

Verification of Loops by Parallelization

Reiner Hähnle
(joint work with Tobias Gedell)

European Science Foundation
Exploratory Workshop
Challenges in Java Program Verification
17 October 2006

Setting

Context

- ▶ Formal verification of functional properties of software
- ▶ Complex, imperative, object-based language
- ▶ Here: JAVA (ideas apply also to C, C++, C#)

Setting

Context

- ▶ Formal verification of functional properties of software
- ▶ Complex, imperative, object-based language
- ▶ Here: JAVA (ideas apply also to C, C++, C#)

The Bottleneck



- ▶ Loops and recursion require interaction
supply invariant, modify/strengthen induction hypothesis
- ▶ Induction/invariant rules for JAVA-like languages are complex
side effects, abrupt termination, aliasing
- ▶ High amount of technical knowledge required from user
logic syntax, calculus design

Avoid interaction as much as possible!

Avoid interaction as much as possible!

Some possibilities to automate loop verification

- ▶ Heuristic methods (rippling, analysis of failed proofs)
mostly in functional setting, but see Angela's talk
- ▶ Abstraction (software model checking)
incomplete, restricted expressivity of specs
- ▶ Approximation (extended static checking)
incomplete, unsound
- ▶ Invariant Generation
- ▶ Finite unwinding
number of iterations must be known and small

Deductive verification method handling well-behaved loops automatically

Contribution

Deductive verification method handling well-behaved loops automatically

Incomplete Parallelizable loops; but no abstraction when it works

Contribution

Deductive verification method handling well-behaved loops automatically

Incomplete Parallelizable loops; but no abstraction when it works

Robust No syntactic restriction on initialization, step, body

Deductive verification method handling well-behaved loops automatically

Incomplete Parallelizable loops; but no abstraction when it works

Robust No syntactic restriction on initialization, step, body

Sound Dependence analysis ensures soundness

Contribution

Deductive verification method handling well-behaved loops automatically

Incomplete Parallelizable loops; but no abstraction when it works

Robust No syntactic restriction on initialization, step, body

Sound Dependence analysis ensures soundness

Automatic First-order proof search methods instead of induction

Contribution

Deductive verification method handling well-behaved loops automatically

Incomplete Parallelizable loops; but no abstraction when it works

Robust No syntactic restriction on initialization, step, body

Sound Dependence analysis ensures soundness

Automatic First-order proof search methods instead of induction

Relevant Frequently applicable in real JAVA CARD programs

Contribution

Deductive verification method handling well-behaved loops automatically

Incomplete Parallelizable loops; but no abstraction when it works

Robust No syntactic restriction on initialization, step, body

Sound Dependence analysis ensures soundness

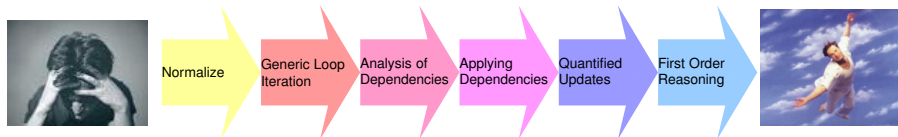
Automatic First-order proof search methods instead of induction

Relevant Frequently applicable in real JAVA CARD programs

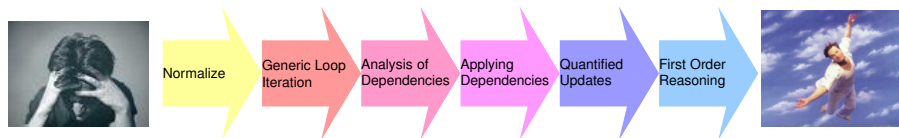
Simplifications made for presentation only

No while-, only for-loops; no abrupt termination (exceptions, break, etc.)

Six Easy Steps to Automatic Verification of Loops



Six Easy Steps to Automatic Verification of Loops



Main idea: transform parallelizable loops into first-order formulas

State Updates

Notation for updates of computation states

$l := r$

- ▶ l location
- ▶ r side effect-free expression
- ▶ r evaluated in old state

State Updates

Notation for updates of computation states

$l := r$

- ▶ l location
- ▶ r side effect-free expression
- ▶ r evaluated in old state

Parallel update (cf. B notation, ASMs)

$\{l_1 := r_1 \parallel \dots \parallel l_n := r_n\}$

- ▶ All updates executed in parallel
- ▶ All right-hand sides evaluated in old state
- ▶ Syntactically last update wins upon conflict on the left

Running Example

Let p be the following loop:

```
for (int i = 1; i < a.length; i++)  
    if (c != 0)  
        a[i] = b[i+1]  
    else  
        a[i] = b[i-1];
```

Running Example

Let p be the following loop:

```
for (int i = 1; i < a.length; i++)
  if (c != 0)
    a[i] = b[i+1]
  else
    a[i] = b[i-1];
```

Proof situation (sequent calculus, dynamic logic)

$$\Gamma \Rightarrow \{c := 1\} \langle p \rangle \phi$$

$\models \langle p \rangle \phi$ iff p terminates normally and in final state ϕ holds

Running Example

Let p be the following loop:

```
for (int i = 1; i < a.length; i++)
  if (c != 0)
    a[i] = b[i+1]
  else
    a[i] = b[i-1];
```

Proof situation (sequent calculus, dynamic logic)

$$\Gamma \Rightarrow \{c := 1\} \langle p \rangle \phi$$

$\models \langle p \rangle \phi$ iff p terminates normally and in final state ϕ holds

Wanted: strongest post condition of p in state $\mathcal{S} = \{c := 1\}$ wrt Γ

Step 1: Normalize Loop

Initialization expression might be complex and have side effects

Step 1: Normalize Loop

Initialization expression might be complex and have side effects

```
{c:=1}
for (int i = 1; i < a.length; i++)
    if (c != 0)
        a[i] = b[i+1]
    else
        a[i] = b[i-1];
```

Step 1: Normalize Loop

Initialization expression might be complex and have side effects

Move initialization up front (use calculus, rename i if necessary)

```
{c:=1}
int i = 1;
for ( ; i < a.length; i++)
    if (c != 0)
        a[i] = b[i+1]
    else
        a[i] = b[i-1];
```

Step 1: Normalize Loop

Initialization expression might be complex and have side effects

Then execute initialization code symbolically

```
{c:=1, i:=1}
```

```
for ( ; i < a.length; i++)  
  if (c != 0)  
    a[i] = b[i+1]  
  else  
    a[i] = b[i-1];
```

Step 2: Symbolic Execution of Generic Loop Iteration

Idea:

Symbolically execute loop body, step, and guard for generic value of i

- ▶ Computes “normalized” effect of one loop iteration
- ▶ Essential for dealing with syntactically unrestricted loops

Step 2: Symbolic Execution of Generic Loop Iteration

Idea:

Symbolically execute loop body, step, and guard for generic value of i

- ▶ Computes “normalized” effect of one loop iteration
- ▶ Essential for dealing with syntactically unrestricted loops

```
S // initial state for generic loop iteration  
if (c != 0) // loop body  
    a[i] = b[i+1]  
else  
    a[i] = b[i-1];  
i++; // step  
boolean _dummy = i < a.length; // guard
```

Step 2:

Symbolic Execution of Generic Loop Iteration

Idea:

Symbolically execute loop body, step, and guard for generic value of i

- ▶ Computes “normalized” effect of one loop iteration
- ▶ Essential for dealing with syntactically unrestricted loops

```
S // initial state for generic loop iteration
if (c != 0) // loop body
    a[i] = b[i+1]
else
    a[i] = b[i-1];
i++; // step
boolean _dummy = i < a.length; // guard
```

In which state S is execution started?

Step 2 (cont'd)

Computing the Initial State for Iteration

```
S // initial state for generic loop iteration
if (c != 0) // loop body
    a[i] = b[i+1]
else
    a[i] = b[i-1];
i++; // step
boolean _dummy = i < a.length; // guard
```

Step 2 (cont'd)

Computing the Initial State for Iteration

```
{c:=1, i:=1}      // state before loop
if (c != 0)       // loop body
  a[i] = b[i+1]
else
  a[i] = b[i-1];
i++;              // step
boolean _dummy = i < a.length; // guard
```

Can't use state before loop (not generic)

Result $\{c := 1, a[1] := b[2], i := 2\}$

Step 2 (cont'd)

Computing the Initial State for Iteration

```
{ } // empty state
if (c != 0) // loop body
    a[i] = b[i+1]
else
    a[i] = b[i-1];
i++; // step
boolean _dummy = i < a.length; // guard
```

Inefficient to use empty state

Result $\{i := i + 1, \backslash \text{if } (! (c=0)) \{a[i] := b[i+1]\} \{a[i] := b[i-1]\}\}$

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

1. Let \mathcal{S} be state before loop; guard executed once $\{c := 1, i := 1\}$

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

1. Let \mathcal{S} be state before loop; guard executed once $\{c := 1, i := 1\}$
2. Symbolically execute an iteration starting in \mathcal{S} , obtain \mathcal{S}'
 $\{c := 1, a[1] := b[2], i := 2\}$

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

1. Let \mathcal{S} be state before loop; guard executed once $\{c := 1, i := 1\}$
2. Symbolically execute an iteration starting in \mathcal{S} , obtain \mathcal{S}'
 $\{c := 1, a[1] := b[2], i := 2\}$
3. Remove from \mathcal{S} all locations modified in \mathcal{S}' $\{c := 1\}$

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

1. Let \mathcal{S} be state before loop; guard executed once $\{c := 1, i := 1\}$
2. Symbolically execute an iteration starting in \mathcal{S} , obtain \mathcal{S}'
 $\{c := 1, a[1] := b[2], i := 2\}$
3. Remove from \mathcal{S} all locations modified in \mathcal{S}' $\{c := 1\}$
4. If \mathcal{S} unchanged, then stop; otherwise, goto 2.

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

1. Let \mathcal{S} be state before loop; guard executed once $\{c := 1, i := 1\}$
2. Symbolically execute an iteration starting in \mathcal{S} , obtain \mathcal{S}'
 $\{c := 1, a[i] := b[i+1], i := i + 1\}$
3. Remove from \mathcal{S} all locations modified in \mathcal{S}' $\{c := 1\}$
4. If \mathcal{S} unchanged, then stop; otherwise, goto 2.

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

1. Let \mathcal{S} be state before loop; guard executed once $\{c := 1, i := 1\}$
2. Symbolically execute an iteration starting in \mathcal{S} , obtain \mathcal{S}'
 $\{c := 1, a[i] := b[i+1], i := i + 1\}$
3. Remove from \mathcal{S} all locations modified in \mathcal{S}' $\{c := 1\}$
4. If \mathcal{S} unchanged, then stop; otherwise, goto 2.

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

1. Let \mathcal{S} be state before loop; guard executed once $\{c := 1, i := 1\}$
2. Symbolically execute an iteration starting in \mathcal{S} , obtain \mathcal{S}'
 $\{c := 1, a[i] := b[i+1], i := i + 1\}$
3. Remove from \mathcal{S} all locations modified in \mathcal{S}' $\{c := 1\}$
4. If \mathcal{S} unchanged, then **stop**; otherwise, goto 2.

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

1. Let \mathcal{S} be state before loop; guard executed once $\{c := 1, i := 1\}$
2. Symbolically execute an iteration starting in \mathcal{S} , obtain \mathcal{S}'
 $\{c := 1, a[i] := b[i+1], i := i + 1\}$
3. Remove from \mathcal{S} all locations modified in \mathcal{S}' $\{c := 1\}$
4. If \mathcal{S} unchanged, then **stop**; otherwise, goto 2.

Terminates: $\mathcal{S}, \mathcal{S}'$ finite (bound by $\#$ of modifiable locations in iteration)

Step 2 (cont'd)

Computing the Initial State for Iteration

Approximation of maximal generic state for loop iteration

Idea: take away from the state before the loop is executed those locations that are modifiable by the loop body in that state

1. Let \mathcal{S} be state before loop; guard executed once $\{c := 1, i := 1\}$
2. Symbolically execute an iteration starting in \mathcal{S} , obtain \mathcal{S}'
 $\{c := 1, a[i] := b[i+1], i := i + 1\}$
3. Remove from \mathcal{S} all locations modified in \mathcal{S}' $\{c := 1\}$
4. If \mathcal{S} unchanged, then **stop**; otherwise, goto 2.

Terminates: $\mathcal{S}, \mathcal{S}'$ finite (bound by $\#$ of modifiable locations in iteration)

Effect of generic loop iteration obtained from **final round**

Step 3: Analyzing Dependencies

Obtained state \mathcal{S} characterizes effect of generic loop iteration over i

Step 3: Analyzing Dependencies

Obtained state \mathcal{S} characterizes effect of generic loop iteration over i

Goal:

Replace **iterative** loop structure by **parallel** assignment for all i in range

Step 3: Analyzing Dependencies

Obtained state \mathcal{S} characterizes effect of generic loop iteration over i

Goal:

Replace **iterative** loop structure by **parallel** assignment for all i in range

- ▶ Need to analyze variable dependencies in \mathcal{S}
- ▶ Problem has been intensely studied in parallelization and vectorization (mainly 1980s)
- ▶ Analysis takes place on state \mathcal{S} (without location i)
Dependence analysis implemented on state updates, not full JAVA
- ▶ Two kinds of dependencies: data input and data output dependence

Step 3 (cont'd): Data Input Dependence

Let \mathcal{S}_j be result of generic loop iteration over i , where i has value j

Step 3 (cont'd): Data Input Dependence

Let \mathcal{S}_j be result of generic loop iteration over i , where i has value j

Definition (Data input dependence)

There is a **data input dependence** between iterations $k \neq l$ in \mathcal{S} iff \mathcal{S}_k writes to a location (appears on LHS of an update) that is read (appears on RHS or guard of an update) in \mathcal{S}_l .

Step 3 (cont'd): Data Input Dependence

Let \mathcal{S}_j be result of generic loop iteration over i , where i has value j

Definition (Data input dependence)

There is a **data input dependence** between iterations $k \neq l$ in \mathcal{S} iff \mathcal{S}_k writes to a location (appears on LHS of an update) that is read (appears on RHS or guard of an update) in \mathcal{S}_l .

Example (Data flow-dependence, $k < l$, backward)

$a[i] = a[i-1]$

Cannot be parallelized!

Step 3 (cont'd): Data Input Dependence

Let \mathcal{S}_j be result of generic loop iteration over i , where i has value j

Definition (Data input dependence)

There is a **data input dependence** between iterations $k \neq l$ in \mathcal{S} iff \mathcal{S}_k writes to a location (appears on LHS of an update) that is read (appears on RHS or guard of an update) in \mathcal{S}_l .

Example (Data flow-dependence, $k < l$, backward)

$a[i] = a[i-1]$

Cannot be parallelized!

Example (Data anti-dependence, $k > l$, forward)

$a[i] = a[i+1]$

Can be parallelized (RHS evaluated in old state)

Step 3 (cont'd): Data Output Dependence

Definition (Data output dependence)

There is an **data output dependence** between iterations $k \neq l$ in S iff S_k writes to a location (appears on LHS of an update) that is overwritten in S_l .

Step 3 (cont'd): Data Output Dependence

Definition (Data output dependence)

There is an **data output dependence** between iterations $k \neq l$ in S iff S_k writes to a location (appears on LHS of an update) that is overwritten in S_l .

Example

$c = a[i]$

Can be parallelized with last-win semantics!

Step 3 (cont'd): Computing Dependencies

Example (for the loop discussed previously)

$\{c := 1, a[i] := b[i+1]\}$

Contains at most data anti-dependence even if $a = b$

Step 3 (cont'd): Computing Dependencies

Example (for the loop discussed previously)

$\{c := 1, a[i] := b[i+1]\}$

Contains at most data anti-dependence even if $a = b$

Example (Non trivial constraint)

$\{\backslash \text{if } !(c=0) \{a[i] := b[i+1]\} \{a[i] := b[i-1]\}\}$

Inequality analysis yields symbolic constraint $!(c=0) \mid !(a=b)$

Step 3 (cont'd): Computing Dependencies

Example (for the loop discussed previously)

$\{c := 1, a[i] := b[i+1]\}$

Contains at most data anti-dependence even if $a = b$

Example (Non trivial constraint)

$\{\backslash \text{if } !(c=0) \{a[i] := b[i+1]\} \{a[i] := b[i-1]\}\}$

Inequality analysis yields symbolic constraint $!(c=0) \mid !(a=b)$

In general, compute symbolic inequality constraint that characterizes whether generic loop iteration \mathcal{S} is parallelizable

Generalization of Banerjee's inequality analysis (symbolic, aliasing)

Step 4: Using Dependencies in a Proof

$$\frac{\Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi}{\vdots}$$

Step 4: Using Dependencies in a Proof

$$\frac{\Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi}{\vdots}$$

Assume dependence analysis yields constraint $\mathcal{C} \neq \text{true}$

\Rightarrow Loop is parallelizable iff \mathcal{C} holds

Step 4: Using Dependencies in a Proof

$$\frac{\Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi, \mathcal{C} \quad \frac{\text{Translate into updates (Step 5)}}{\mathcal{C}, \Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi}}{\Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi} \textit{cut}$$

⋮

Assume dependence analysis yields constraint $\mathcal{C} \neq \text{true}$

\Rightarrow Loop is parallelizable iff \mathcal{C} holds

Step 4: Using Dependencies in a Proof

$$\frac{\Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi, \mathcal{C} \quad \frac{\text{Translate into updates (Step 5)}}{\mathcal{C}, \Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi}}{\Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi} \text{ cut}$$

□
⋮
⋮

Assume dependence analysis yields constraint $\mathcal{C} \neq \text{true}$

\Rightarrow Loop is parallelizable iff \mathcal{C} holds

$\Gamma \Rightarrow \mathcal{C}$ is provable

Step 4: Using Dependencies in a Proof

$$\frac{\frac{\text{If not } \Gamma \Rightarrow \mathcal{C} \text{ invariant/induction}}{\Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi, \mathcal{C}} \quad \frac{\text{Translate into updates (Step 5)}}{\mathcal{C}, \Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi}}{\Gamma \Rightarrow \langle \text{for } \dots ; \rangle \phi} \text{ cut}$$

⋮

Assume dependence analysis yields constraint $\mathcal{C} \neq \text{true}$

\Rightarrow Loop is parallelizable iff \mathcal{C} holds

May use $\neg \mathcal{C}$ in invariant/induction-based proof

Even when the method fails, it still yields useable results!

Step 5: Quantified State Updates

Definition (Quantified State Update)

\mathcal{S} state update with occurrence of a program variable v over well-ordered type T , ϕ formula, then **for v ; ϕ ; \mathcal{S}** is **quantified state update**

Semantics

In model \mathcal{M} , variable assignment β :

Parallel execution of updates $\{\{v := x^{\mathcal{M}}\}\mathcal{S} \mid x \in T, \mathcal{M}, \beta_v^x \models \phi\}$

Clash Resolution

Last update with respect to well-order on T , syntactic order in \mathcal{S} wins

Step 5: Quantified State Updates

Definition (Quantified State Update)

\mathcal{S} state update with occurrence of a program variable v over well-ordered type T , ϕ formula, then **for** v ; ϕ ; \mathcal{S} is **quantified state update**

Semantics

In model \mathcal{M} , variable assignment β :

Parallel execution of updates $\{\{v := x^{\mathcal{M}}\}\mathcal{S} \mid x \in T, \mathcal{M}, \beta_v^x \models \phi\}$

Clash Resolution

Last update with respect to well-order on T , syntactic order in \mathcal{S} wins

Example

Let $i:\text{int}$, then **for** i ; **true**; $\{x := i\}$ is $\{x := \text{Integer.MAX_VALUE}\}$

Step 5 (cont'd)

Translating Loops into Quantified State Updates

```
{c:=1, i:=1} // state before loop execution
for ( ; i < a.length; i++)
  if (c != 0)
    a[i] = b[i+1]
  else
    a[i] = b[i-1];
```

Result of generic loop iteration: $\{c:=1, i:=i+1, a[i]:=b[i+1]\}$
(no dependencies)

Step 5 (cont'd)

Translating Loops into Quantified State Updates

```
{c:=1, i:=1} // state before loop execution
for ( ; i < a.length; i++)
  if (c != 0)
    a[i] = b[i+1]
  else
    a[i] = b[i-1];
```

Result of generic loop iteration: $\{c:=1, i:=i+1, a[i]:=b[i+1]\}$
(no dependencies)

Synthesize quantified update from **initial state**, **guard**, **generic iteration**:

```
for i; (i >= 1 & i < a.length); {c:=1, i:=i+1, a[i]:=b[i+1]}
```

Step 6:

First Order Reasoning with Quantified Updates

No output dependence: quantified update variable replaced by any value

$$\frac{\Rightarrow \{v := t\}\phi \quad \Rightarrow (\{v := t\}\mathcal{S})\psi}{\Rightarrow \{\text{for } v; \phi; \mathcal{S}\}\psi} \text{quantUpdateElim}$$

t any term with type T (+ side condition on admissible location in ψ)

Step 6:

First Order Reasoning with Quantified Updates

No output dependence: quantified update variable replaced by any value

$$\frac{\Rightarrow \{v := t\}\phi \quad \Rightarrow (\{v := t\}\mathcal{S})\psi}{\Rightarrow \{\text{for } v; \phi; \mathcal{S}\}\psi} \text{quantUpdateElim}$$

t any term with type T (+ side condition on admissible location in ψ)

$$\Rightarrow \{\text{for } i; (i \geq 1 \ \& \ i < \text{a.length}); \{c := 1, \text{a}[i] := \text{b}[i+1]\}\}(\text{a}[2] = \text{b}[3])$$

Step 6:

First Order Reasoning with Quantified Updates

No output dependence: quantified update variable replaced by any value

$$\frac{\Rightarrow \{v := t\}\phi \qquad \Rightarrow (\{v := t\}\mathcal{S})\psi}{\Rightarrow \{\text{for } v; \phi; \mathcal{S}\}\psi} \text{ quantUpdateElim}$$

t any term with type T (+ side condition on admissible location in ψ)

$$\frac{\Rightarrow \{i := 2\}(i \geq 1 \ \& \ i < a.length) \qquad \Rightarrow \{i := 2\}\{c := 1, a[i] := b[i+1]\}(a[2]=b[3])}{\Rightarrow \{\text{for } i; (i \geq 1 \ \& \ i < a.length); \{c := 1, a[i] := b[i+1]\}\}(a[2]=b[3])}$$

Step 6:

First Order Reasoning with Quantified Updates

No output dependence: quantified update variable replaced by any value

$$\frac{\Rightarrow \{v := t\}\phi \qquad \Rightarrow (\{v := t\}\mathcal{S})\psi}{\Rightarrow \{\text{for } v; \phi; \mathcal{S}\}\psi} \text{quantUpdateElim}$$

t any term with type T (+ side condition on admissible location in ψ)

$$\frac{\begin{array}{l} \Rightarrow (2 \geq 1 \ \& \ 2 < \text{a.length}) \qquad \Rightarrow \{c := 1, \text{a}[2] := \text{b}[3]\}(\text{a}[2] = \text{b}[3]) \\ \Rightarrow \{i := 2\}(i \geq 1 \ \& \ i < \text{a.length}) \qquad \Rightarrow \{i := 2\}\{c := 1, \text{a}[i] := \text{b}[i+1]\}(\text{a}[2] = \text{b}[3]) \end{array}}{\Rightarrow \{\text{for } i; (i \geq 1 \ \& \ i < \text{a.length}); \{c := 1, \text{a}[i] := \text{b}[i+1]\}\}(\text{a}[2] = \text{b}[3])}$$

Step 6:

First Order Reasoning with Quantified Updates

No output dependence: quantified update variable replaced by any value

$$\frac{\Rightarrow \{v := t\}\phi \qquad \Rightarrow (\{v := t\}\mathcal{S})\psi}{\Rightarrow \{\text{for } v; \phi; \mathcal{S}\}\psi} \text{quantUpdateElim}$$

t any term with type T (+ side condition on admissible location in ψ)

$$\frac{\begin{array}{l} \Rightarrow (2 \geq 1 \ \& \ 2 < \text{a.length}) \\ \Rightarrow \{i := 2\}(i \geq 1 \ \& \ i < \text{a.length}) \end{array} \qquad \Rightarrow \{c := 1, \text{a}[2] := \text{b}[3]\}(\text{a}[2] = \text{b}[3])}{\Rightarrow \{\text{for } i; (i \geq 1 \ \& \ i < \text{a.length}); \{c := 1, \text{a}[i] := \text{b}[i+1]\}\}(\text{a}[2] = \text{b}[3])}$$

First-order reasoning patterns: simplification, quantifier elimination

Evaluation

- ▶ Prototypic implementation in Haskell, reimplementing under way
- ▶ Three real JAVA CARD programs

Evaluation

- ▶ Prototypic implementation in Haskell, reimplementing under way
- ▶ Three real JAVA CARD programs

	DeMoney	SafeApplet	IButtonAPI	Total	Remarks
LoC	1633	514	102	2249	
Size (kB)	182	22	3	207	
# loops	10	6	1	17	
handled	4	0	1	5	future work
with ext.	3	1	0	4	
remaining	3	5	0	8	

Evaluation

- ▶ Prototypic implementation in Haskell, reimplementing under way
- ▶ Three real JAVA CARD programs

	DeMoney	SafeApplet	IButtonAPI	Total	Remarks
LoC	1633	514	102	2249	
Size (kB)	182	22	3	207	
# loops	10	6	1	17	
handled	4	0	1	5	future work
with ext.	3	1	0	4	
remaining	3	5	0	8	

Could be further improved by rewriting some loops (SafeApplet)

Develop design guidelines for easy verifiability of loops

Summary

- ▶ **Parallelize loops** whose bodies can be executed independently
- ▶ Effect of one loop iteration computed by **symbolic execution**
- ▶ Dependencies derived automatically by **symbolic analysis**
- ▶ Parallelizable loops represented with **quantified state updates**
- ▶ Symbolic execution/updates \Rightarrow **unrestricted loop syntax**
- ▶ Quantified updates allow automatic **first order reasoning**
- ▶ Yields useful result (constraint) even when not applicable
- ▶ Case studies suggest that approach is **practically relevant**

Paper published at LPAR 2006, Cambodia, LNCS