Precise widenings for proving termination by abstract interpretation

Nathanaël Courant

July 26, 2016

Context

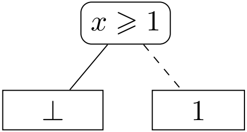
- FUNCTION: a termination prover using *abstract interpretation*
- Improve its widening operator

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.

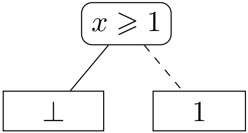
Example

```
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```



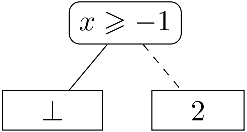
Example – Assignment

```
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```



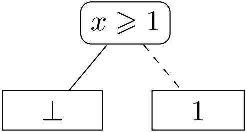
Example – Assignment

```
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```



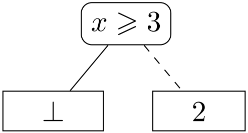
Example – Assignment

```
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```



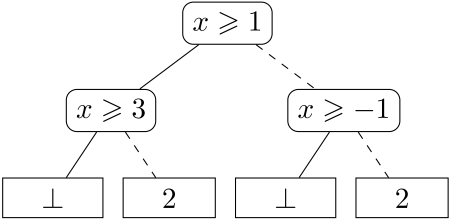
Example – Assignment

```
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```



Example – Condition

```
int main() {
    int x;
    if (x > 0) {
        x -= 2;
    } else {
        x += 2;
    }
    while (x > 0) {};
}
```



Example – Loop

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

0

Example – Loop

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

$\bot$

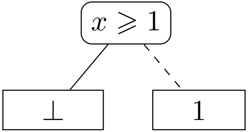
Example – Loop

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

$\bot$

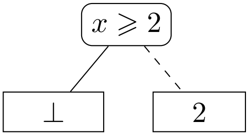
Example – Loop

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



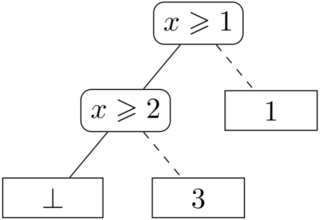
Example – Loop

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



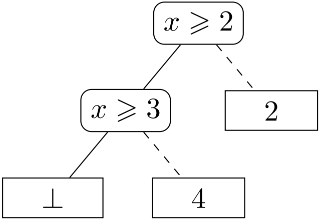
Example – Loop

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



Example – Loop

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

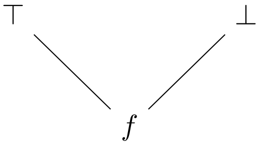


Example – Loop

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

And now?

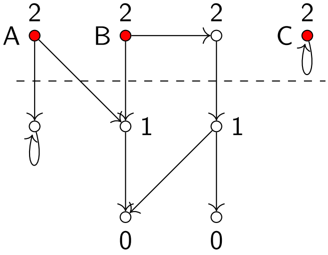
Comparing



Approximation order $\preccurlyeq$

Computational order $\sqsubseteq$

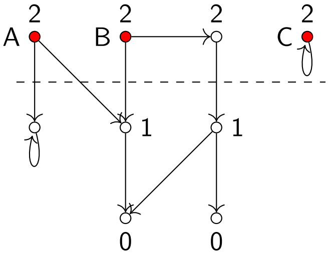
Widening



$$y_0 = \bot$$

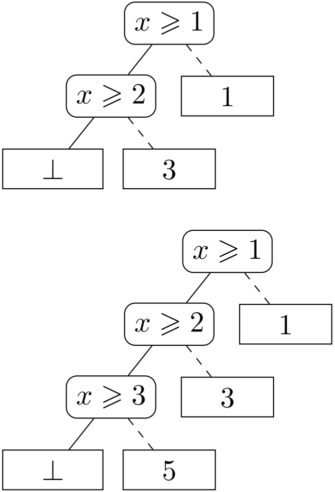
$$y_{n+1} = \begin{cases} y_n & \text{if } \phi(y_n) \sqsubseteq y_n \\ & \text{and } \phi(y_n) \preccurlyeq y_n \\ y_n \nabla \phi(y_n) & \text{otherwise} \end{cases}$$

Widening

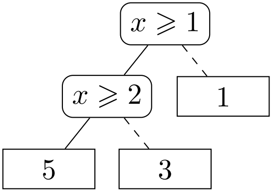


- Check for case A: if $f_1 \not\sqsubseteq f_2$, replace f_2 by $\top$.
- Perform *left unification*: keep only nodes occuring in t_1.
- Check for cases B and C: if f_1 defined and $f_2 \not\preccurlyeq f_1$, replace f_2 by $\top$. This is $f_1 \blacktriangledown f_2$.
- If f_1 not defined and f_2 is,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

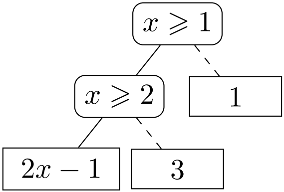
Widening



Left unification

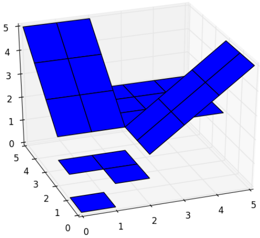


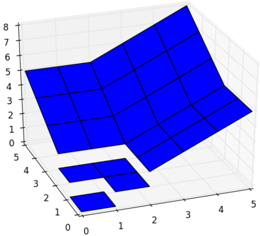
Result



Retrying when prediction was incorrect

```
int main() {
    int x, y;
    while (x > 0 || y > 0) {
        x--;
        y--;
    }
}
```





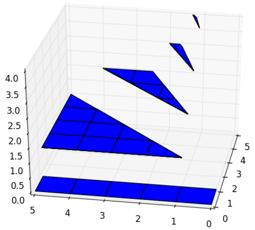
Retrying when prediction was incorrect

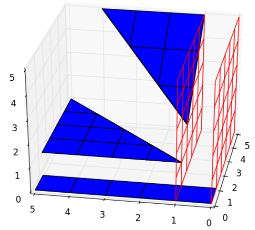
- Check for case A: if $f_1 \not\sqsubseteq f_2$, replace f_2 by $\top$.
- Perform *left unification*: keep only nodes occuring in t_1.
- Check for cases B and C: if f_1 defined and $f_2 \not\preccurlyeq f_1$, replace f_2 by $\top$. This is $f_1 \blacktriangledown f_2$.
- If f_1 not defined and f_2 is,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

$$f_1 \blacktriangledown f_2 = \begin{cases} f_2 & \begin{array}{l} \text{the first } b \text{ times} \\ \text{or if } f_1 \text{ is not defined} \\ \text{or if } f_2 \preccurlyeq f_1 \end{array} \\ \top & \text{otherwise} \end{cases}$$

Evolving rays

```
int main() {
    int x, y;
    if (y > 0) {
        while (x > 0) {
            x -= y;
        }
    }
}
```





Evolving rays

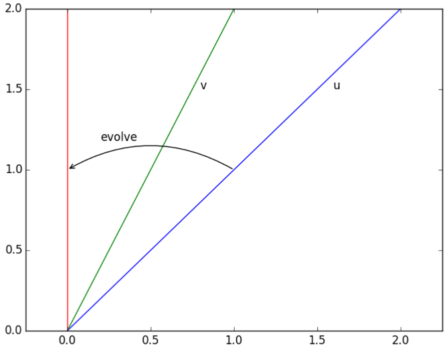
- Check for case A: if $f_1 \not\sqsubseteq f_2$, replace f_2 by $\top$.
- Perform *left unification*: keep only nodes occuring in t_1.
- Check for cases B and C: if f_1 defined and $f_2 \not\preccurlyeq f_1$, replace f_2 by $\top$. This is $f_1 \triangledown f_2$.
- If f_1 not defined and f_2 is, extend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

$$\textbf{evolve}(u, v) = w$$

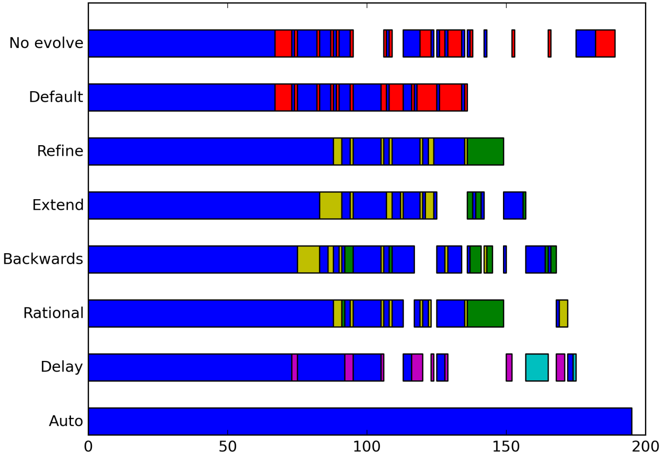
$$w_i = \begin{cases} 0 & \text{if } \exists j \in [\![1, n]\!] / \\ & \quad (u_i v_j - u_j v_i) u_i u_j < 0 \\ u_i & \text{otherwise} \end{cases}$$



Other improvements

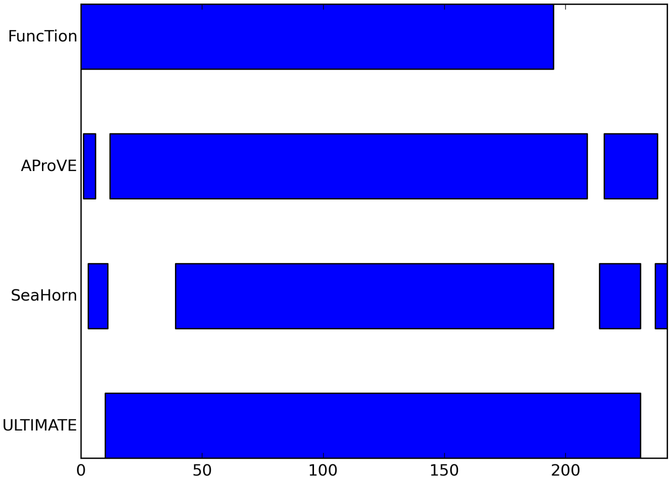
- Extending towards relevant segments instead of adjacent
- Rational coefficients
- More precise backwards assignment operator

Results



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

Results



Conclusion

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