Widening with Landmarks

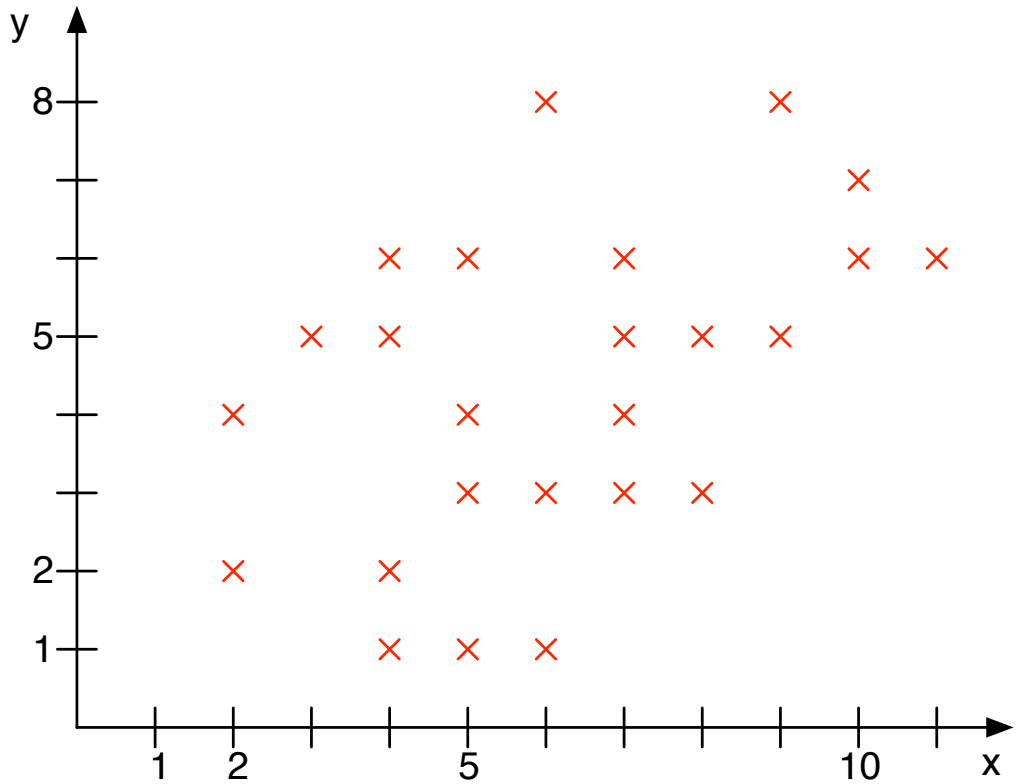
Axel Simon and Andy King

Computing Laboratory, University of Kent,
Canterbury, CT2 7NF, UK
{A.Simon,A.King}@kent.ac.uk
November 9, 2006

Outline of the Talk

The Idea of Polyhedra in Program Analysis

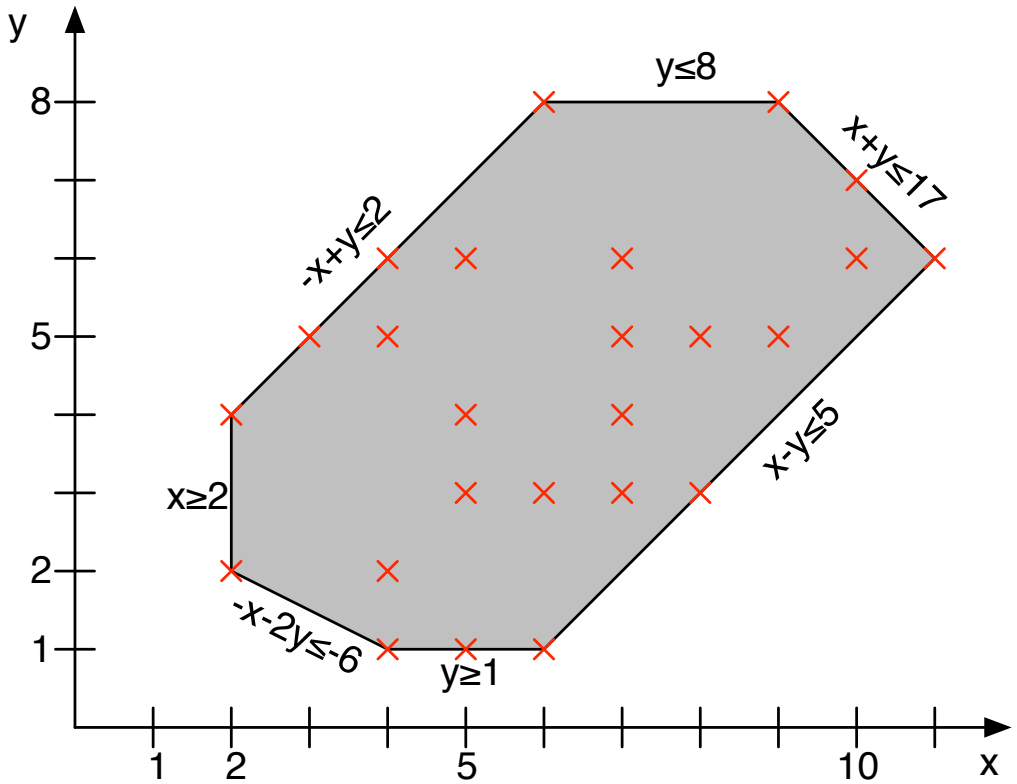
Treat valuations of x, y as points in 2-dimensional space:



Each point is valuation in one run.

The Idea of Polyhedra in Program Analysis

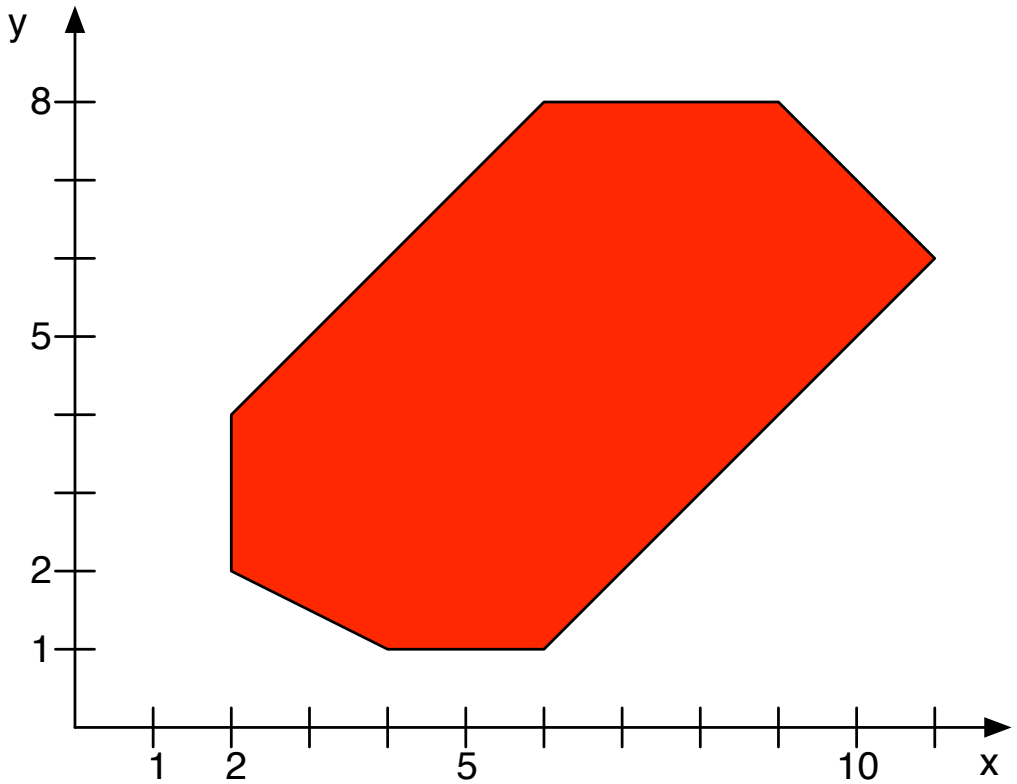
Treat valuations of x, y as points in 2-dimensional space:



Approximate with finite set of inequalities.

The Idea of Polyhedra in Program Analysis

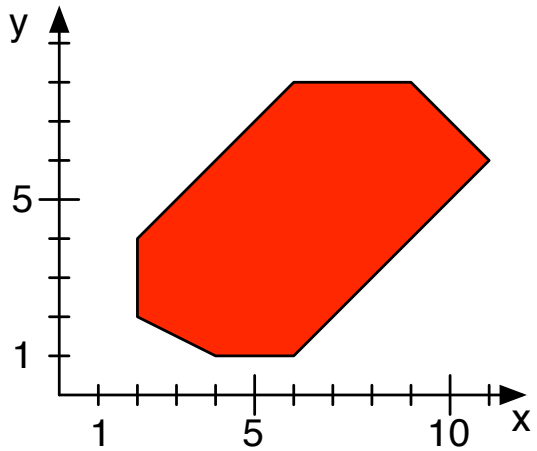
Treat valuations of x, y as points in 2-dimensional space:



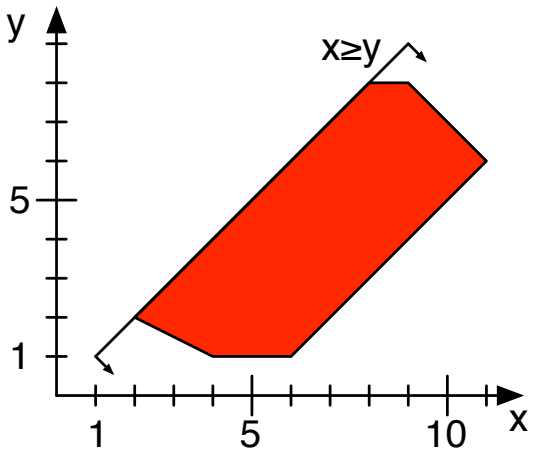
Use polyhedra to describe possible program states.

Principal Operations on Polyhedra

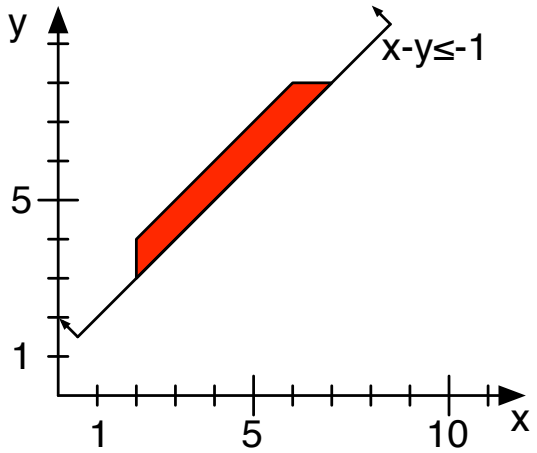
Evaluate: `if (x<y) { x=3; }`



Initial state: P



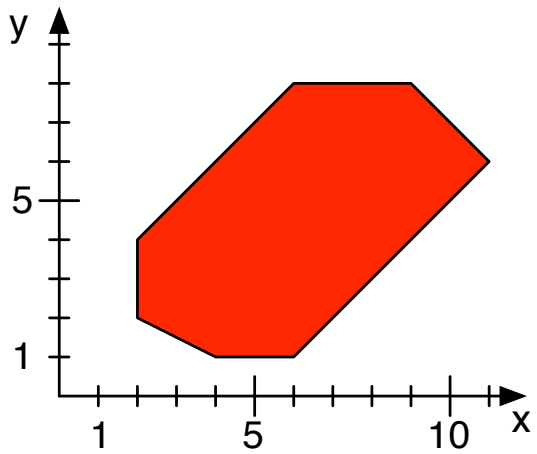
Else-branch:
$Q = P \sqcap \{x \geq y\}$



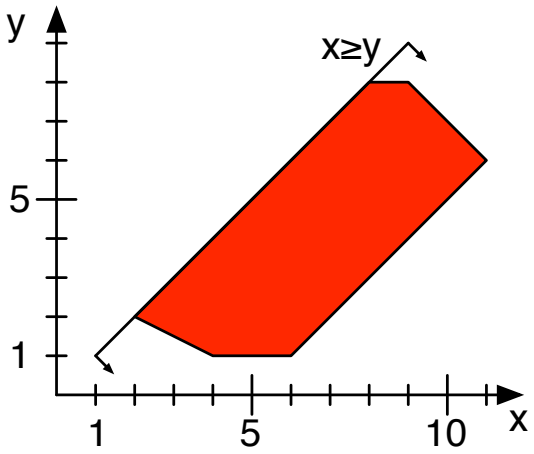
Then-branch:
$R = P \sqcap \{x < y\}$

Principal Operations on Polyhedra

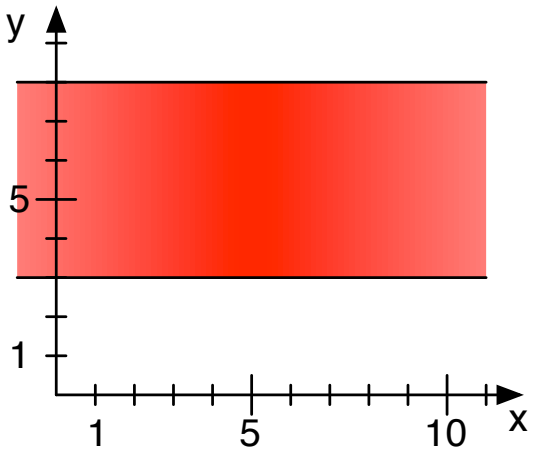
Evaluate: `if (x<y) { x=3; }`



Initial state: P



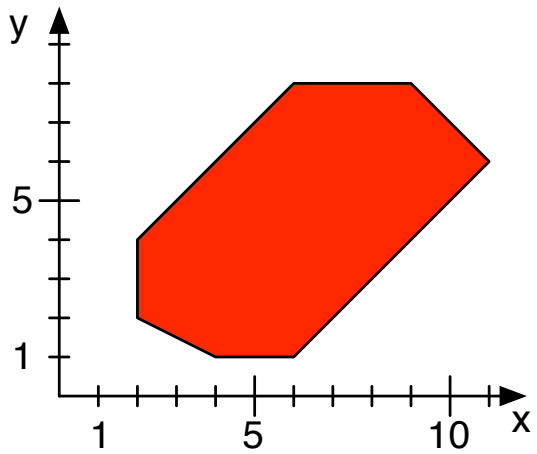
Else-branch:
$Q = P \sqcap \{x \geq y\}$



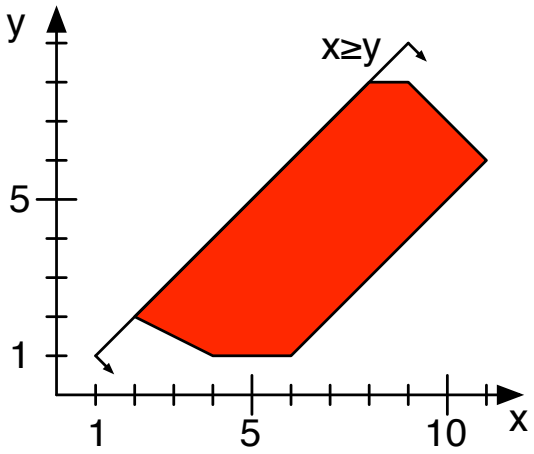
Evaluate `x=3`:
$S = \exists_x(R)$

Principal Operations on Polyhedra

Evaluate: `if (x<y) { x=3; }`

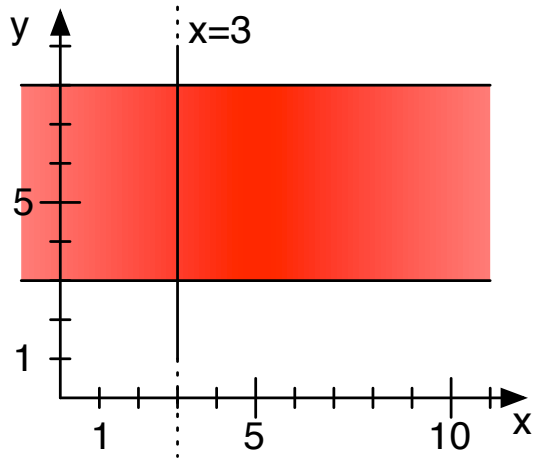


Initial state: P



Else-branch:

$Q = P \sqcap \{x \geq y\}$

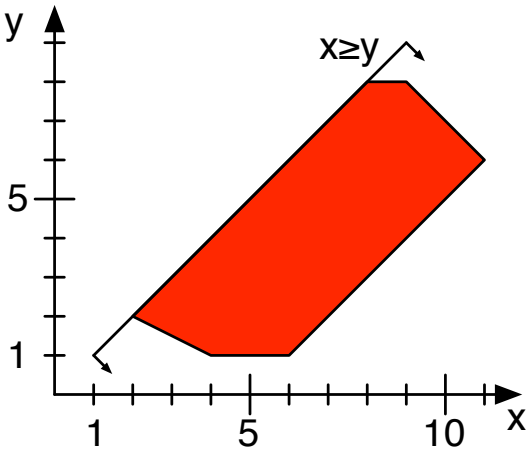


$S = \exists_x(R)$

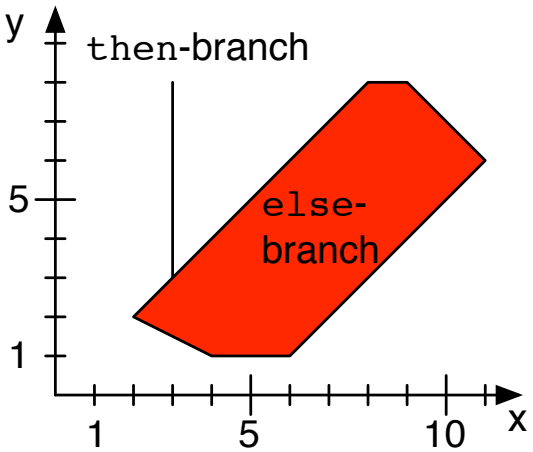
$T = S \sqcap \{x = 3\}$

Principal Operations on Polyhedra

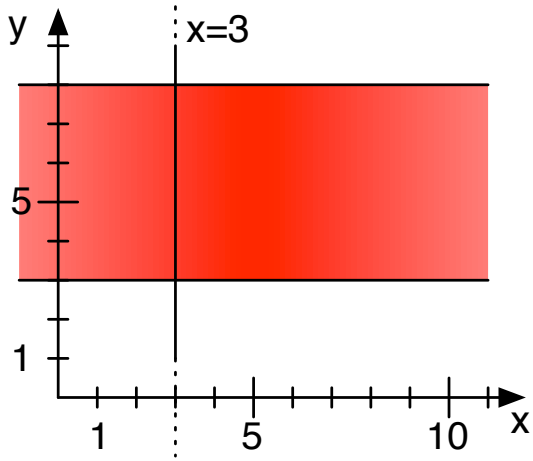
Evaluate: `if (x<y) { x=3; }`



Else-branch:
$Q = P \sqcap \{x \geq y\}$



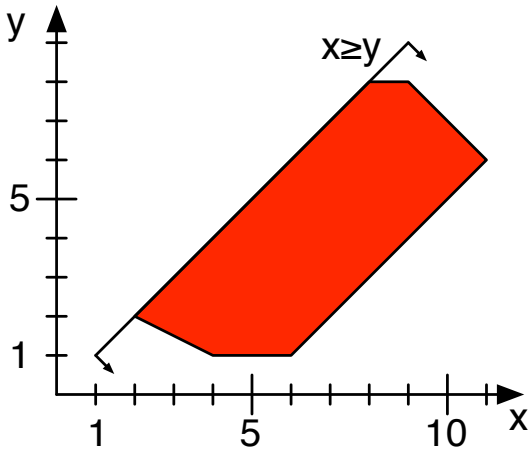
Join branches:
$U = Q \sqcup T$



$S = \exists_x(R)$
$T = S \sqcap \{x = 3\}$

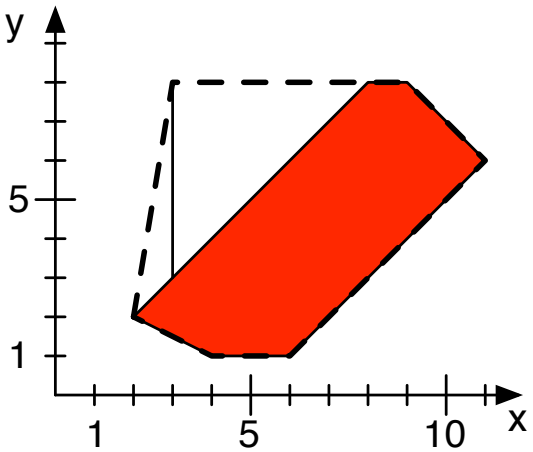
Principal Operations on Polyhedra

Evaluate: `if (x<y) { x=3; }`



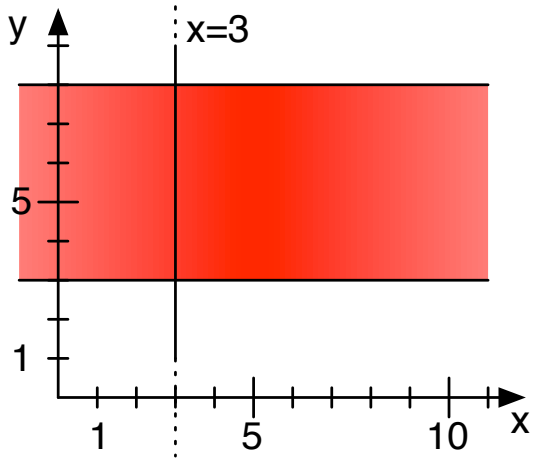
Else-branch:

$Q = P \sqcap \{x \geq y\}$



Join branches:

$U = Q \sqcup T$

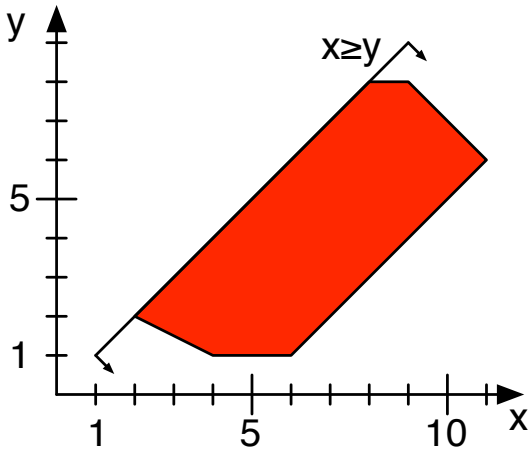


$S = \exists_x(R)$

$T = S \sqcap \{x = 3\}$

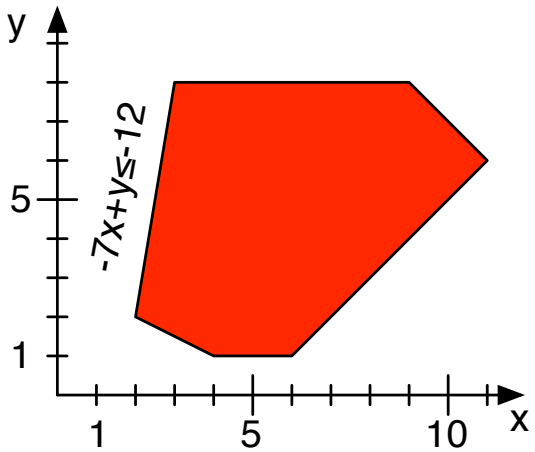
Principal Operations on Polyhedra

Evaluate: `if (x<y) { x=3; }`



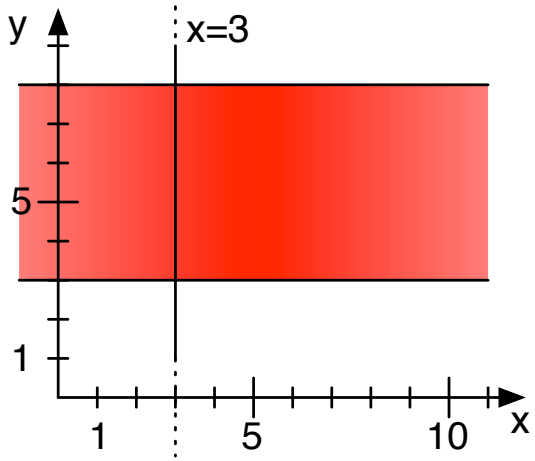
Else-branch:

$Q = P \sqcap \{x \geq y\}$



Join branches:

$U = Q \sqcup T$



$S = \exists_x(R)$

$T = S \sqcap \{x = 3\}$

Idea of Polyhedral Analysis

1. Infer all possible states of a program.

Idea of Polyhedral Analysis

1. Infer all possible states of a program.

2. Ensure that none of these are erroneous.

Idea of Polyhedral Analysis

1. Infer all possible states of a program.
   - Track polyhedron for each basic block.

2. Ensure that none of these are erroneous.

Idea of Polyhedral Analysis

1. Infer all possible states of a program.
   - Track polyhedron for each basic block.
   - Define semantics of statements on polyhedra.
     - Approximate non-linear functions.
     - Assume worst-case for all inputs.
     - Assume unrestricted polyhedron at `main`.

2. Ensure that none of these are erroneous.

Idea of Polyhedral Analysis

1. Infer all possible states of a program.
   - Track polyhedron for each basic block.
   - Define semantics of statements on polyhedra.
     - Approximate non-linear functions.
     - Assume worst-case for all inputs.
     - Assume unrestricted polyhedron at `main`.
   - Fixpoint: Apply transfer functions until stable.
2. Ensure that none of these are erroneous.

Practical Polyhedral Analysis

Polyhedral operations are exponential in no. of variables.

- cannot infer information on individual array elements
- use TVPI sub-domain of general polyhedra: only $ax_i + bx_j \leq c$
- tightening around contained integral points available for TVPI

Practical Polyhedral Analysis

Polyhedral operations are exponential in no. of variables.

- cannot infer information on individual array elements
- use TVPI sub-domain of general polyhedra: only $ax_i + bx_j \leq c$
- tightening around contained integral points available for TVPI

For precise analysis of string buffers:

- ignore string buffer contents, but track NUL character

Prove String-Buffer Operation Correct

Prove the absence of out-of-bound array accesses:

```
char s[32] = "the string";
while (*s) s++;
```

Method:

- Expand `while`-loop.

Prove String-Buffer Operation Correct

Prove the absence of out-of-bound array accesses:

```
char s[32] = "the string";
while (*s) s++;
```

Method:

- Expand `while`-loop.
- Use the following polyhedral variables:
  - s is offset of pointer `s`
  - n is first NUL position in string buffer
  - c is character at `*s`

Calculating a fixpoint ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /* Q = ∅ */
   c = *s;
   /*R = ∅*/
   if (c==0) break;
   /*  T = ∅*/
   s++;
   /*  U = ∅*/
};
/* S = ∅  */
```

Let $P \equiv \{s = 0, n = 10\}$, $Q = P \sqcup U$, $T = R \sqcap \{c > 0\}$, $S = \ldots$,

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Calculating a fixpoint ($n = 10$ omitted for brevity):

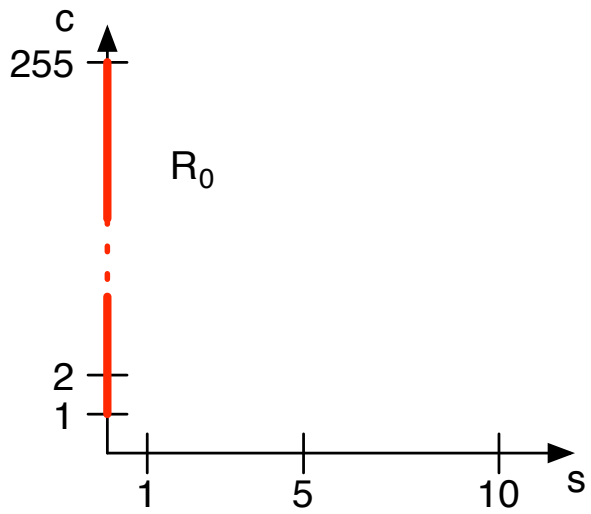
```
char s[32] = "the string";
while (true) {
   /*  Q0 ≡ {s = 0} */
   c = *s;
   /*R = ∅*/
   if (c==0) break;
   /*  T = ∅*/
   s++;
   /*  U = ∅*/
};
/* S = ∅  */
```

Calculate $Q_0 = P \sqcup U = P$.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Calculating a fixpoint ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /* Q0 ≡ {s = 0} */
   c = *s;
   /* R0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /*  T = ∅*/
   s++;
   /*  U = ∅*/
};
/* S = ∅  */
```

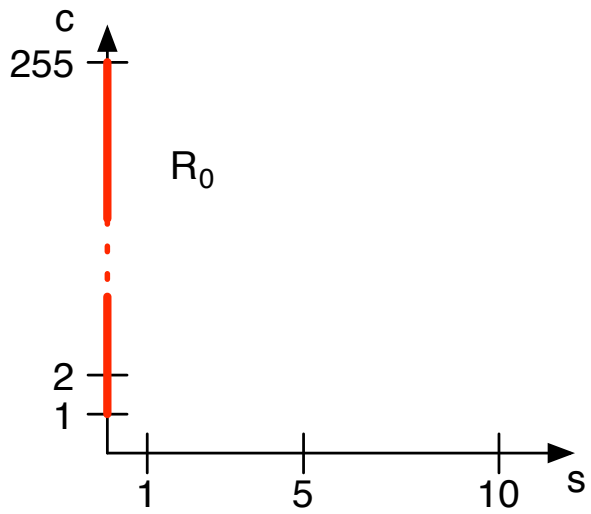


Intersection with $s = n$ and $s > n$ yields two empty polyhedra.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Calculating a fixpoint ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /* Q0 ≡ {s = 0} */
   c = *s;
   /*R0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /*  T0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   s++;
   /*  U = ∅*/
};
/* S = ∅  */
```

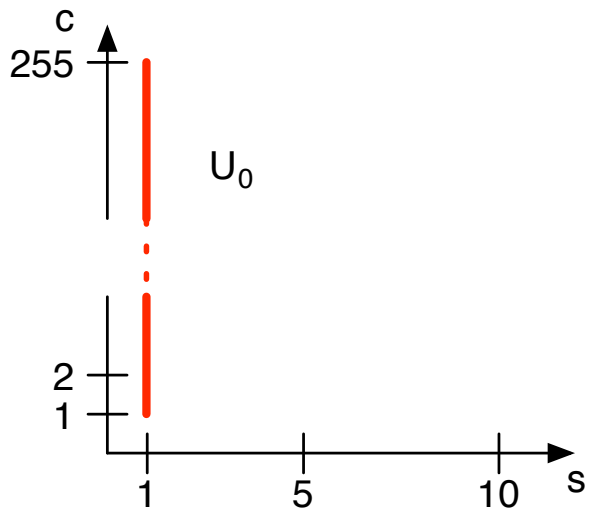


Loop invariant c!=0 holds for all of R.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Calculating a fixpoint ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /* Q0 ≡ {s = 0} */
   c = *s;
   /*R0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /* T0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   s++;
   /* U0 ≡ {s = 1, 1 ≤ c ≤ 255}*/
};
/* S = ∅ */
```

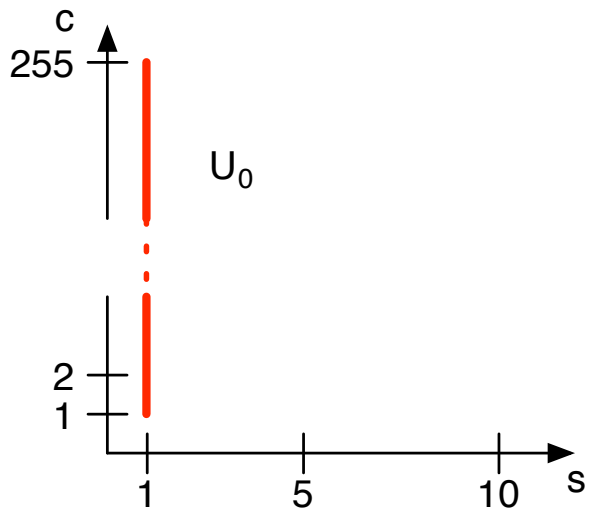


Increment s by linearly translating T.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Calculating a fixpoint ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /*  Q1 ≡ {0 ≤ s ≤ 1} */
   c = *s;
   /*R0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /* T0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   s++;
   /* U0 ≡ {s = 1, 1 ≤ c ≤ 255}*/
};
/* S = ∅  */
```

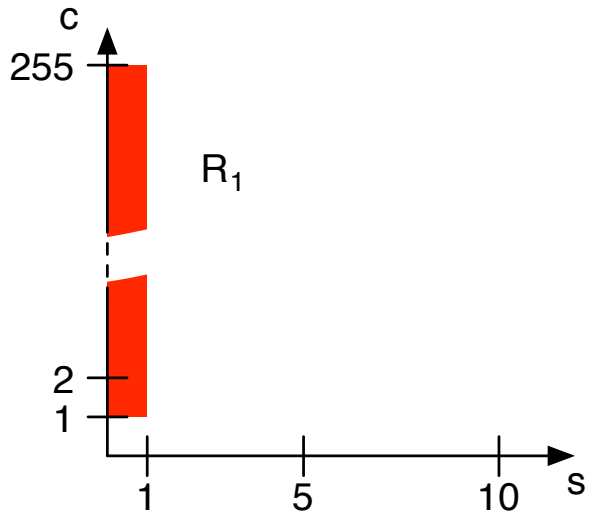


Calculate $Q_1 = P \sqcup U_0$.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Calculating a fixpoint ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /* Q1 ≡ {0 ≤ s ≤ 1} */
   c = *s;
   /* R1 ≡ {0 ≤ s ≤ 1, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /* T0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   s++;
   /* U0 ≡ {s = 1, 1 ≤ c ≤ 255}*/
};
/* S = ∅ */
```

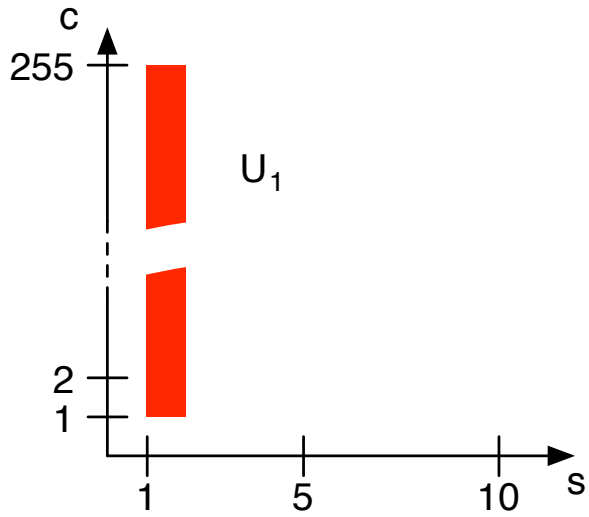


First case of R is enabled.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Calculating a fixpoint ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /* Q1 ≡ {0 ≤ s ≤ 1} */
   c = *s;
   /*R1 ≡ {0 ≤ s ≤ 1, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /*  T1 ≡ {0 ≤ s ≤ 1, 1 ≤ c ≤ 255}*/
   s++;
   /* U1 ≡ {1 ≤ s ≤ 2, 1 ≤ c ≤ 255}*/
};
/* S = ∅  */
```

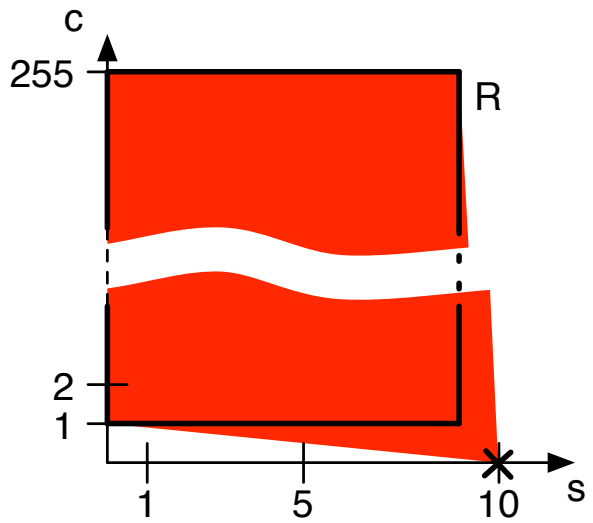


This process continues for another 8 iterations.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Calculating a fixpoint ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /*  Q10 ≡ {0 ≤ s ≤ 10} */
   c = *s;
   /* R10 ≡ {c ≤ 255, i + 10c ≥ 10,  */
                255i + c ≤ 2550}
   if (c==0) break;
   /*  T9 ≡ {0 ≤ s ≤ 9, 1 ≤ c ≤ 255}*/
   s++;
   /* U9 ≡ {1 ≤ s ≤ 10, 1 ≤ c ≤ 255}*/
};
/* S = ∅  */
```

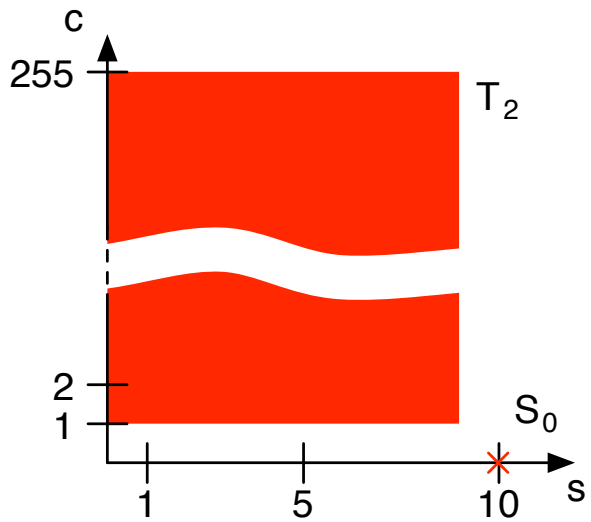


The first two of the following equations now contribute:

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \le c \le 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \le c \le 255\} \end{aligned}$$

Calculating a fixpoint ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
```
$/* \; Q_{10} \equiv \{0 \leq s \leq 10\} \; */$
```
  c = *s;
```
$/* R_{10} \equiv \{c \leq 255, i + 10c \geq 10, \; */$
$\qquad 255i + c \leq 2550\}$
```
  if (c==0) break;
```
$/* \; T_{10} \equiv \{0 \leq s \leq 9, 1 \leq c \leq 255\} */$
```
  s++;
```
$/* \; U_{10} \equiv \{1 \leq s \leq 10, 1 \leq c \leq 255\} */$
```
};
```
$/* \; S_0 \equiv \{c = 0, s = 10\} \; */$



The loop invariant separates the two behaviours again.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Accelerating the Fixpoint-Calculation

Analyzing a loop using polyhedra:

- Requires repeated analysis of the loop body.
- Needs acceleration to ensure practical analysis times.
- Apply Widening in one node of every loop (SCC).

Accelerating the Fixpoint-Calculation

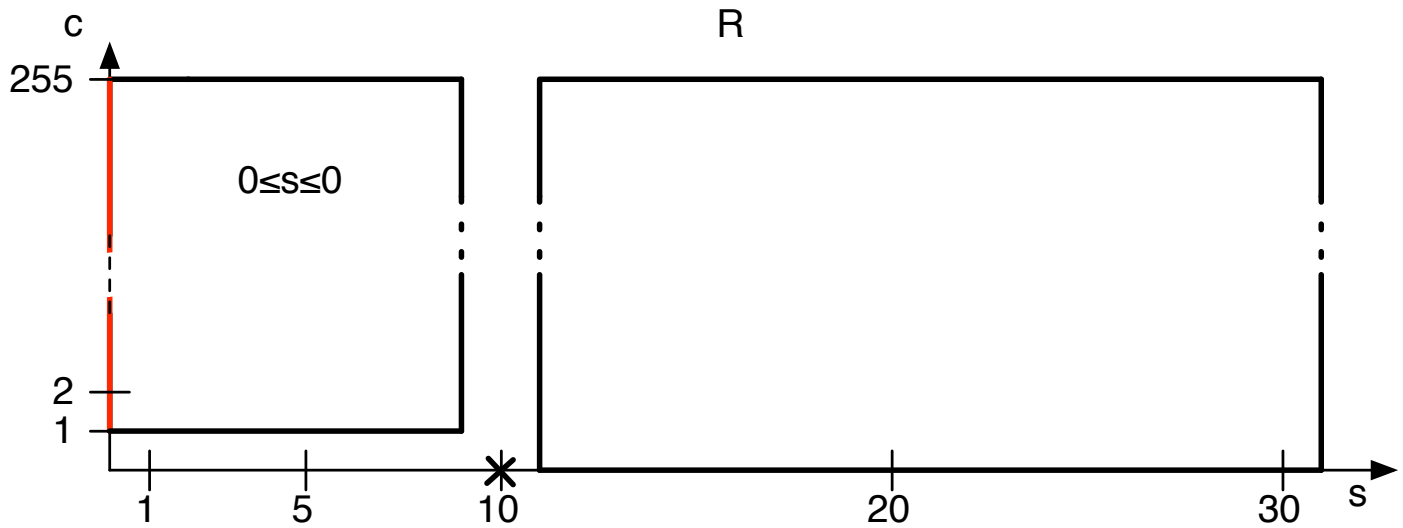
Analyzing a loop using polyhedra:

- Requires repeated analysis of the loop body.
- Needs acceleration to ensure practical analysis times.
- Apply Widening in one node of every loop (SCC).
- Idea: Given iterates Q_{i-1} and Q_i, calculate $Q_{i+1} = Q_{i-1} \nabla Q_i$ by removing unstable bounds from Q_{i-1}.

Accelerating the String-Buffer Example

Applying Widening to the Example:

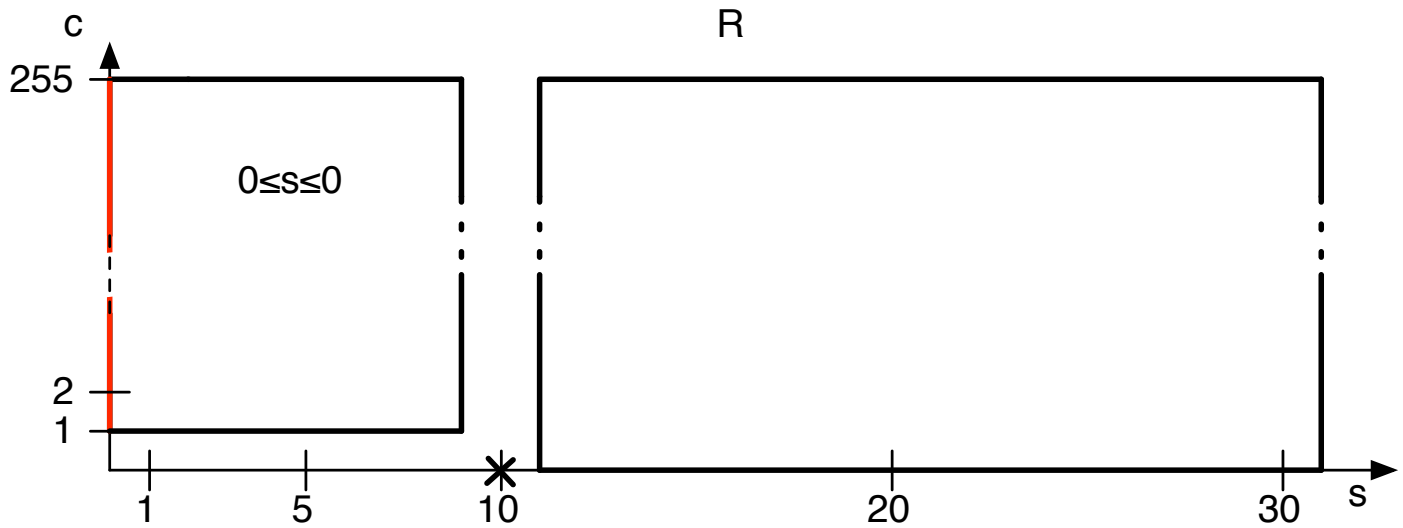
- Calculate $Q_{i+1} = Q_{i-1} \nabla Q_i$ by removing unstable bounds in Q_{i-1}.



Accelerating the String-Buffer Example

Applying Widening to the Example:

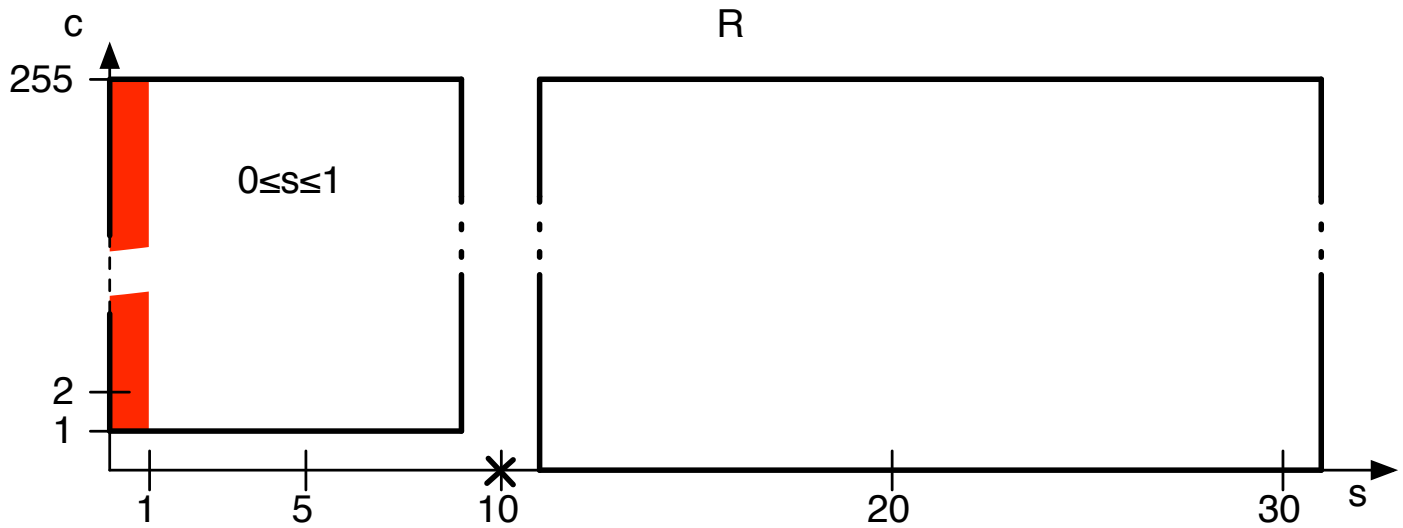
- Calculate $Q_{i+1} = Q_{i-1} \nabla Q_i$ by removing unstable bounds in Q_{i-1}.
- $Q_0 \equiv \{0 \leq s \leq 0\}$



Accelerating the String-Buffer Example

Applying Widening to the Example:

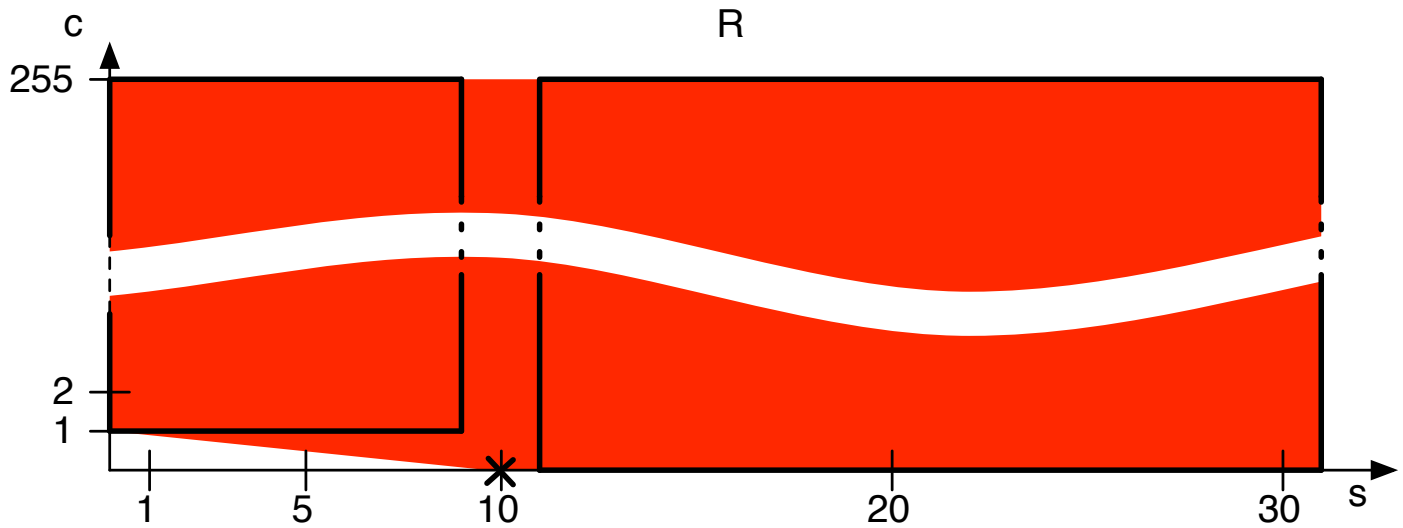
- Calculate $Q_{i+1} = Q_{i-1} \nabla Q_i$ by removing unstable bounds in Q_{i-1}.
- $Q_0 \equiv \{0 \leq s \leq 0\}$
- $Q_1 \equiv \{0 \leq s \leq 1\}$



Accelerating the String-Buffer Example

Applying Widening to the Example:

- Calculate $Q_{i+1} = Q_{i-1} \nabla Q_i$ by removing unstable bounds in Q_{i-1}.
- $Q_0 \equiv \{0 \leq s \leq 0\}$
- $Q_1 \equiv \{0 \leq s \leq 1\}$
- $Q_2 \equiv \{0 \leq s\}$



Recovering Precision: Narrowing

Narrowing: recover precision after widening.

- calculate $Q_{i+1} = Q_{i-1} \sqcap Q_i$ for a few loop iterations

Recovering Precision: Narrowing

Narrowing: recover precision after widening.

- calculate $Q_{i+1} = Q_{i-1} \sqcap Q_i$ for a few loop iterations
- has no effect in example

Recovering Precision: Narrowing

Narrowing: recover precision after widening.

- calculate $Q_{i+1} = Q_{i-1} \sqcap Q_i$ for a few loop iterations
- has no effect in example
- can recover precision for `for (i=0; i<100; i++) {}`:
  - $Q_0 \equiv \{i = 0\}$, $Q_1 \equiv \{0 \leq i \leq 1\}$

Recovering Precision: Narrowing

Narrowing: recover precision after widening.

- calculate $Q_{i+1} = Q_{i-1} \sqcap Q_i$ for a few loop iterations
- has no effect in example
- can recover precision for `for (i=0; i<100; i++) {}`:
  - $Q_0 \equiv \{i = 0\}$, $Q_1 \equiv \{0 \leq i \leq 1\}$
  - Widen: $Q_2 \equiv \{0 \leq i\}$

Recovering Precision: Narrowing

Narrowing: recover precision after widening.

- calculate $Q_{i+1} = Q_{i-1} \sqcap Q_i$ for a few loop iterations
- has no effect in example
- can recover precision for `for (i=0; i<100; i++) {}`:
  - $Q_0 \equiv \{i = 0\}$, $Q_1 \equiv \{0 \leq i \leq 1\}$
  - Widen: $Q_2 \equiv \{0 \leq i\}$
  - Narrow: $Q_3 \equiv \{0 \leq i \leq 99\}$

Recovering Precision: Narrowing

Narrowing: recover precision after widening.

- calculate $Q_{i+1} = Q_{i-1} \sqcap Q_i$ for a few loop iterations
- has no effect in example
- can recover precision for `for (i=0; i<100; i++) {}`:
  - $Q_0 \equiv \{i = 0\}$, $Q_1 \equiv \{0 \leq i \leq 1\}$
  - Widen: $Q_2 \equiv \{0 \leq i\}$
  - Narrow: $Q_3 \equiv \{0 \leq i \leq 99\}$
- cannot recover precision for `for (i=0; i!=100; i++) {}`

Recovering Precision: Narrowing

Narrowing: recover precision after widening.

- calculate $Q_{i+1} = Q_{i-1} \sqcap Q_i$ for a few loop iterations
- has no effect in example
- can recover precision for `for (i=0; i<100; i++) {}`:
  - $Q_0 \equiv \{i = 0\}$, $Q_1 \equiv \{0 \leq i \leq 1\}$
  - Widen: $Q_2 \equiv \{0 \leq i\}$
  - Narrow: $Q_3 \equiv \{0 \leq i \leq 99\}$
- cannot recover precision for `for (i=0; i!=100; i++) {}`
- other difficulties due to non-monotone state growth

Widening with Landmarks ($n = 10$ omitted for brevity):

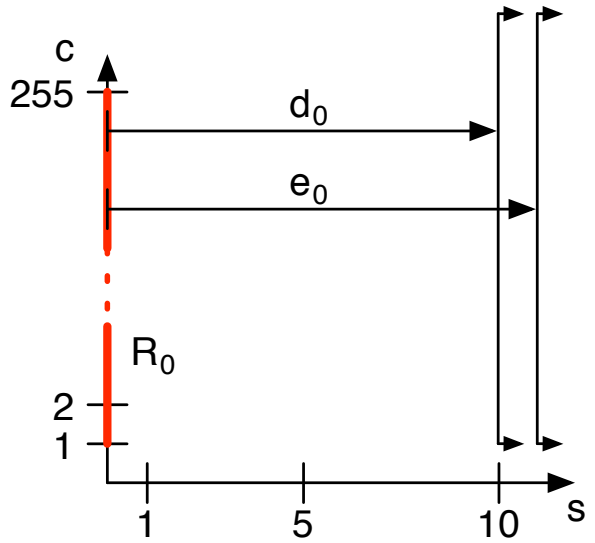
```
char s[32] = "the string";
while (true) {
   /*  Q0 ≡ {s = 0} */
   c = *s;
   /*R = ∅*/
   if (c==0) break;
   /*  T = ∅*/
   s++;
   /*  U = ∅*/
};
/* S = ∅ */
```

Calculate $Q_0 = P \sqcup U = P$.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Widening with Landmarks ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /*  Q0 ≡ {s = 0} */
   c = *s;
   /* R0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /*  T = ∅*/
   s++;
   /*  U = ∅*/
};
/* S = ∅ */
```

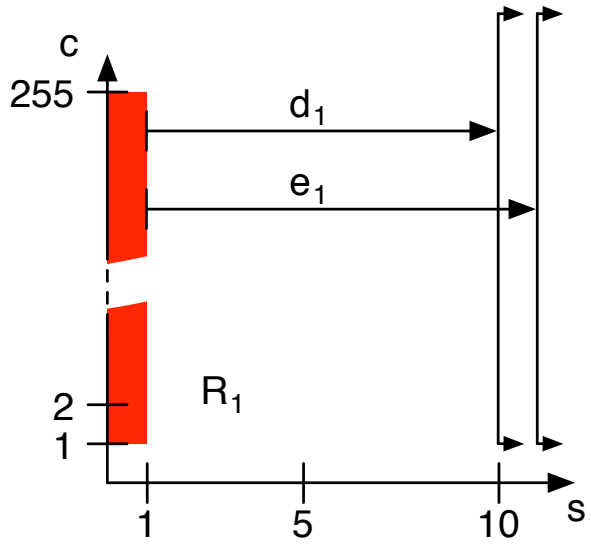


Measure distance d_0, e_0 to unsatisfiable inequalities.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Widening with Landmarks ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /*  Q1 ≡ {0 ≤ s ≤ 1} */
   c = *s;
   /* R1 ≡ {0 ≤ s ≤ 1, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /*  T0 ≡ {s = 0, 1 ≤ c ≤ 255}*/
   s++;
   /*  U0 ≡ {s = 1, 1 ≤ c ≤ 255}*/
};
/* S = ∅ */
```

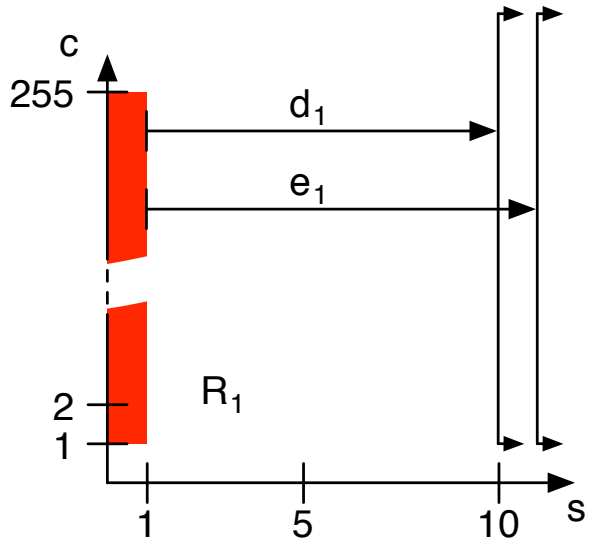


Measure a second sample d_1, e_1.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Widening with Landmarks ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /*  Q2 ≡ {0 ≤ s ≤ 2} */
   c = *s;
   /*R1 ≡ {0 ≤ s ≤ 1, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /*  T1 ≡ {0 ≤ s ≤ 1, 1 ≤ c ≤ 255}*/
   s++;
   /* U1 ≡ {1 ≤ s ≤ 2, 1 ≤ c ≤ 255}*/
};
/* S = ∅ */
```

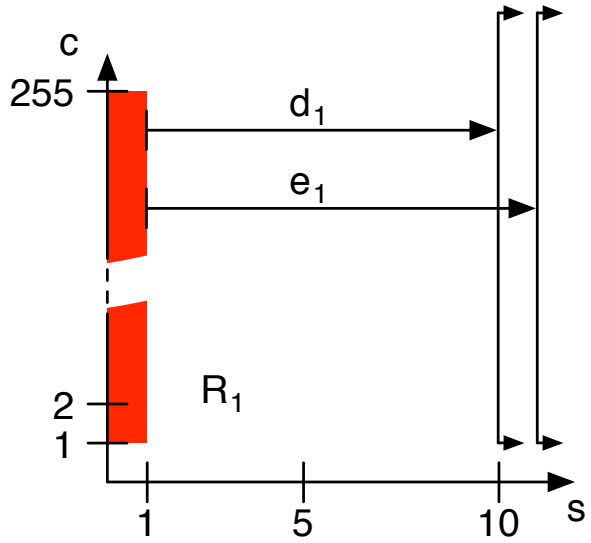


Relax inequalities in Q_1 until closest landmark is reached

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Widening with Landmarks ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /*  Q3 ≡ {0 ≤ s ≤ 10} */
   c = *s;
   /*R1 ≡ {0 ≤ s ≤ 1, 1 ≤ c ≤ 255}*/
   if (c==0) break;
   /*  T1 ≡ {0 ≤ s ≤ 1, 1 ≤ c ≤ 255}*/
   s++;
   /* U1 ≡ {1 ≤ s ≤ 2, 1 ≤ c ≤ 255}*/
};
/* S = ∅ */
```

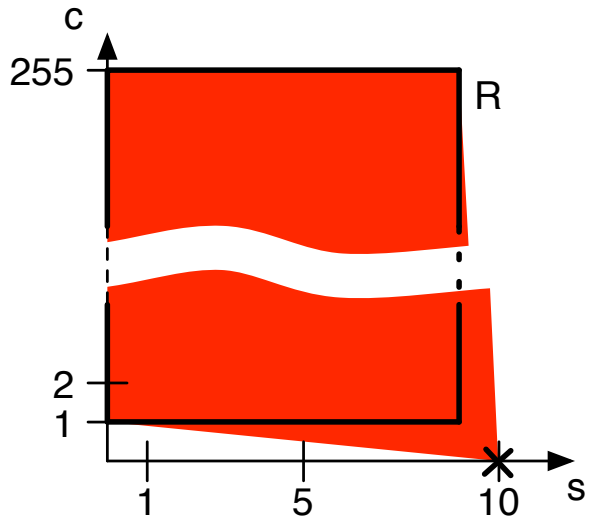


Relax $d_1/(d_0 - d_1)$ times difference of $s \leq 0$, $s \leq 1$.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \leq c \leq 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \leq c \leq 255\} \end{aligned}$$

Widening with Landmarks ($n = 10$ omitted for brevity):

```
char s[32] = "the string";
while (true) {
   /* Q3 ≡ {0 ≤ s ≤ 10} */
   c = *s;
   /* R3 ≡ {c ≤ 255, i + 10c ≥ 10,  */
                255i + c ≤ 2550}
   if (c==0) break;
   /*  T1 ≡ {0 ≤ s ≤ 1, 1 ≤ c ≤ 255}*/
   s++;
   /* U1 ≡ {1 ≤ s ≤ 2, 1 ≤ c ≤ 255}*/
};
/* S = ∅ */
```



The extrapolated state enables the next behavior.

$$\begin{aligned} R &= \exists_c(Q) \sqcap \{s < n\} \sqcap \{1 \le c \le 255\} \\ &\sqcup \exists_c(Q) \sqcap \{s = n\} \sqcap \{c = 0\} \\ &\sqcup \exists_c(Q) \sqcap \{s > n\} \sqcap \{0 \le c \le 255\} \end{aligned}$$

Widening with Landmarks

Landmark: inequality which renders a polyhedron empty on intersection.

- Track landmarks in two consecutive loop iterations.
- Extrapolate changing inequalities at widening point.
- Apply standard widening if no more landmarks are found.

Widening with Landmarks

Landmark: inequality which renders a polyhedron empty on intersection.

- Track landmarks in two consecutive loop iterations.
- Extrapolate changing inequalities at widening point.
- Apply standard widening if no more landmarks are found.

Observations:

- Need two loop iterations to perform linear extrapolation.
- Landmarks can contain variables not in scope at widening point.
- Measuring distance may have no overhead.

Related Work: Cousot et al.

Narrowing [Cousot and Halbwachs, POPL 78]:

- Insufficient precision.
- Difficult to implement.

Related Work: Cousot et al.

Narrowing [Cousot and Halbwachs, POPL 78]:

- Insufficient precision.
- Difficult to implement.

Widening with thresholds [Cousot et al., ESOP 05]:

- Manually set thresholds for extrapolation.
- Can only deal with bounds on single variables.

Related Work: Gopan and Reps

Lookahead Widening [Gopan, Reps, CAV '06]:

- Use 2 polyhedra (pilot and main).
- Perform widening/narrowing on pilot.
- Only evaluate behaviors that are enabled w.r.t. main.
- Promote pilot to main once stable.

Related Work: Gopan and Reps

Lookahead Widening [Gopan, Reps, CAV '06]:

- Use 2 polyhedra (pilot and main).
- Perform widening/narrowing on pilot.
- Only evaluate behaviors that are enabled w.r.t. main.
- Promote pilot to main once stable.

Compared to Widening with Landmarks:

- Can use off-the-shelf polyhedra libraries.
- Uses two polyhedra (but can often share one).
- More algebraic.

Related Work: Gonnord and Halbwachs

Widening and Acceleration [Gonnord and Halbwachs, SAS '06]:

- Geared towards timed automata.
- Infer displacement (delta) of state for different paths.
- Accelerate state growth for each delta.

Related Work: Gonnord and Halbwachs

Widening and Acceleration [Gonnord and Halbwachs, SAS '06]:

- Geared towards timed automata.
- Infer displacement (delta) of state for different paths.
- Accelerate state growth for each delta.

Compared to Widening with Landmarks:

- Set of paths in C programs not obvious.
- Not clear where loop invariants reside in programs.
- Cannot analyze programs with deltas (pointer dereferences).

Conclusion

Widening with Landmarks:

- Measure distance to unsatisfiable inequalities.
- Extrapolate until nearest unsatisfiable inequality becomes satisfiable.

Properties:

- Consecutive test for fixpoint when reaching next behavior.
- Low overhead.
- Fully automatic.
- Precise for linear state space growth.