tightcenter

panikzettel.htwr-aachen.de

Static Program Analysis Panikzettel

Philipp Schröer

Version 2 — 15.02.2019

Contents

1 Introduction

This Panikzettel is about the lecture Static Program Analysis by Prof. Noll held in the summer semester 2018.

This Panikzettel is Open Source. We appreciate comments and suggestions at `https://git.rwth-aachen.de/philipp.schroer/panikzettel`.

2 Dataflow Analysis

Dataflow analysis is a form of program analysis that is based on flows of information through the analyzed program. There are several ways to distinguish program analyses: Dependence on statement orders (*flow-sensitivity*), the flow direction (*forward*/*backward*), the quantification of flows over paths (*may*, i.e. union, or *must*, i.e. intersection), and the scope w.r.t. procedures (*interprocedural* vs. *intraprocedural*).

2.1 The WHILE Language

We describe flows between labelled statements. We use WHILE programs as our example language. Here, dataflow information is associated with `skip` statements, assignments and tests in `if` and `while` statements.

Definition: Labelled WHILE programs

$$\begin{array}{lll}
\texttt{a} & ::= & z \mid x \mid a_1{+}a_2 \mid a_1{-}a_2 \mid a_1{*}a_2 \in \text{AExp} \\
\texttt{b} & ::= & t \mid a_1{=}a_2 \mid a_1{>}a_2 \mid \neg\ b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in \text{BExp} \\
\texttt{c} & ::= & [\texttt{skip}]^l \mid [x \texttt{ := } a]^l \mid c_1\texttt{;}c_2 \mid \\
 & & \texttt{if } [b]^l \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end} \mid \\
 & & \texttt{while } [b]^l \texttt{ do } c \texttt{ end} \in \text{Cmd}
\end{array}$$

We assume all labels in $c \in$ `Cmd` are distinct, and denoted by Lab_c.

Labelled fragments of c are called *blocks*, written as Blk_c.

2.2 Algebraic Foundations

Definition: Partial Order

A *partial order* $(D, \sqsubseteq)$ has a domain D and a relation $\sqsubseteq \subseteq D \times D$, where $\forall d_1, d_2, d_3 \in D$:

Reflexivity
$d_1 \sqsubseteq d_1$

Transitivty
$d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_3 \;\Rightarrow\; d_1 \sqsubseteq s_3$

Antisymmetry
$d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_1 \;\Rightarrow\; d_1 = d_2$

To combine information in our dataflow systems, we use *complete lattices*. A complete lattice is a partial order with upper bounds for all subsets (also called *supremum* or *join*[a]). Equivalently, one can require lower bounds for all subsets (also called *infimum* or *meet*[b]).

Definition: (Least) Upper Bound, (Greatest) Lower Bound

Let $(D, \sqsubseteq)$ be a partial order with $S \subseteq D$.

1. $d \in D$ is an *upper bound* of S $(S \sqsubseteq d)$ if $s \sqsubseteq d \;\; \forall s \in S$.
2. d is a *least upper bound of* S $(d = \bigsqcup S)$ if $d \sqsubseteq d'$ for every upper bound d' of S.

Analogous definitions for *lower bound* and *greatest lower bound*.

Definition: Complete Lattice

A *complete lattice* is a

- partial order $(D, \sqsubseteq)$,
- such that all $S \subseteq D$ have least upper bounds, or equivalently, greatest lower bounds.

The least element is $\bot := \bigsqcup \emptyset$.
The greatest element is $\top := \bigsqcap \emptyset$.

The *Ascending Chain Condition (ACC)* guarantees termination of the fixpoint algorithm by limiting ascending chains to be finite.

For example, $(\mathbb{N}, \leq)$ does not fulfill ACC because e.g. $1, 2, \ldots$ is an ascending chain that does not stabilise.

Definition: Ascending Chain Condition

A partial order $(D, \sqsubseteq)$ satisfies *ACC* if each ascending chain eventually stabilises: $\exists n \in \mathbb{N} : d_n = d_{n+1} = \ldots$.

Chain: A subset $S \subseteq D$ with $\forall d_1, d_2 \in S$: $d_1 \sqsubseteq d_2$ or $d_2 \sqsubseteq d_1$. *Ascending chain*: $d_i \sqsubseteq d_{i+1} \quad \forall i \in \mathbb{N}$.

Definition: Monotonicity

Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be partial orders. $\Phi : D \to D'$ is *monotonic* if $\forall d_1, d_2 \in D$: $d_1 \sqsubseteq d_2 \Rightarrow \Phi(d_1) \sqsubseteq' \Phi(d_2)$.

Definition: Fixpoint

d is a *fixpoint* of $\Phi : D \to D$ if $\Phi(d) = d$.

Theorem: Fixpoint Theorem by Kleene

Let $(D, \sqsubseteq)$ be a complete lattice satisfying ACC and $\Phi : D \to D$ monotonic. Then

$$\mathrm{fix}(\Phi) := \bigsqcup \left\{ \Phi^k(\bot) \;\middle|\; k \in \mathbb{N} \right\}$$

is the *least fixpoint* of Φ where

$$\begin{aligned} \Phi^0(d) &:= d \\ \Phi^{k+1}(d) &:= \Phi(\Phi^k(d)) \end{aligned}$$

[a] from set theory, the union is what you get when you join sets

[b] from set theory, the intersection is where two sets meet

2.3 Dataflow System

Definition: Initial and Final Labels

$$\text{init} : \text{Cmd} \to \text{Lab}$$
$$\text{final} : \text{Cmd} \to 2^{\text{Lab}}$$

Definition: Flow Relation

$$\text{flow}(c) \subseteq \text{Lab} \times \text{Lab}$$

$\text{flow}^R(c) := \{ (l', l) \mid (l, l') \in \text{flow}(c) \}$

$c \in \text{Cmd}$	$\text{init}(c)$	$\text{final}(c)$	$\text{flow}(c)$
`[skip]`^{l}	l	$\{ l \}$	$\emptyset$
`[`x `:=` a`]`^{l}	l	$\{ l \}$	$\emptyset$
c_1`;`c_2	$\text{init}(c_1)$	$\text{final}(c_2)$	$\text{flow}(c_1) \cup \text{flow}(c_2) \cup \text{final}(c_1) \times \{ \text{init}(c_2) \}$
`if [`b`]`^{l} `then` c_1 `else` c_2 `end`	l	$\text{final}(c_1) \cup \text{final}(c_2)$	$\text{flow}(c_1) \cup \text{flow}(c_2) \cup \{ (l, \text{init}(c_1)), (l, \text{init}(c_2)) \}$
`while [`b`]`^{l} `do` c `end`	l	$\{ l \}$	$\text{flow}(c) \cup \{ (l, \text{init}(c) \} \cup \text{final}(c) \times \{ l \}$

A *dataflow system* describes how program analysis data flows between statements in our labelled program. If we are doing a forward analysis, we choose $\{ \text{init}(c) \}$ as the *extremal labels* to start the analysis with. Otherwise we start at all $\text{final}(c)$. All extremal labels are assigned the *extremal value* ι. Then, using our transfer functions ϕ_l for each label $l \in \text{Lab}$, we propagate and process information through the program. The edges for propagation are given by the *flow relation* F, which is either $\text{flow}(c)$ or $\text{flow}^R(c)$.

Definition: Dataflow System

A *dataflow system* $S = (\text{Lab}, E, F, (D, \sqsubseteq), \iota, \phi)$ consists of

- a finite set of program *labels* Lab,
- a set of extremal labels $E \subseteq \text{Lab}$ (here: $\{ \text{init}(c) \}$ or $\text{final}(c)$),
- a *flow relation* $F \subseteq \text{Lab} \times \text{Lab}$ (here: $\text{flow}(c)$ or $\text{flow}^R(c)$),
- a *complete lattice* $(D, \sqsubseteq)$ satisfying ACC
  (with LUB operator $\sqcup$ and least element $\bot$),
- an *extremal value* $\iota \in D$ (for the extremal labels),
- a familiy of *monotonic transfer functions* $\{ \phi_l \mid l \in \text{Lab} \}$ of type $\phi_l : D \to D$.

Definition: Dataflow Equation System

Given a dataflow system $S = (\text{Lab}, E, F, (D, \sqsubseteq), \iota, \phi)$ and w.l.o.g. $\text{Lab} = \{ 1, \dots, n \}$

- S determines the *equation system* (where $l \in \text{Lab}$) $\text{AI}_l = \begin{cases} \iota & \text{if } l \in E, \\ \bigsqcup \{ \phi_{l'}(\text{AI}_{l'}) \mid (l, l') \in F \} & \text{otherwise} \end{cases}$
- $(d_1, \dots, d_n) \in D^n$ is called a *solution* if $d_l = \begin{cases} \iota & \text{if } l \in E, \\ \bigsqcup \{ \phi_{l'}(d_{l'}) \mid (l, l') \in F \} & \text{otherwise} \end{cases}$
- S determines the *transformation* $\Phi_S : D^n \to D^n : (d_1, \dots, d_n) \to (d'_1, \dots, d'_n)$ where
$$d'_l = \begin{cases} \iota & \text{if } l \in E, \\ \bigsqcup \{ \phi_{l'}(d_{l'}) \mid (l, l') \in F \} & \text{otherwise} \end{cases}$$

Note that non-minimal solutions to dataflow equation systems are not always unique.

2.4 Fixpoint Iteration

One can prove that $(d_1, \dots, d_n) \in D^n$ solves the equation system if and only if it is a fixpoint of Φ_S. Basically it's because we have a complete lattice satisfying ACC and a monotonic transfer function.

We can compute this (least) fixpoint by fixpoint iteration: $\text{fix}(\Phi_S) = \bigsqcup \left\{ \Phi_S^k(\bot_{D^n}) \;\middle|\; k \in \mathbb{N} \right\}$ (see Kleene's Fixpoint theorem). It requires at most $m \cdot n$ steps, where m is the height of the lattice.

We can also compute the fixpoint more efficiently by using a *worklist algorithm*. The algorithm keeps a list of control-flow edges to be processed. It iteratively removes an edge (l, l') from the list and computes $\phi_l(\text{AI}_l)$. If it detects that a fixpoint has not been reached ($\phi_l(\text{AI}_l) \not\sqsubseteq \text{AI}_{l'}$), then $\text{AI}_{l'}$ is updated: $\text{AI}_{l'} := \text{AI}_{l'} \sqcup \phi_l(\text{AI}_l)$, and all edges from l', (l', l''), are added to the worklist.

2.5 Meet Over All Paths (MOP)

Definition: MOP Solution

$$\text{mop}(S) = (\text{mop}(l_1), \dots, \text{mop}(l_n)) \in D^n$$

where, for every $l \in \text{Lab}$,

$$\text{mop}(l) = \bigsqcup \{ \phi_\pi(\iota) \mid \pi \in \text{Path}(l) \} .$$

Definition: Paths

Given a dataflow system
$S = (\text{Lab}, E, F, (D, \sqsubseteq), \iota, \phi)$,
the *set of paths up to* $l \in \text{Lab}$ is given by

$$\text{Path}(l) = \left\{ [l_1, \dots, l_{k-1}] \;\middle|\; \begin{array}{l} k \geq 1,\ l_1 \in E, \\ (l_i, l_{i+1}) \in F \\ \forall\, 1 \leq i < k,\ l_k = l \end{array} \right\}$$

The other method to solve dataflow equation systems is *Meet Over All Paths* (MOP). It is the "reference solution", but unfortunately it is undecidable to compute in the general case.

The MOP solution for a block B^l is given by the least upper bound over all paths leading to l.

2.6 Coincidence of MOP and Fixpoint Solution

Theorem: The Coincidence

If the dataflow system S is distributive:

$\text{mop}(S) = \text{fix}(\varphi_S)$

If a dataflow system is distributive, the MOP solution and the fixpoint solution are equal. This is nice, as we can guarantee the optimal result of MOP while still guaranteeing decidability, as well as decent performance.

Definition: Distributivity

Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be complete lattices. A function $F : D \to D'$ is called *distributive*, if, for every $d_1, d_2 \in D$,

$$F(d_1 \sqcup_D d_2) = F(d_1) \sqcup_{D'} F(d_2).$$

A dataflow system is distributive if every φ_l is distributive.

2.7 Dataflow Analysis with Non-ACC Domains

We can also do a dataflow analysis with domains that do not fulfill the Ascending Chain Condition. The trick to prevent nontermination is to use *widening operators* which generate an imprecise result to ensure ACC again.

Definition: Widening Operator

Let $(D, \sqsubseteq)$ be a complete lattice.
A mapping $\nabla : D \times D \to D$ is called a *widening operator* if

- $\forall\, d_1, d_2 \in D\colon \quad d_1 \sqcup d_2 \sqsubseteq d_1 \nabla d_2$,
- for all ascending chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$, the ascending chain $d_0^\nabla \sqsubseteq d_1^\nabla \sqsubseteq \ldots$ eventually stabilises, where $d_0^\nabla = d_0$ and $d_{i+1}^\nabla = d_i^\nabla \nabla d_{i+1}$.

We can modify the worklist algorithm to use the widening operator. In the update step, we simply do $\mathrm{AI}_{l'} := \mathrm{AI}_{l'} \nabla \phi_l(\mathrm{AI}_l)$ instead of $\mathrm{AI}_{l'} := \mathrm{AI}_{l'} \sqcup \phi_l(\mathrm{AI}_l)$. Note that in contrast to $\sqcup$ the widening operator ∇ does not require commutativity, associativity, monotonicity nor absorption ($d \nabla d = d$). On the other hand, only $\mathrm{fix}^\nabla(\Phi_S) \sqsupseteq \mathrm{fix}(\Phi_S)$ (soundness) is guaranteed.

Widening may lead to very imprecise results, which can be improved by *narrowing* again. Let $\mathrm{fix}^\nabla(\Phi_S)$ be the result of widening (e.g. using the algorithm above). Narrowing applies Φ_S k times (for some k), so the end result is $\Phi_S^k(\mathrm{fix}^\nabla(\Phi_S))$.

2.8 Analysis Using Conditional Branches

Previously, our analyses did not use information from branches – so for example code following a successful equality check $a = 3$ could not be analysed with this fact. In the lecture, we introduced an assert [fact] statement that is inserted after each branch. Then it is possible to simply extend the transfer functions to work with assert information.

3 Abstract Interpretation

Abstract interpretation is a theory of sound approximation of program semantics. The execution is abstracted to work on abstract values (e.g. parity, intervals, or types!).

3.1 Galois Connections

A *Galois connection* is a pair of monotonic functions between lattices: The abstraction α and the concretisation γ. L is the domain of sets of concrete values and M the one for abstract values.

The two conditions ensure that α is an over-approximation (1) and that concretisation followed by abstraction does not lose precision (2).

Definition: Galois Connection

Let $(L, \sqsubseteq_L)$, $(M, \sqsubseteq_M)$ be complete lattices.
A pair (α, γ) of monotonic functions $\alpha : L \to M$, $\gamma : M \to L$ is called a *Galois connection* if

1. $\forall\, l \in L : \; l \sqsubseteq_L \gamma(\alpha(l))$ and
2. $\forall\, m \in M : \; \alpha(\gamma(m)) \sqsubseteq_M m$.

Galois connections have a few useful properties:

- $\alpha(l) \sqsubseteq_M m \iff l \sqsubseteq_L \gamma(m)$.
- γ is uniquely determined by α: $\gamma(m) = \bigsqcup \{\, l \in L \mid \alpha(l) \sqsubseteq_M m \,\}$.
- α is uniquely determined by γ: $\alpha(m) = \bigsqcap \{\, m \in M \mid l \sqsubseteq_L \gamma(m) \,\}$.
- α is completely distributive: $\forall\, L' \subseteq L : \; \alpha(\bigsqcup L') = \bigsqcup \{\, \alpha(l) \mid l \in L' \,\}$.
- γ is completely multiplicative: $\forall\, M' \subseteq M : \; \gamma(\bigsqcap M') = \bigsqcap \{\, \gamma(m) \mid m \in M' \,\}$.

3.2 Safe Approximation of Functions

Given a Galois connection, we want to approximate functions directly and safely. E.g. when we are doing parity abstraction, we want to directly approximate $f = +$ in our approximation domain ($\{ 0, 1 \}$) by some $f^{\#} = +^{\#}$. The intuition is that $f^{\#}$ should cover all concrete results of f.

Definition: Safe Approximation

Let (α, γ) be a Galois connection and let $f : L^n \to L$ and $f^{\#} : M^n \to M$ be functions of rank $n \in \mathbb{N}$. We call $f^{\#}$ a *safe approximation* of f if

$$\alpha(f(\gamma(m_1), \dots, \gamma(m_n))) \sqsubseteq_M f^{\#}(m_1, \dots, m_n) \qquad \forall\, m_1, \dots, m_n \in M.$$

Additionally, $f^{\#}$ is called *most precise* if the reverse inclusion is also true:

$$\alpha(f(\gamma(m_1), \dots, \gamma(m_n))) = f^{\#}(m_1, \dots, m_n) \qquad \forall\, m_1, \dots, m_n \in M.$$

Usually, f and $f^{\#}$ are monotonic. Then, the following theorem allows us to simplify the safety proof:

Theorem: Safe Approximation with Monotonic f

If $f : L^n \to L$ and $f^{\#} : M^n \to M$ are both monotonic, then $f^{\#}$ is a safe approximation of f if

$$\alpha(f(l_1, \dots, l_n)) \sqsubseteq_M f^{\#}(\alpha(l_1), \dots, \alpha(l_n)) \qquad \forall\, l_1, \dots, l_n \in L.$$

3.3 Safe Approximation of Extraction Functions

We define an *extraction function* β that determines a Galois connection (α, γ):

$$\begin{aligned} \alpha &: L \to M, & l &\mapsto \{ \beta(c) \mid c \in l \}, \\ \gamma &: M \to L, & m &\mapsto \{ c \in C \mid \beta(c) \in m \}. \end{aligned}$$

Definition: Extraction Function

If $L = 2^C$ and $M = 2^A$ with $\sqsubseteq_L = \sqsubseteq_M = \subseteq$, then $\beta : C \to A$ is an *extraction function*.

Theorem: Most Precise Abstraction of an Extraction Function

If $\beta : C \to A$ is an extraction function, the most precise abstraction of f is

$$f^{\#} : M^n \to M, (m_1, \dots, m_n) \mapsto \left\{ \beta(f(c_1, \dots, c_n)) \,\middle|\, \forall\, i \in \{ 1, \dots, n \} : c_i \in \beta^{-1}(m_i) \right\}.$$

3.4 Abstract Semantics (of WHILE)

A normal *execution relation* for WHILE statements $\to \subseteq (\mathrm{Cmd} \times \Sigma) \times (\mathrm{Cmd}_{\downarrow} \times \Sigma)$ is a relation of *configurations* $\langle c, \sigma \rangle$. $\sigma \in \Sigma$ is the mapping of variable valuations $\sigma : \mathrm{Var} \to \mathbb{Z}$. c is either a statement or, for $\mathrm{Cmd}_{\downarrow}$, also possibly the $\downarrow$ symbol which indicates execution termination.

From this we get the *concrete domain* $L := (2^{\Sigma}, \subseteq)$ and the *concrete transition function* family with $c \in \mathrm{Cmd}$, $c' \in \mathrm{Cmd} \cup \{ \downarrow \}$: $\mathrm{next}_{c,c'} : 2^{\Sigma} \to 2^{\Sigma}$, $S \mapsto \{ \sigma' \in \Sigma \mid \exists \sigma \in S : \langle c, \sigma \rangle \to \langle c', \sigma' \rangle \}$.

For an *abstract domain* Abs, the *abstract semantics* is defined by the function family $\mathrm{next}^{\#}_{c,c'} : \mathrm{Abs} \to \mathrm{Abs}$ (as above) and where each $\mathrm{next}^{\#}_{c,c'}$ safely approximates $\mathrm{next}_{c,c'}$, i.e.
$\alpha(\mathrm{next}_{c,c'}(\gamma(abs))) \sqsubseteq_{\mathrm{Abs}} \mathrm{next}^{\#}_{c,c'}(abs)$.

Definition: Abstract WHILE-Program State

Let $\beta : \mathbb{N} \to A$ be an extraction function.

- An *abstract program state* ρ is an element of the set $\{\, \rho \mid \rho : \mathrm{Var} \to A \,\}$ called the *abstract state space*.
- The *abstract domain* is denoted by $\mathrm{Abs} := 2^{\mathrm{Var} \to A}$.
- The *abstraction function* $\alpha : 2^{\Sigma} \to \mathrm{Abs}$ is given by $\alpha(S) := \{\, \beta \circ \sigma \mid \sigma \in S \,\}$.

The abstract transition function for WHILE is defined via the *abstract execution relation* (not in this Panikzettel): $\mathrm{next}^{\#}_{c,c'}(abs) := \bigcup \{\, abs' \in \mathrm{Abs} \mid \langle c, abs \rangle \Rightarrow \langle c', abs' \rangle \,\}$. We have shown the abstract transition function's soundness: It is a safe approximation for the concrete transition function.

3.5 Counterexample-Guided Abstraction Refinement (CEGAR)

CEGAR addresses the problem of choosing the appropiate approximation for an analysis by refining abstractions until the property is satisifed or an error is found. The iterative refinement may not terminate.

3.5.1 Predicate Abstraction

Definition: Predicate Abstraction

Let Var be a set of variables.

- A *predicate* is a Boolean expression $p \in \mathrm{BExp}$ over Var.
- A state $\sigma \in \Sigma$ *satisfies* p ($\sigma \models p$) if $\mathrm{val}_{\sigma}(p) = \mathrm{true}$.
- p *implies* q if $\sigma \models q$ whenever $\sigma \models p$.
- p and q are *equivalent* if $p \models q$ and $q \models p$.
- p and q are *independent* if $p \not\models q$ and $q \not\models p$.
- Let $P = \{\, p_1, \dots, p_n \,\} \subset \mathrm{BExp}$ be a finite set of predicates.
  Let $\neg P = \{\, \neg p_1, \dots, \neg p_n \,\}$. An element of $P \cup \neg P$ is called a *literal*.

The *predicate abstraction lattice* is defined by:

$$\mathrm{Abs}(P) := \left(\left\{ \bigwedge Q \;\middle|\; Q \subseteq P \cup \neg P \right\} \cup \{\, \mathrm{false} \,\}, \models \right)$$

We abbreviate $\mathrm{true} := \bigwedge \emptyset$, and $\mathrm{false} := \bigwedge \{\, p_i, \neg p_i, \dots \,\}$ if $P \neq \emptyset$.

The predicate abstraction lattice $\mathrm{Abs}(P)$ is a complete lattice with
$\bot = \mathrm{false}, \quad \top = \mathrm{true}, \quad Q_1 \sqcap Q_2 = \overline{Q_1 \wedge Q_2}, \quad Q_1 \sqcup Q_2 = \overline{Q_1 \vee Q_2}$.
$\overline{b} := \bigwedge \{\, q \in P \cup \neg P \mid b \models q \,\}$ is the strongest fomula that is implied by b.

Handy rules only if predicates in P are pairwise independent:
$Q_1 \sqcap Q_2 \ (= \overline{Q_1 \wedge Q_2}) = \bigwedge(Q_1 \cup Q_2) \qquad Q_1 \sqcup Q_2 \ (= \overline{Q_1 \vee Q_2}) = \bigwedge(Q_1 \cap Q_2)$

The *Galois connection for predicate abstraction* is determined by

$$\begin{aligned} \alpha : 2^\Sigma &\to \mathrm{Abs}(P) & \gamma : \mathrm{Abs}(P) &\to 2^\Sigma \\ \alpha(S) &:= \bigsqcup \{\, Q_\sigma \mid \sigma \in S \,\} & \gamma(Q) &:= \{\, \sigma \in \Sigma \mid \sigma \models Q \,\} \end{aligned}$$

where $Q_\sigma := \bigwedge (\{\, p \in P \mid \sigma \models p \,\} \cup \{\, \neg p \in \neg P \mid \sigma \not\models p \,\})$.

In the lecture, the *execution relation for predicate abstraction* was defined (not shown here). $\langle c, \mathrm{false} \rangle$ represents an unreachable configuration (there is no $\sigma \in \Sigma$ s.t. $\sigma \models \mathrm{false}$). If $P = \emptyset$, then $\mathrm{Abs}(P) = \{\, \mathrm{true},\ \mathrm{false} \,\}$. If additionally no $b \in \mathrm{BExp}_c$ is a tautology or a contradiction, then the abstract transition system with initial configuration $\langle c, \mathrm{true} \rangle$ corresponds to the control flow graph of c.

Also of note: Computing the strongest formula in $\mathrm{Abs}(P)$ implied by b, $\overline{b}$, is not possible in general.

3.5.2 Counterexamples

Definition: Counterexample

A *counterexample* is a sequence of $k \geq 1$ abstract transitions of the form $\langle c_0, Q_0 \rangle \Rightarrow \langle c_1, Q_1 \rangle \Rightarrow \ldots \Rightarrow \langle c_k, Q_k \rangle$ where $c_0, \ldots, c_k \in \mathrm{Cmd}$ (or $c_k = \downarrow$), $Q_0, \ldots, Q_k \in \mathrm{Abs}(P)$ with $Q_0 = \mathrm{true}$ and $Q_k \not\equiv \mathrm{false}$.

A counterexample is *real* (otherwise *spurious*) if there are concrete states $\sigma_0, \ldots, \sigma_k \in \Sigma$ s.t. $\forall i \in \{\, 1, \ldots, k \,\} : \sigma_i \models Q_i$ and $\langle c_{i-1}, \sigma_{i-1} \rangle \to \langle c_i, \sigma_i \rangle$

Assume now we have analyzed the program with some abstraction and found a counterexample for the property. The counterexample is a path to a location that does not satisfy the property. But we need to find out if the path is really possible (if the counterexample is spurious or not).

If the counterexample is spurious, we can collect conditions (strongest postconditions) along the path showing that the end is unreachable.

Theorem: Elimination of Spurious Counterexamples

If $\langle c_0, \mathrm{true} \rangle \Rightarrow \ldots \Rightarrow \langle c_k, Q_k \rangle$ is a spurious counterexample, there are Boolean expressions $b_0, \ldots, b_k$ with $b_0 \equiv \mathrm{true}$ and $b_k \equiv \mathrm{false}$ and $\forall i \in \{\, 1, \ldots, k \,\}, \sigma, \sigma' \in \Sigma : \sigma \models b_{i-1} \wedge (\langle c_{i-1}, \sigma \rangle \to \langle c_i, \sigma' \rangle) \Rightarrow \sigma' \models b_i$.

3.5.3 Abstraction Refinement

With $b_0, \ldots, b_k$ from the previous theorem, we can *refine* our abstraction so that the spurious counterexample cannot happen anymore. Specifically, we set $P' := P \cup \{\, p_1, \ldots, p_n \,\}$ where $p_1, \ldots, p_n$ are the atomic conjuncts occurring in $b_1, \ldots, b_k$.

3.5.4 Using Craig Interpolants

Craig interpolants help us to simplify predicates so that they only contain variables relevant for the current execution path.

We omit the algorithm for computing Craig Interpolants here.

Definition: Craig Interpolant

Let $b_1, b_2 \in \text{BExp}$ where $b_1 \models b_2$.
A *Craig interpolant* of b_1 and b_2 is a formula $b_3 \in \text{BExp}$ with $b_1 \models b_3$, $b_3 \models b_2$ and $\text{Var}_{b_3} \subseteq \text{Var}_{b_1} \cap \text{Var}_{b_2}$.

4 Interprocedural Dataflow Analysis

Our dataflow analysis framework from before was only *intraprocedural*, that is without support for user-defined functions. This section extends the framework to support *interprocedural* analysis. The extension follows the *functional* approach which summarizes function effects. Declarations are only allowed on the top level, and mutual recursion is supported. Function calls have one call-by-value and one call-by-result parameter.

Definition: A Language With Procedures

Syntactic Categories

Category	Domain	Meta variable
Procedure identifiers	$Pid = \{ P, Q, \dots \}$	P
Procedure declarations	$PDec$	p
Commands (statements)	Cmd	c

Context-free grammar

$$\begin{array}{lll}
\texttt{p} & \texttt{:==} & \texttt{proc[}P\texttt{(val }x\texttt{,res }y\texttt{)]}^{l_n}\ \texttt{is}\ c\ \texttt{[end]}^{l_x}\texttt{;}\ \texttt{p} \ \mid\ \varepsilon\ \in PDec \\
\texttt{c} & \texttt{:==} & \texttt{[skip]}^{l}\ \mid\ \texttt{[}x\ \texttt{:=}\ a\texttt{]\^{}l}\ \mid\ c_1\texttt{;}\ c_2\ \mid\ \texttt{if [}b\texttt{]\^{}l then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}\ \mid \\
 & & \texttt{while [}b\texttt{]}^{l}\ \texttt{do}\ c\ \texttt{end}\ \mid\ \texttt{[call }P\texttt{(}a\texttt{, }x\texttt{)]}^{l_c}_{l_r}\ \in \text{Cmd}
\end{array}$$

Note that calls are annotated with call (l_c) and return (l_r) locations.

init, final and flow relations are extended from WHILE as follows:

	$\texttt{proc[}P\texttt{(val }x\texttt{,res }y\texttt{)]}^{l_n}\ \texttt{is}\ c\ \texttt{[end]}^{l_x}$	$\texttt{[call }P(a,x)\texttt{]}^{l_c}_{l_r}$
init(...)	l_n	l_c
final(...)	$\{ l_x \}$	$\{ l_r \}$
flow(...)	$\{ (l_n, \text{init}(c) \} \cup \text{flow}(c) \cup \{ (l, l_x) \mid l \in \text{final}(c) \}$	$\{ (l_c; l_n), (l_x; l_r) \}$

The *interprocedural flow* of a program is defined by:

$$\text{iflow} = \left\{ (l_c, l_n, l_x, l_r) \;\middle|\; \begin{array}{l} p \text{ contains proc with } l_n \text{ and } l_r \text{ and} \\ c \text{ contains call with } l_r \text{ and } l_c \end{array} \right\} \subseteq \text{Lab}^4$$

4.1 Meet Over All Valid Paths (MVP)

Similar to the Meet Over all Paths (MOP) solution we define the *Meet over all Valid Paths (MVP)* solution.

Definition: Valid Path Fragments

Given a dataflow system $S = (\mathrm{Lab}, E, F, (D, \sqsubseteq), \iota, \phi)$ and $l_1, l_2 \in \mathrm{Lab}$, the set of *valid paths from l_1 to l_2* is generated by the nonterminal symbol $P[l_1, l_2]$ according to the following context-free grammar:

$$\begin{aligned} P[l_1, l_2] &\to l_1 && \text{if } l_1 = l_2 \\ P[l_1, l_3] &\to l_1, P[l_2, l_3] && \text{if } (l_1, l_2) \in F \\ P[l_c, l] &\to l_c, P[l_n, l_x], P[l_r, l] && \text{if } (l_c, l_n, l_x, l_r) \in \mathrm{iflow} \end{aligned}$$

Definition: Complete Valid Paths

Let S be a dataflow system. For every $l \in \mathrm{Lab}$, the set of *valid paths up to l* is given by

$$\mathrm{VPath}(l) = \{\, [l_1, \ldots, l_{k-1}] \mid k \geq 1, l_1 \in E, l_k = l, [l_1, \ldots, l_k] \text{ valid path from } l_1 \text{ to } l_k \,\}$$

For $\pi = [l_1, \ldots, l_{k-1}] \in \mathrm{VPath}(l)$, we define the *transfer function* by

$$\phi_\pi = \phi_{k-1} \circ \ldots \circ \phi_{l_1} \circ \mathrm{id}_D$$

Definition: MVP solution

Let S be a dataflow system where $\mathrm{Lab} = \{\, l_1, \ldots, l_n \,\}$. The *MVP solution* for S is determined by

$$\mathrm{mvp}(S) = (\mathrm{mvp}(l_1), \ldots, \mathrm{mvp}(l_n)) \in D^n$$

where, for every $l \in \mathrm{Lab}$,

$$\mathrm{mvp}(l) = \bigsqcup \{\, \phi_\pi(\iota) \mid \pi \in \mathrm{VPath}(l) \,\}.$$

As an extension of MOP, MVP is undecidable as well.

4.2 Fixpoint Solution

The *interprocedural fixpoint solution* extends its intraprocedural counterpart as well. The idea is to represent the combined effect of procedure executions by *procedure summaries*.

Definition: Interprocedural Dataflow Equation System

- *Dataflow equations* for each $l \in \text{Lab}$:

$$\text{AI}_l = \begin{cases} \iota & \text{if } l \in E, \\ \text{AI}_{l_c} & \text{if } l = l_r \text{ for some } (l_c, l_n, l_x, l_r) \in \text{iflow}, \\ \bigsqcup \{\, \varphi_{l'}(\text{AI}_{l'} \mid (l', l) \in F,\ (l'; l) \in F \,\} & \text{otherwise} \end{cases}$$

- *Node transfer functions* for each $l \in \text{Lab} \setminus \{\, l_x \mid (l_c, l_n, l_x, l_r) \in \text{iflow} \,\}$:

$$\varphi_l(d) = \begin{cases} \text{combine}(d, \Phi_{l_x}(\varphi_{l_c}(d)) & \text{if } l = l_r \text{ for some } (l_c, l_n, l_x, l_r) \in \text{iflow}, \\ \text{specific to analysis} & \text{otherwise} \end{cases}$$

- *Procedure summary functions* for each $l \in \text{Lab}$ in a procedure:

$$\Phi_l(d) = \begin{cases} d & \text{if } l = l_n \text{ for some } (l_c, l_n, l_x, l_r) \in \text{iflow}, \\ \Phi_{l_c}(d) & \text{if } l = l_r \text{ for some } (l_c, l_n, l_x, l_r) \in \text{iflow}, \\ \bigsqcup \{\, \varphi_{l'}(\Phi_{l'}(d)) \mid (l', l) \in F \,\} & \text{otherwise} \end{cases}$$

The above equation system is recursive in both AI_l (type D) and Φ_l (type $D \to D$).

To solve this equation system, we can use the induced *monotonic functional on the complete lattice*:

$$\Psi_{\hat{S}} : \underbrace{D^n}_{\text{AI}} \times \underbrace{(D \to D)^m}_{\Phi} \to D^n \times (D \to D)^m \quad (n = |\text{Lab}|,\ m \leq n)$$

Now we can use fixpoint iteration:

$$\text{fix}(\Psi_{\hat{S}}) = \bigsqcup \left\{ \Psi^k_{\hat{S}}(\bot) \,\middle|\, k \in \mathbb{N} \right\} \in D^n \times (D \to D)^m, \quad \bot = (\bot^n_D,\ [d \to \bot_D \mid d \in D]^m)$$