Loop Analysis in KeY

Tobias Gedell

gedell@cs.chalmers.se

Introduction

There are currently two ways of handling loops in KeY:

- **Symbolic execution - repeated unwinding of loops**

  Can be performed automatically by the system, but time consuming and not always possible.
- **Induction**

  Hard to use and cannot automatically be applied by the system.

Symbolic execution - drawbacks

Example:

```
for(int i = 0; i < a.length; i++) a[i] = i;
```

- **Time consuming:** If `a.length` is large than we need to execute the loop many times.
- **Not possible:** If `a.length` is unknown we do not know when to stop.

We want to do better!

Symbolic execution - how it works

```
for(int i = 0; i < a.length; i++) a[i] = i; ...
```

Symbolic execution - how it works

```
for(int i = 0; i < a.length; i++) a[i] = i; ... ⇝

{a[0] := 0}
for(int i = 1; i < a.length; i++) a[i] = i; ...
```

Symbolic execution - how it works

```
for(int i = 0; i < a.length; i++) a[i] = i; ... ⇝
```

```
{a[0] := 0}
for(int i = 1; i < a.length; i++) a[i] = i; ... ⇝
```

```
{a[0] := 0, a[1] := 1}
for(int i = 2; i < a.length; i++) a[i] = i; ...
```

Symbolic execution - how it works

```
for(int i = 0; i < a.length; i++) a[i] = i; ... ⇝
```

```
{a[0] := 0}
for(int i = 1; i < a.length; i++) a[i] = i; ... ⇝
```

```
{a[0] := 0, a[1] := 1}
for(int i = 2; i < a.length; i++) a[i] = i; ... ⇝
```

```
{a[0] := 0, a[1] := 1, a[2] := 2, ...} ...
```

Symbolic execution - how it works

`for(int i = 0; i < a.length; i++) a[i] = i; ...` $\leadsto$

`{a[0] := 0}`
`for(int i = 1; i < a.length; i++) a[i] = i; ...` $\leadsto$

`{a[0] := 0, a[1] := 1}`
`for(int i = 2; i < a.length; i++) a[i] = i; ...` $\leadsto$

`{a[0] := 0, a[1] := 1, a[2] := 2,` $\ldots$`} ...`

We iteratively construct the update describing all side-effects of the loop.

Symbolic execution - how it works

For this example the update could have been constructed in a much more direct way!

We can see that for each iteration of the loop the update `{a[I] := I}` will be added. We also know that these updates do not clash with each other.

We can, therefore, skip the execution of the loop and instead directly construct the update:

$$\{\forall I \in [0, \texttt{a.length} - 1].\ a[I] := I,\ \texttt{i} := \texttt{a.length}\}$$

Loop Analysis - the idea

The idea behind the new treatment of loops is that we systematically:

1. Calculate the update of the loop body and abstract over the value of the loop variable,

   ```
   {a[I] := I}
   ```

Loop Analysis - the idea

The idea behind the new treatment of loops is that we systematically:

1. Calculate the update of the loop body and abstract over the value of the loop variable,

   ```
   {a[I] := I}
   ```

2. calculate the range of the loop variable,

   ```
   [0, a.length - 1]
   ```

Loop Analysis - the idea

The idea behind the new treatment of loops is that we systematically:

1. Calculate the update of the loop body and abstract over the value of the loop variable,

   ```
   {a[I] := I}
   ```

2. calculate the range of the loop variable,

   ```
   [0, a.length - 1]
   ```

3. make sure that some properties are fulfilled by the loop,

   For example no clashing.

Loop Analysis - the idea

The idea behind the new treatment of loops is that we systematically:

1. Calculate the update of the loop body and abstract over the value of the loop variable,

   `{a[I] := I}`

2. calculate the range of the loop variable,

   `[0, a.length - 1]`

3. make sure that some properties are fulfilled by the loop,

   For example no clashing.

4. replace the loop by the abstracted update, quantified over by the range of the loop variable.

   $\{\forall I \in [0, \texttt{a.length} - 1].\ a[I] := I,\ \texttt{i} := \texttt{a.length}\}$

Loop Analysis - calculating the update

When calculating the abstract update for the loop body, there are mainly two ways to go:

- ▷ We can create a program analysis that calculates all assignments that are made.

  Pros: could be tailor made for specific purposes like checking for clashes.

  Cons: much implementation work, can already be done by KeY.

Loop Analysis - calculating the update

When calculating the abstract update for the loop body, there are mainly two ways to go:

- ▷ We can create a program analysis that calculates all assignments that are made.

  Pros: could be tailor made for specific purposes like checking for clashes.

  Cons: much implementation work, can already be done by KeY.
- ▷ We can also let KeY compute the update.

  Pros: little implementation work, can check additional properties.

Loop Analysis - soundness properties

Observation: What we want to do is quite similar to loop vectorization and parallelization. Instead of executing the loop in a sequential order we execute it in parallel.

This can only be done when some properties are fulfilled:

- ▷ The loop variable is monotonically increasing/decreasing. (The order of the updates must be clear.)

Loop Analysis - soundness properties

Observation: What we want to do is quite similar to loop vectorization and parallelization. Instead of executing the loop in a sequential order we execute it in parallel.

This can only be done when some properties are fulfilled:

- ▷ The loop variable is monotonically increasing/decreasing.
  (The order of the updates must be clear.)
- ▷ The loop condition is of the form, i *op* e, where the value of e is not modified by the loop body.
  (We need to be able to calculate the range of the loop variable.)

Loop Analysis - soundness properties

Observation: What we want to do is quite similar to loop vectorization and parallelization. Instead of executing the loop in a sequential order we execute it in parallel.

This can only be done when some properties are fulfilled:

- ▷ The loop variable is monotonically increasing/decreasing.
  (The order of the updates must be clear.)
- ▷ The loop condition is of the form, i *op* e, where the value of e is not modified by the loop body.
  (We need to be able to calculate the range of the loop variable.)
- ▷ The loop body does not terminate the loop by executing a `break`, raising an exception or something similar.

Loop Analysis - soundness properties

Observation: What we want to do is quite similar to loop vectorization and parallelization. Instead of executing the loop in a sequential order we execute it in parallel.

This can only be done when some properties are fulfilled:

- The loop variable is monotonically increasing/decreasing.
  (The order of the updates must be clear.)
- The loop condition is of the form, i *op* e, where the value of e is not modified by the loop body.
  (We need to be able to calculate the range of the loop variable.)
- The loop body does not terminate the loop by executing a `break`, raising an exception or something similar.
- There is no dependence between the loop iterations.

Loop Analysis - soundness properties

There are mainly two different kinds of dependencies:

`for(int i = 0; i <= 10; i++)` s: `a[i] = a[i - 1];`

s_v - the statement s where the loop variable has the value v.

- **Data dependence**
  A statement s_k writes to a location that is read by a statement s_l.

  If $k < l$,
  (`a[i] = a[i - 1]`),
  we cannot execute the loop in parallel.

  If $k > l$,
  (`a[i] = a[i + 1]`),
  we execute it in parallel and replace a on the RHS with an array containing the original values of a.

Loop Analysis - soundness properties

There are mainly two different kinds of dependencies:

- **Output dependence**
  A statement s_k writes to a location that is overwritten by a statement s_l.

  Both the cases where $k > l$ and $l < k$,
  (`a = f(i)`),
  can be handled by using a last-win clash semantics for the constructed quantified updates.

  We must only make sure that the updates comes in the right order.

Loop Analysis - benefits of KeY

Traditionally, in the field of loop vectorization and parallelization, the test for dependence gives just a boolean answer.

If some part of the program is unknown, it must approximate and say that there is a dependence.

We, on the other hand, have a theorem prover backing us up!

Loop Analysis - benefits of KeY

Traditionally, in the field of loop vectorization and parallelization, the test for dependence gives just a boolean answer.

If some part of the program is unknown, it must approximate and say that there is a dependence.

We, on the other hand, have a theorem prover backing us up!

Consider for example:

```
for(i = 0; i <= 10; i = i + 1) a[i] = b[i + c];
```

Loop Analysis - benefits of KeY

Traditionally, in the field of loop vectorization and parallelization, the test for dependence gives just a boolean answer.

If some part of the program is unknown, it must approximate and say that there is a dependence.

We, on the other hand, have a theorem prover backing us up!

Consider for example:

```
for(i = 0; i <= 10; i = i + 1) a[i] = b[i + c];
```

Question: Is there any dependence between the loop iterations?

Loop Analysis - benefits of KeY

Answer: It depends on the value of a, b and c.

```
for(i = 0; i <= 10; i = i + 1) a[i] = b[i + c];
```

If a and b are the same array and c is between -10 and -1 then there is a dependence.

Loop Analysis - benefits of KeY

Answer: It depends on the value of a, b and c.

```
for(i = 0; i <= 10; i = i + 1) a[i] = b[i + c];
```

If a and b are the same array and c is between -10 and -1 then there is a dependence.

Instead of giving up and approximating, we calculate a constraint describing when no dependence is present.

If we can then show that the constraint is fulfilled, we can replace the loop with a quantified update.

Loop Analysis - example

Example:

```
for(i = 0; i <= 10; i = i + 1) a[i] = b[i + c];
```

Constraint for non-dependence:

```
(a != b \/ 0 > 10 + c \/ -1 < c)
```

New rule for loops:

LOOP

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{<\{}..\ \textit{old-rule}(loop)\ ...\texttt{\}>}\phi, \Delta, c \\ \Gamma \vdash \texttt{<\{}..\ \textit{replace-by-update}(loop)\ ...\texttt{\}>}\phi, \Delta, !c\end{array}}{\Gamma \vdash \texttt{<\{}..\ loop\ ...\texttt{\}>}\phi, \Delta}$$

where

$$loop \equiv for(..;..;..)..$$

$$c = \textit{non-dependence-constraint}(loop)$$

Loop Analysis - In practice

How many loops can we handle with this method?

DeMoney & IButtonAPI & SafeApplet

Yes	Need extensions	No
3	3	4

Extensions:

- Transform `offset += LEN_KEY`
  into `offset = ` $\texttt{offset}_0$ `+ LEN_KEY * i`. (2)
- Create objects in updates. (1)

Loop Analysis - observations

- The dependence analysis is tailor made for solving constraints of a certain kind and the more information KeY gives to it, the better result do we get.

  The dependence analysis can be seen as a specialized prover for a limited subset of integer problems.
- We need quantified updates with a deterministic semantics for clashes—last-win clash semantics.
- Only works for a special class of loops.