Program analysis in KeY

Tobias Gedell

gedell@cs.chalmers.se

Outline

- ▷ Who am I?
- ▷ Some words on static program analysis
- ▷ Reachable definitions analysis
- ▷ Why is this interesting for KeY?
- ▷ Presentation of mini-project
- ▷ Demo of mini-project

Work in progress!

This is work in progress!

Who am I?

My position:

- ▷ First year PhD student at Chalmers
- ▷ Supervised by Reiner Hähnle

My main interests are:

- ▷ Program languages
- ▷ Program analysis

What I have done in the past:

- ▷ Written a compiler for the functional language STG.
- ▷ Worked on and implemented a cache sensitive Haskell compiler.
- ▷ Worked on and implemented a usage analysis for Haskell in GHC.

Static program analysis

Program analyses are designed to calculate properties about programs. For example:

- ▷ Which variables are live at a certain program point?
- ▷ What values can a variable hold at a certain program point?
- ▷ Which functions are going to be called during program execution?

Static program analysis

Program analyses are designed to calculate properties about programs. For example:

- ▷ Which variables are live at a certain program point?
- ▷ What values can a variable hold at a certain program point?
- ▷ Which functions are going to be called during program execution?

A key point is that a program analysis must be decidable and computationally cheap, which is achieved by approximation.

Static program analysis

Program analyses are designed to calculate properties about programs. For example:

- ▷ Which variables are live at a certain program point?
- ▷ What values can a variable hold at a certain program point?
- ▷ Which functions are going to be called during program execution?

A key point is that a program analysis must be decidable and computationally cheap, which is achieved by approximation.

Program analysis	Verification (TP)
cheap	expensive
low precision	high precision

Reaching definitions

An assignment is reachable at a certain program point if there exist a program trace such that the variable has not been reassign when the point is reached.

Program: a $= 1;$ b $= 1;$ if(cond) $\{$b $= 1;\}$ else $\{$b $= 1;\}$

Reaching definitions

An assignment is reachable at a certain program point if there exist a program trace such that the variable has not been reassign when the point is reached.

Program: $\underbrace{\mathsf{a} = 1;}_{0}\ \underbrace{\mathsf{b} = 1;}_{1}\ \underbrace{\mathsf{if(cond)}\ \{\overbrace{\mathsf{b} = 1;}^{3}\}\ \mathsf{else}\ \{\overbrace{\mathsf{b} = 1;}^{4}\}}_{2}$

Reaching definitions

An assignment is reachable at a certain program point if there exist a program trace such that the variable has not been reassign when the point is reached.

Program: $\underbrace{\mathsf{a} = 1;}_{0}\ \underbrace{\mathsf{b} = 1;}_{1}\ \underbrace{\mathsf{if(cond)}\ \{\overbrace{\mathsf{b} = 1;}^{3}\}\ \mathsf{else}\ \{\overbrace{\mathsf{b} = 1;}^{4}\}}_{2}$

A solution: {(a, 0), (b, 3), (b, 4)}

Reaching definitions

An assignment is reachable at a certain program point if there exist a program trace such that the variable has not been reassign when the point is reached.

Program: $\underbrace{\mathsf{a} = 1;}_{0}\ \underbrace{\mathsf{b} = 1;}_{1}\ \underbrace{\mathsf{if(cond)}\ \{\overbrace{\mathsf{b} = 1;}^{3}\}\ \mathsf{else}\ \{\overbrace{\mathsf{b} = 1;}^{4}\}}_{2}$

A solution: {(a, 0), (b, 3), (b, 4)}

The solution must be a safe approximation! But we want to get the least solution possible.

Why is this interesting for KeY?

Program analysis can be used by the KeY system in at least two ways:

- We can integrate analyses to for example reduce branching.

```
void m1(SomeClass o)
{
 ...
 o.m2();
 ...
}
```

Why is this interesting for KeY?

Program analysis can be used by the KeY system in at least two ways:

- ▷ We can integrate analyses to for example reduce branching.

```
void m1(SomeClass o)
{
 ...
 o.m2();
 ...
}
```

- ▷ We can try to build a framework based on abstract interpretation which could allow the user to abstract away from certain things when making the proof. (This idea is still a little vague, more theoretical work is needed.)

Mini-project

In order to get going I started with trying to implement the reaching definitions analysis in KeY.

I choose to implement it for the WHILE language.

program	::=	*stmt*
stmt	::=	*var* $\overset{loc}{\mathbf{=}}$ *term*;
	|	**if**^{loc}**(***term***)** *stmt* **else** *stmt*
	|	**while**^{loc}**(***term***)** *stmt*
	|	**{***stmt****}**
term	::=	*literal*
	|	*var*
	|	*term op term*
	|	**(***term***)**

Reaching definitions - rules

$S \subseteq \mathsf{Var} \times \mathsf{Loc}$

```
functions {
 VarSet Union(VarSet, VarSet);
 VarSet Singleton(Quoted, ...);
 VarSet CutVar(VarSet, Quoted);
 VarSet Empty;
}
```

Reaching definitions - rules

RULE

$$\frac{\ldots}{S_0 \vdash stmt \ \Downarrow \ S_1}$$

```
predicates {
    wrapper(VarSet, VarSet);
}
```

```
rules {
 rdef_rule {
     find (==> diamond{{stmt}}(wrapper(vs0, vs1)))
         ...
 };
}
```

Reaching definitions - rules

ASSIGN

$$\frac{}{S \vdash x \overset{loc}{=} v\textbf{;} \Downarrow \underbrace{(S \backslash \{x, l \mid l \in Loc\})}_{\text{Remove previous assignments of } x} \cup \{x, loc\}}$$

```
rdef_assign  {
 find (==> diamond{{#var = #se;}}
        (wrapper(vs, Union( CutVar(vs, #var),
                            Singleton(#var, ...)))))
     close goal
};
```

Reaching definitions - rules

IF

$$\frac{S_0 \vdash stmt_0 \ \Downarrow \ S_1 \qquad S_0 \vdash stmt_1 \ \Downarrow \ S_2}{S_0 \vdash \textbf{if (}e\textbf{)}\ stmt_0 \ \textbf{else}\ stmt_1 \ \Downarrow \ S_1 \cup S_2}$$

$\leftarrow$ Approximation!

```
rdef_if_else {
 find (==> diamond{{if(#se) #stmt0 else #stmt1}}
    (wrapper(vs0, Union(vs1, vs2))))
   replacewith (==> diamond{{#stmt0}}(wrapper(vs0, vs1)));
   replacewith (==> diamond{{#stmt1}}(wrapper(vs0, vs2)))
};
```

Getting the variable names

First problem:

- ▷ I need to store variable names in formulas, which is not possible since variable names are flexible terms.

  Solved by adding a quoting mechanism. With it you can say: `varcond (#qvar quotes #var)`, which adds a constraint.

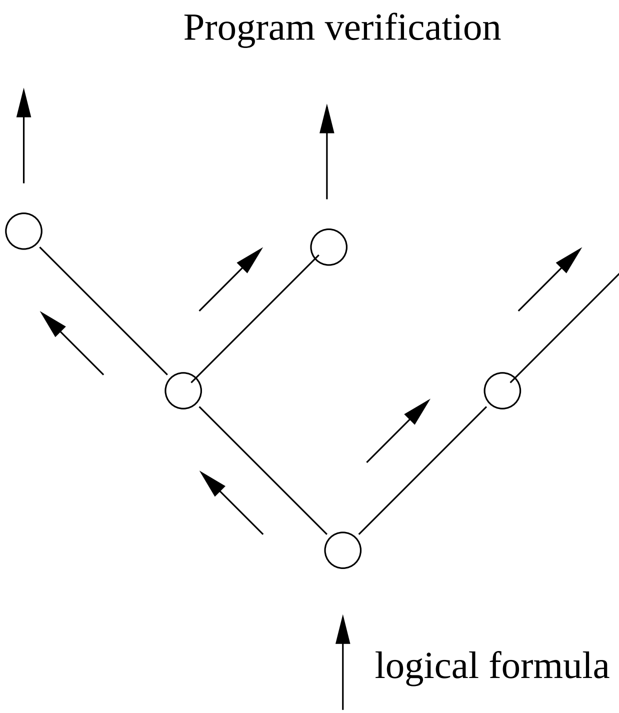
```
schema variables { Quoted #qvar; }

rdef_assign  {
 find (==> diamond{{#var = #se;}}
        (wrapper(vs, Union( CutVar(vs, #qvar),
                            Singleton(#qvar, ...)))))
     varcond (#qvar quotes #var)
     close goal
};
```

Flow of information

Second problem:

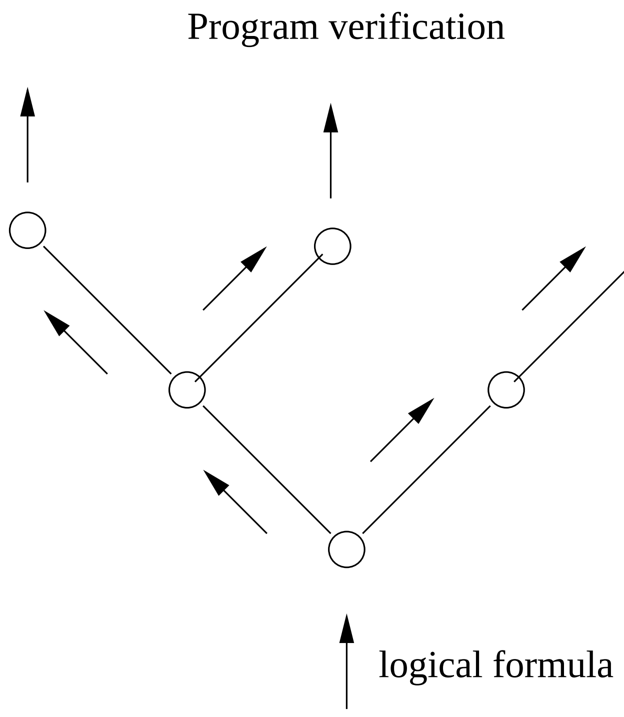
- ▷ Flow of information.

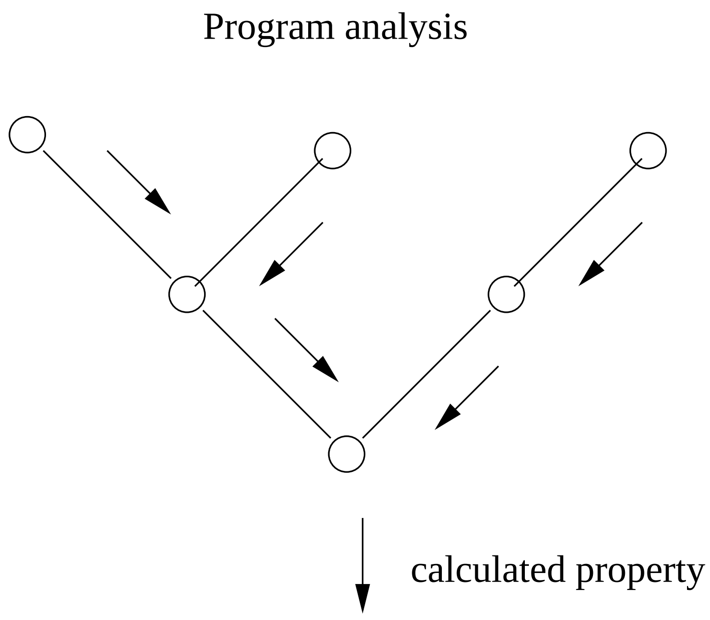


Flow of information

Second problem:

▷ Flow of information.



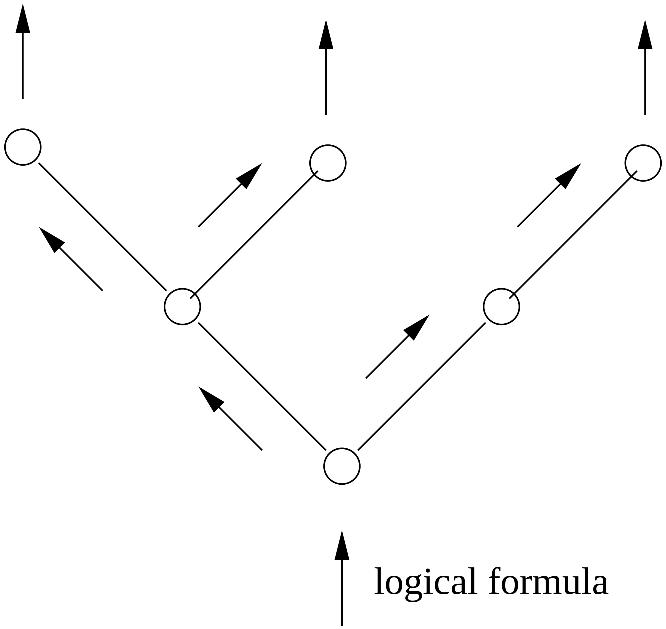


Flow of information

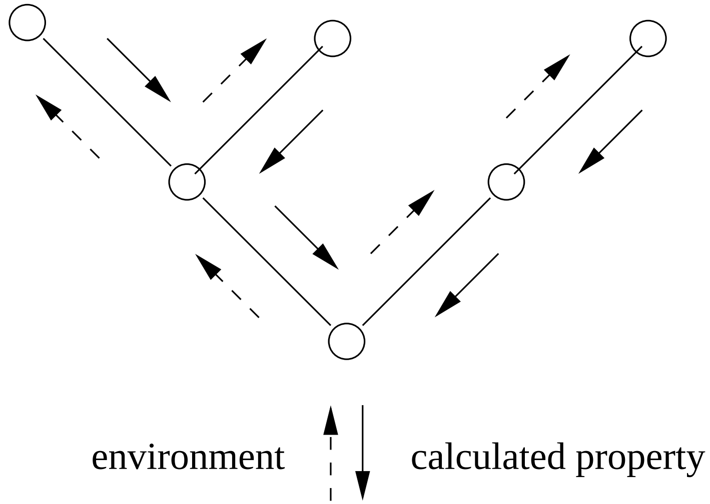
Second problem:

- Flow of information.

Program verification



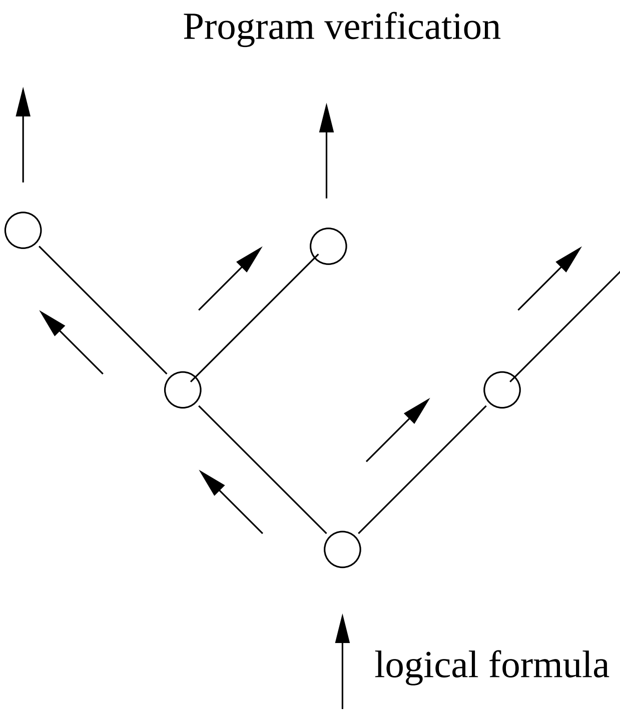
Program analysis

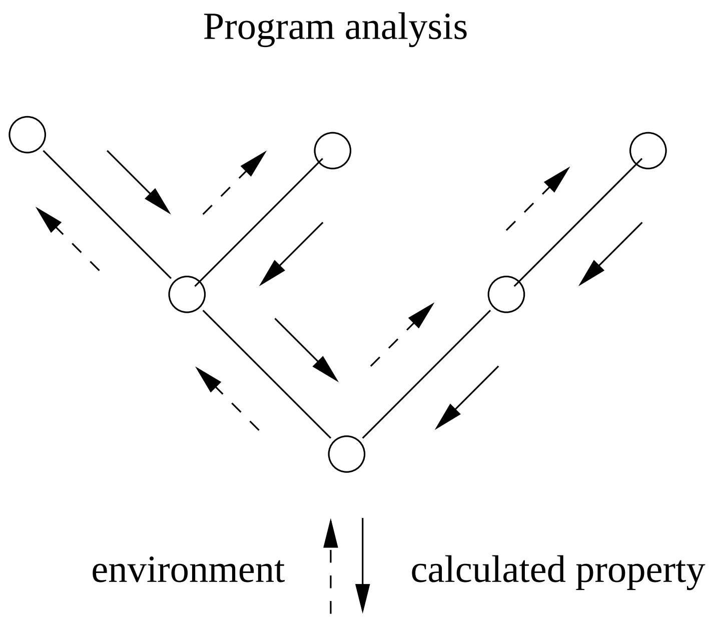


Flow of information

Second problem:

- Flow of information.





Program verification is like type checking while reaching definitions analysis is more like type inferens.

Flow of information

Meta variables solves the information flow problem.

```
problem {
  <{ if(cond) { a = 1; } else { b = 1; } }>
   wrapper(Empty, Union(..., ...))
}
```

Now I must give the solution...

Flow of information

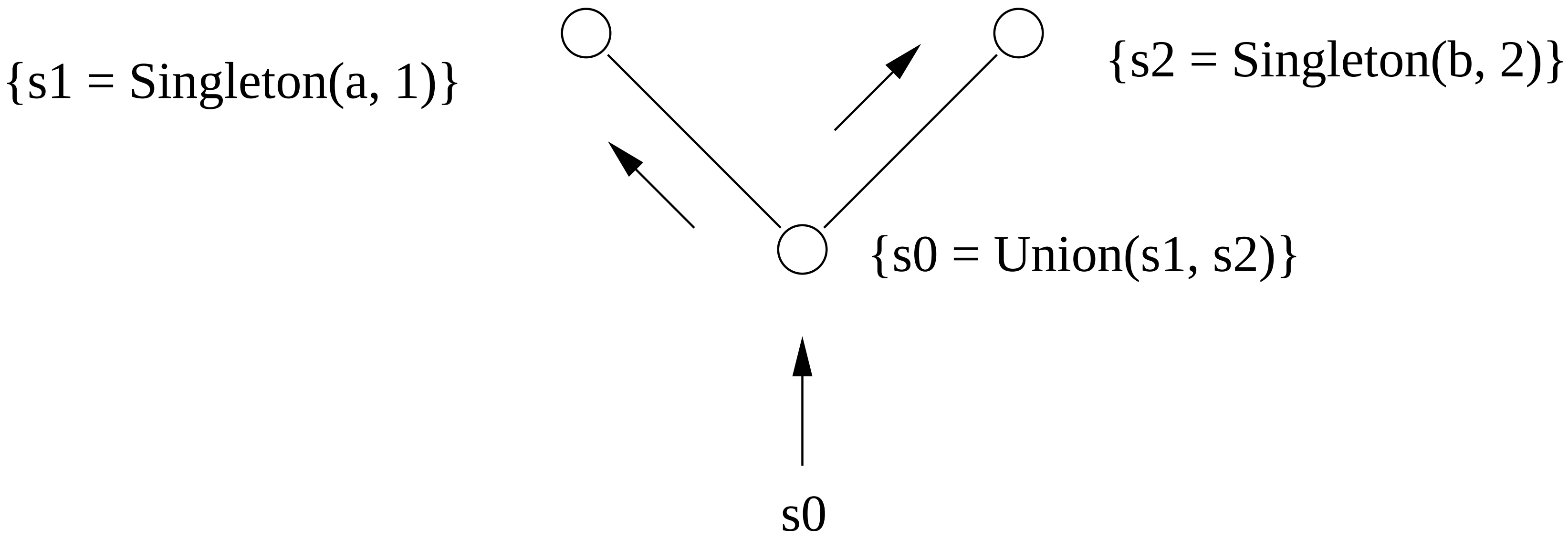
Meta variables solves the information flow problem.

```
problem {
 ex s:VarSet.
  <{ if(cond) { a = 1; } else { b = 1; } }>
   wrapper(Empty, s)
}
```

Now the system checks if there exists a solution!

Flow of information

```
problem {
 ex s:VarSet.
  <{ if(cond) { a = 1; } else { b = 1; } }>
   wrapper(..., s)
}
```



If I just get hold of the collection of constraints I can find the solution!

Demo

Time for a demo!

Demo - example 1

Program:

$$\underbrace{\mathsf{a} = 1;}_{0}\ \underbrace{\mathsf{b} = 1;}_{1}\ \underbrace{\mathsf{if(cond)}\ \{\overbrace{\mathsf{b} = 1;}^{3}\}}_{2}$$

Solution:

$$\{(a, 0), (b, 1), (b, 3)\}$$

Demo - example 2

Program:

$$\underbrace{\text{a} = 1;}_{0}\ \underbrace{\text{b} = 1;}_{1}\ \underbrace{\text{if(cond) } \{\overbrace{\text{b} = 1;}^{3}\} \text{ else } \{\overbrace{\text{b} = 1;}^{4}\ \overbrace{\text{c} = 1;}^{5}\}}_{2}$$

Solution:

$$\{(a, 0), (b, 3), (b, 4), (c, 5)\}$$

Demo - example 3

Program:

$$\underbrace{\text{a} = 1;}_{0}\ \underbrace{\text{b} = 1;}_{1}\ \underbrace{\text{if(cond) }\{\overbrace{\text{b} = 1;}^{3}\}\text{ else }\{\overbrace{\text{b} = 1;}^{4}\ \overbrace{\text{c} = 1;}^{5}\ \overbrace{\text{while(cond) }\overbrace{\text{c} = 1;}^{7}}^{6}\}}_{2}$$

Solution:

```
Entry_0 = {}
Entry_1 = {(a, 0)}
Entry_2 = {(a, 0), (b, 1)}
Entry_3 = {(a, 0), (b, 1)}
Entry_4 = {(a, 0), (b, 1)}
Entry_5 = {(a, 0), (b, 4)}
Entry_6 = {(a, 0), (b, 4), (c, 5), (c, 7)}
Entry_7 = {(a, 0), (b, 4), (c, 5), (c, 7)}
Exit_0 = {(a, 0)}
Exit_1 = {(a, 0), (b, 1)}
Exit_2 = {(a, 0), (b, 3), (b, 4), (c, 7)}
Exit_3 = {(a, 0), (b, 3)}
...
```

Conclusion

Conclusion:

- ▷ It is possible to implement syntax directed program analyses in KeY. (At least for simple languages.)
- ▷ It would be nice to do computation in the taclets.
- ▷ To me KeY + meta variables + constraints seems very similar to for example Prolog.

Future work

I will now look into:

- What analyses are useful to KeY?
- Can we create a framework based on abstract interpretation?
- If we can, how can the framework be used?