Precise widenings for proving termination by abstract interpretation

Nathanaël Courant

July 28, 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.
```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) {};
}
```

The diagram illustrates the possible values for `x` after the program executes.

- `x ≥ 1` is a possible value after the program finishes.
- There are two possibilities for `x`:
  - `⊥` (undefined)
  - `1` (true)

The program ensures that `x` eventually becomes greater than or equal to 1.
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;
  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--;
    }
}
```
Example – Loop

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

Nathanaël Courant
Precise widenings
July 28, 2016
```c
int main() {
    int x;
    while (x > 0) {
        x--;
    }
}
```

```
x \geq 1
```

```
x \geq 2
```

```
1
```

```
3
```

```
⊥
```

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

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

And now?
Comparing

Approximation order \( \preceq \)

Computational order \( \sqsubseteq \)

\( \top \quad \quad \quad \quad \bot \quad \quad \quad \bot \)

\( f \quad \quad \quad \quad f \)

\( f \)

\( \bot \)
\[ y_0 = \bot \]

\[ y_{n+1} = \begin{cases} 
  y_n & \text{if } \phi(y_n) \sqsubseteq y_n \\
  y_{n \triangledown \phi(y_n)} & \text{otherwise}
\end{cases} \]

\text{if } \phi(y_n) \sqsubseteq y_n \\
\text{and } \phi(y_n) \preceq y_n \\
\text{otherwise}
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 $f_1 \llwedge f_2$.

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

\[
\begin{align*}
&x \geq 1 \\
&\quad \begin{cases} 
  x \geq 2 & 1 \\
  x \geq 3 & 3 \\
\end{cases} \\
&\quad \begin{cases} 
  \bot & 5 \\
\end{cases}
\end{align*}
\]

Left unification

\[
\begin{align*}
&x \geq 1 \\
&\quad \begin{cases} 
  x \geq 2 & 1 \\
\end{cases} \\
&\quad \begin{cases} 
  5 & 3 \\
\end{cases}
\end{align*}
\]

Result

\[
\begin{align*}
&x \geq 1 \\
&\quad \begin{cases} 
  x \geq 2 & 1 \\
\end{cases} \\
&\quad \begin{cases} 
  2x - 1 & 3 \\
\end{cases}
\end{align*}
\]

Nathanaël Courant
Retrying when prediction was incorrect

```c
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 $f_1 \triangledown f_2$.
- If $f_1$ not defined and $f_2$ is, extend $f_2$ towards adjacent segments in $t_1$.

$$f_1 \triangledown f_2 =$$

\[
\begin{cases} 
    f_2 & \text{the first } b \text{ times} \\
    \top & \text{or if } f_1 \text{ is not defined}
    \text{or if } f_2 \preceq f_1
\end{cases}
\]
```c
int main() {
    int x, y;
    if (y > 0) {
        while (x > 0) {
            x -= y;
        }
    }
}
```
Check for case A: if $f_1 \not\subseteq f_2$, 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 $f_2 \not\preceq f_1$, 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

- With retrying.
- With refining.
- Without refining.
- With improved backwards assignment.
- Without improved backwards assignment.
Conclusion

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