Precise widenings for proving termination by abstract interpretation

Nathanaël Courant

July 26, 2016
Function: a termination prover using abstract interpretation

- Improve its widening operator
Context – Abstract interpretation

- Statically infer properties of programs
- *Abstract* a set of states
- *Interpret* the program with abstract values
- Using a *widening* operator to accelerate (post-)fixpoint computation.
```c
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```
```c
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};  
}
```
```c
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```
```c
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) { }
}
```
```c
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```

Nathanaël Courant
Precise widenings
July 26, 2016
```c
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```
Example – Loop

```c
int main() {
    int x;
    while (x > 0) {
        x--;
    }
}
```
```c
int main() {
    int x;
    while (x > 0) {
        x--;  
    }
}
```
Example – Loop

```c
int main() {
    int x;
    while (x > 0) {
        x--;
    }
}
```

⊥
```c
int main() {
    int x;
    while (x > 0) {
        x--;  // x ⩾ 1
    }
}
```
Example – Loop

```c
int main() {
    int x;
    while (x > 0) {
        x--;
    }
}
```

Nathanaël Courant
Precise widenings
July 26, 2016
Example – Loop

```c
int main() {
    int x;
    while (x > 0) {
        x--;
    }
}
```

\[
x \geq 1
\]

\[
x \geq 2
\]

\[
1
\]
Example – Loop

```c
int main() {
    int x;
    while (x > 0) {
        x--;
    }
}
```

```
x \geq 2
```

```
x \geq 3
```

```
⊥
```

```
4
```

```
2
```

Precise widenings

Nathanaël Courant

July 26, 2016
Example – Loop

```c
int main() {
    int x;
    while (x > 0) {
        x--;
    }
}
```

And now?
Comparing

Approximation order \preceq

Computational order \sqsubseteq

\top \quad f \quad \bot

f

\bot
\[ y_0 = \bot \]

\[ y_{n+1} = \begin{cases} y_n & \text{if } \phi(y_n) \sqsubseteq y_n \\
\text{and } \phi(y_n) \preceq y_n \\
y_n \triangledown \phi(y_n) & \text{otherwise} \end{cases} \]
Check for case A: if \( f_1 \not\sqsupseteq f_2 \), replace \( f_2 \) by \( \top \).

Perform *left unification*: keep only nodes occurring in \( t_1 \).

Check for cases B and C: if \( f_1 \) defined and \( f_2 \not\ll f_1 \), replace \( f_2 \) by \( \top \). This is \( f_1 \triangledown f_2 \).

If \( f_1 \) not defined and \( f_2 \) is, extend \( f_2 \) towards adjacent segments in \( t_1 \).
Widening

\[
\begin{align*}
x & \geq 1 \\
x & \geq 2 \quad 1 \\
\bot & \quad 3 \\
x & \geq 1 \\
x & \geq 2 \quad 1 \\
x & \geq 3 \quad 3 \\
\bot & \quad 5 \\
x & \geq 1 \\
x & \geq 2 \\
5 & \quad 3 \\
2x - 1 & \quad 3 \\
x & \geq 1 \\
x & \geq 2 \\
1 & \\
\end{align*}
\]
int main() {
    int x, y;
    while (x > 0 || y > 0) {
        x--;
        y--;
    }
}
Retrying when prediction was incorrect

- Check for case A: if $f_1 \not\sqsubseteq f_2$, replace $f_2$ by $\top$.
- Perform *left unification*: keep only nodes occurring in $t_1$.
- Check for cases B and C: if $f_1$ defined and $f_2 \not\ll f_1$, replace $f_2$ by $\top$. This is $f_1 \triangleleft f_2$.
- If $f_1$ not defined and $f_2$ is, extend $f_2$ towards adjacent segments in $t_1$.

\[
\begin{align*}
  f_1 \triangleleft f_2 &= \\
  &= \begin{cases} 
  f_2 & \text{the first } b \text{ times} \\
  \top & \text{or if } f_1 \text{ is not defined} \\
  & \text{or if } f_2 \ll f_1 \\
  \top & \text{otherwise}
  \end{cases}
\end{align*}
\]
Evolving rays

```c
int main() {
    int x, y;
    if (y > 0) {
        while (x > 0) {
            x -= y;
        }
    }
}
```
Check for case A: if $f_1 \nsubseteq f_2$, replace $f_2$ by $\top$.

Perform *left unification*: keep only nodes occurring in $t_1$.

Check for cases B and C: if $f_1$ defined and $f_2 \not\ll f_1$, replace $f_2$ by $\top$. This is $f_1 \triangledown f_2$.

If $f_1$ not defined and $f_2$ is, extend $f_2$ towards adjacent segments in $t_1$.

Compute a set of allowed nodes: *evolve* the constraints in $t_1$ towards their neighbours.

*Replace* not allowed nodes in $t_2$ by some allowed nodes.

Heuristics to reduce the number of allowed nodes.
Evolving rays

\( \text{evolve}(u, v) = w \)

\( w_i = \begin{cases} 
0 & \text{if } \exists j \in [1, n] / (u_i v_j - u_j v_i) u_i u_j < 0 \\
u_i & \text{otherwise}
\end{cases} \)
Other improvements

- Extending towards relevant segments instead of adjacent
- Rational coefficients
- More precise backwards assignment operator
Results

- Only with retrying.
- Only with refining.
- Only without refining.
- Only with improved backwards assignment.
- Only without improved backwards assignment.

Nathanaël Courant
Precise widenings
July 26, 2016
Conclusion

- Automated version much more efficient
- No unique widening better than all others