sel(M, b) & \text{otherwise.} \end{cases}$$

- Given a memory variable M, a variable x and an lvalue $*l$, define
  - $M.x = sel(M, x)$
  - $M.(*l) = sel(M, M.l)$

Handling Pointers

- Theory of arrays defines functions **sel**, **upd** and the axiom

$$sel(upd(M, a, v), b) = \begin{cases} v & \text{if } a = b, \\ sel(M, b) & \text{otherwise.} \end{cases}$$

- Given a memory variable M, a variable x and an lvalue $*l$, define
  - $M.x = sel(M, x)$
  - $M.(*l) = sel(M, M.l)$
- We use M as a map from lvalues to memory expressions over M.

Some Definitions

- Given a set X of variables, define
  - **Reach.X** $= \{*^k x \mid x \in X \textit{ and } k \geq 0\}$

Some Definitions

- Given a set X of variables, define
  - **Reach.X** $= \{*^k x \mid x \in X \text{ and } k \geq 0\}$
  - k is bounded by the type of x, thus *Reach.X* is always finite.

Some Definitions

- Given a set X of variables, define
  - **Reach.X** $= \{*^k x \mid x \in X \textit{ and } k \geq 0\}$
  - k is bounded by the type of x, thus *Reach.X* is always finite.
- For a function f, we redefine the set sym_f:
  - $\mathbf{sym_f} = \{\phi_l \mid l \in Reach.X_f\}$

Some Definitions

- Given a set X of variables, define
  - **Reach.X** $= \{*^k x \mid x \in X \text{ and } k \geq 0\}$
  - k is bounded by the type of x, thus *Reach.X* is always finite.
- For a function f, we redefine the set sym_f:
  - $\mathbf{sym_f} = \{\phi_l \mid l \in Reach.X_f\}$
- At the beginning of each function f, add `assume`-operations such that
  - each lvalue x $(*x \ldots)$ reachable from the formals reads its value from symbolic variable ϕ_x $(\phi_{*x} \ldots)$:

Some Definitions

- Given a set X of variables, define
  - **Reach.X** $= \{*^k x \mid x \in X \ and \ k \geq 0\}$
  - k is bounded by the type of x, thus $Reach.X$ is always finite.
- For a function f, we redefine the set sym_f:
  - $\mathbf{sym_f} = \{\phi_l \mid l \in Reach.X_f\}$
- At the beginning of each function f, add `assume`-operations such that
  - each lvalue x ($*x \ldots$) reachable from the formals reads its value from symbolic variable ϕ_x ($\phi_{*x} \ldots$):
  - `assume`$(\bigwedge_{l \in Reach.X_f} l = \phi_l)$

PIII and PIV programs

- A function may now be passed a pointer into the local memory of the caller

PIII and PIV programs

- A function may now be passed a pointer into the local memory of the caller
- Polymorphic abstraction requires to summarize all effects that the callee may have had on the caller's store at the point of return

PIII and PIV programs

- A function may now be passed a pointer into the local memory of the caller
- Polymorphic abstraction requires to summarize all effects that the callee may have had on the caller's store at the point of return
- Solution:
  - the callee starts with a copy of the caller's store

PIII and PIV programs

- A function may now be passed a pointer into the local memory of the caller
- Polymorphic abstraction requires to summarize all effects that the callee may have had on the caller's store at the point of return
- Solution:
  - the callee starts with a copy of the caller's store
  - upon return, the caller refreshes his store using the callee's store

PIII and PIV programs

- A function may now be passed a pointer into the local memory of the caller
- Polymorphic abstraction requires to summarize all effects that the callee may have had on the caller's store at the point of return
- Solution:
  - the callee starts with a copy of the caller's store
  - upon return, the caller refreshes his store using the callee's store
- No explicit return variable, instead update some cell passed as argument

PIII and PIV programs

- We require that every cell of the caller which is unreachable from the parameters remains unchanged after a call.

PIII and PIV programs

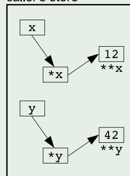
- We require that every cell of the caller which is unreachable from the parameters remains unchanged after a call.
- Upon return, we copy into the caller's store the contents of the callee's store that are reachable from the passed parameters.

PIII and PIV programs

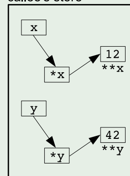
Example

Function call:

caller's store



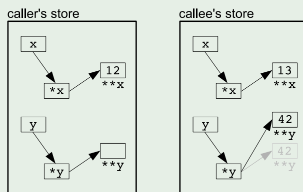
callee's store



PIII and PIV programs

Example

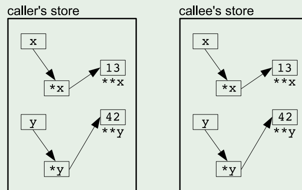
Upon return:



PIII and PIV programs

Example

Refresh caller’s store:



PIII and PIV programs

Example

$(*y := 0 : pc_1);$ $\langle *\langle y, 0\rangle, 1\rangle = 0$

PIII and PIV programs

Example

$(*y := 0 : pc_1);$
$(\text{inc}(y) : pc_2);$

$\langle *\langle y, 0\rangle, 1\rangle = 0$
$\langle \phi_x, 1\rangle = \langle y, 0\rangle \wedge \langle \phi_{*x}, 1\rangle = \langle *\langle y, 0\rangle, 1\rangle$

PIII and PIV programs

Example

$(*y := 0 : pc_1);$
$(\mathrm{inc}(y) : pc_2);$
$\quad(assume(x = \phi_x$
$\quad\ \wedge * x = \phi_{*x}) : pc_3)$

$\langle *\langle y, 0\rangle, 1\rangle = 0$
$\langle \phi_x, 1\rangle = \langle y, 0\rangle \wedge \langle \phi_{*x}, 1\rangle = \langle *\langle y, 0\rangle, 1\rangle$
$\quad\langle x, 4\rangle = \langle \phi_x, 1\rangle$
$\quad\wedge\langle *\langle x, 4\rangle, 5\rangle = \langle *\langle \phi_x, 1\rangle, 7\rangle$
$\quad\wedge\langle *\langle x, 4\rangle, 5\rangle = \langle \phi_{*x}, 2\rangle$

PIII and PIV programs

Example

$(*y := 0 : pc_1);$
$(\text{inc}(y) : pc_2);$
$\quad(\mathit{assume}(x = \phi_x$
$\quad\ \wedge * x = \phi_{*x}) : pc_3)$

$\quad(*x := *x + 1 : pc_4);$

$\langle *\langle y, 0\rangle, 1\rangle = 0$
$\langle \phi_x, 1\rangle = \langle y, 0\rangle \wedge \langle \phi_{*x}, 1\rangle = \langle *\langle y, 0\rangle, 1\rangle$
$\quad\langle x, 4\rangle = \langle \phi_x, 1\rangle$
$\quad\wedge\langle *\langle x, 4\rangle, 5\rangle = \langle *\langle \phi_x, 1\rangle, 7\rangle$
$\quad\wedge\langle *\langle x, 4\rangle, 5\rangle = \langle \phi_{*x}, 2\rangle$
$\langle *\langle x, 4\rangle, 6\rangle = \langle *\langle x, 4\rangle, 5\rangle + 1$
$\quad \mathit{ite}.(\langle x, 4\rangle = \langle \phi_x, 1\rangle)$
$\qquad .(\langle *\langle \phi_x, 1\rangle, 8\rangle = \langle *\langle x, 4\rangle, 5\rangle + 1)$
$\qquad .(\langle *\langle \phi_x, 1\rangle, 8\rangle = \langle *\langle \phi_x, 1\rangle, 7\rangle)$

PIII and PIV programs

Example

Program	Constraints
$(*y := 0 : pc_1)$;	$\langle *\langle y, 0\rangle, 1\rangle = 0$
$(\mathrm{inc}(y) : pc_2)$;	$\langle \phi_x, 1\rangle = \langle y, 0\rangle \wedge \langle \phi_{*x}, 1\rangle = \langle *\langle y, 0\rangle, 1\rangle$
$(assume(x = \phi_x$	$\langle x, 4\rangle = \langle \phi_x, 1\rangle$
$\wedge * x = \phi_{*x}) : pc_3)$	$\wedge\langle *\langle x, 4\rangle, 5\rangle = \langle *\langle \phi_x, 1\rangle, 7\rangle$
	$\wedge\langle *\langle x, 4\rangle, 5\rangle = \langle \phi_{*x}, 2\rangle$
$(*x := *x + 1 : pc_4)$;	$\langle *\langle x, 4\rangle, 6\rangle = \langle *\langle x, 4\rangle, 5\rangle + 1$
	$ite.(\langle x, 4\rangle = \langle \phi_x, 1\rangle)$
	$.(\langle *\langle \phi_x, 1\rangle, 8\rangle = \langle *\langle x, 4\rangle, 5\rangle + 1)$
	$.(\langle *\langle \phi_x, 1\rangle, 8\rangle = \langle *\langle \phi_x, 1\rangle, 7\rangle)$
$(return : pc_5)$;	$ite.(\langle y, 0\rangle = \langle \phi_x, 1\rangle)$
	$.(\langle *\langle y, 0\rangle, 9\rangle = \langle *\langle \phi_x, 1\rangle, 8\rangle)$
	$.(\langle *\langle y, 0\rangle, 9\rangle = \langle *\langle y, 0\rangle, 1\rangle)$

PIII and PIV programs

Example

$(*y := 0 : pc_1);$	$\langle *\langle y,0\rangle,1\rangle = 0$
$(\text{inc}(y) : pc_2);$	$\langle \phi_x,1\rangle = \langle y,0\rangle \wedge \langle \phi_{*x},1\rangle = \langle *\langle y,0\rangle,1\rangle$
$(assume(x = \phi_x$	$\langle x,4\rangle = \langle \phi_x,1\rangle$
$\wedge *x = \phi_{*x}) : pc_3)$	$\wedge\langle *\langle x,4\rangle,5\rangle = \langle *\langle \phi_x,1\rangle,7\rangle$
	$\wedge\langle *\langle x,4\rangle,5\rangle = \langle \phi_{*x},2\rangle$
$(*x := *x + 1 : pc_4);$	$\langle *\langle x,4\rangle,6\rangle = \langle *\langle x,4\rangle,5\rangle + 1$
	$ite.(\langle x,4\rangle = \langle \phi_x,1\rangle)$
	$.(\langle *\langle \phi_x,1\rangle,8\rangle = \langle *\langle x,4\rangle,5\rangle + 1)$
	$.(\langle *\langle \phi_x,1\rangle,8\rangle = \langle *\langle \phi_x,1\rangle,7\rangle)$
$(return : pc_5);$	$ite.(\langle y,0\rangle = \langle \phi_x,1\rangle)$
	$.(\langle *\langle y,0\rangle,9\rangle = \langle *\langle \phi_x,1\rangle,8\rangle)$
	$.(\langle *\langle y,0\rangle,9\rangle = \langle *\langle y,0\rangle,1\rangle)$
$(assume(*y \neq 1) : pc_6)$	$\langle *\langle y,0\rangle,9\rangle \neq 1$

Summary

- The described techniques were implemented in BLAST
- They can be invoked with option `-craig`

Thank you!

For Further Reading I

Henzinger, Jhala, Majumdar, McMillan
Abstractions from Proofs
In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, 2004*, 2(1):50–100, 2000.