Mostly Automated Formal Verification of Loop Dependencies

Thomas Grégoire,
Adam Chlipala

August 21, 2016
A bit of motivation

- Consider a PDE in dimension $\leq 3$, e.g., the 1D Heat equation:

$$\frac{\partial u}{\partial t} - \alpha \frac{\partial^2 u}{\partial x^2} = 0$$

- One way of solving it: finite-difference schemes.

$$\frac{\partial u}{\partial t} \approx \frac{u(t + \Delta t, x, y) - u(t, x, y)}{\Delta t}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t, x + \Delta x, y) - 2u(t, x, y) + u(t, x - \Delta x, y)}{\Delta x^2}$$
A bit of motivation (2)

- Yields an equation of the form:

\[ u[t + 1, x] = F(u[t, x], u[t, x + 1], u[t, x - 1]) \]

- Graphically:
Implementing stencil code (1)

- Naive algorithm: left-to-right, bottom-up traversal.
- We can do better: \textit{cache-oblivious} implementation!
Implementing stencil code (2)

- Stencil code calls for **parallel** implementation.
- Typical implementation: each node computes values until it needs more data. Synchronize, then repeat.
- Limiting factor = **number of synchronizations**. Goal is to minimize it.
Implementing stencil code in parallel
Goals of this talk

- Define stencils and stencil algorithms within Coq.
- Formalize the notion of “algorithm A satisfies the dependencies of stencil S.”
- Investigate automation.

Note: larger scope. Applies to dynamic programming, some image filters, Gauss-Seidel iterations, etc.
Representing stencils

Parameters $T$, $I$, $J : \mathbb{Z}$.

Module $\text{Jacobi2D} <:(\text{PROBLEM Z3}).$

Local Open Scope $\text{aexpr}$.

Definition $\text{space} := [0, T] \times [0, I] \times [0, J]$.
Definition $\text{target} := [0, T] \times [0, I] \times [0, J]$.
Definition $\text{dep } c :=$

\[
\begin{align*}
\text{match } c \text{ with} & \\
& | (t, i, j) \Rightarrow [(t-1, i, j); (t-1, i-1, j); \\
& (t-1, i+1, j); (t-1, i, j-1); (t-1, i, j+1)]
\end{align*}
\]

end.

End $\text{Jacobi2D}$.
Programs and their correctness

- Programs are Hoare-logic style, with a “flag” command.
- Intuitively (for now): flag = compute this cell.
- Trivial program for the Jacobi 2D stencil:

\[
\begin{align*}
\text{for } t=0 \text{ to } T \text{ do} \\
&\text{for } i=0 \text{ to } I \text{ do} \\
&\quad \text{for } j=0 \text{ to } J \text{ do} \\
&\quad \quad \text{flag } u_t[i, j]
\end{align*}
\]
Correctness of stencil code

- **Completeness**: all cells we need to compute are eventually computed.
- **Correctness**: no dependency is violated. Checked through a translation process:

```plaintext
for t=0 to T do
  for i=0 to I do
    for j=0 to J do
      flag u_t[i,j] ⇒
      assert (t-1, i, j);
      assert (t-1, i+1, j);
      assert (t-1, i-1, j);
      assert (t-1, i, j+1);
      assert (t-1, i, j-1);
      flag (t, i, j)
```
Syntax and semantics

- We keep a boolean for each cell, indicating whether it has already been computed.
- \textbf{flag} \( c \) marks cell \( c \) as computed.
- \textbf{assert} \( c \) checks that \( c \) has been computed and fails if not.
- Syntax:
  
  \[
  \begin{align*}
  p & ::= \text{nop} \mid p; p \mid \text{flag} \; c \mid \text{assert} \; c \mid \text{if} \; b \; \text{then} \; p \; \text{else} \; p \mid \text{for} \; v = e \; \text{to} \; e \; \text{do} \; p \\
  e & ::= k \mid v \mid e + e \mid e - e \mid e \times e \mid e / e \mid e \mod e \\
  b & ::= \epsilon \mid \text{not} \; b \mid b \; \text{or} \; b \mid b \; \text{and} \; b \mid e = e \mid e \neq e \mid e \leq e \mid e \geq e \mid e < e \mid e > e \\
  k & \in \mathbb{Z}, \; \epsilon \in \{\top, \bot\}
  \end{align*}
  \]
Semantics

- Given by a judgment $\rho \vdash (C_1, p) \Downarrow C_2$.
- “If I execute $p$ in environment $\rho$ knowing the cells in $C_1$, then it terminates without any assertion failing, and I will know the cells in $C_2$.”
- Assert and flag:

  $\begin{align*}
  \text{Assert: } & \quad [[c]] \rho \in C \lor [[c]] \rho \notin \text{space} \\
  \rho \vdash (C, \text{assert } c) \Downarrow C \\
  \text{Flag: } & \quad \rho \vdash (C, \text{flag } c) \Downarrow C \cup \{[[c]] \rho\}
  \end{align*}$

- Remaining rules are inherited from Hoare logic.
Verification Conditions

- As usual, we can generate verification conditions recursively on every program.
- More precisely, we prove a statement of the form:

“Let \( p \) be a program, \( \rho \) an environment, and \( C \) a set of cells. If \( \text{VC}_{\rho, C}(p) \) holds, then \( \rho \vdash (C, p) \Downarrow (C \cup \text{Shape}_\rho(p)) \)”.

- Intuitively, \( \text{Shape}_\rho(p) \) is the set of cells computed by \( p \) if it does not fail.
Intuitively, \( \text{Shape}_\rho(p) \) is the set of cells computed by \( p \) if it does not fail.

\[
\begin{align*}
\text{Shape}_\rho(\text{nop}) & := \emptyset, \quad \text{Shape}_\rho(\text{flag } c) := \{ [c]_\rho \} \\
\text{Shape}_\rho(\text{if } b \text{ then } p_1 \text{ else } p_2) & := \begin{cases} 
\text{Shape}_\rho(p_1) \text{ if } [b]_\rho = \top \\
\text{Shape}_\rho(p_2) \text{ otherwise}
\end{cases} \\
\text{Shape}_\rho(p_1; p_2) & := \text{Shape}_\rho(p_1) \cup \text{Shape}_\rho(p_2) \\
\text{Shape}_\rho(\text{assert } c) & := \emptyset \\
\text{Shape}_\rho(\text{for } x = a \text{ to } b \text{ do } p) & := \bigcup_{k \in [A, B]} \text{Shape}_\rho[x \leftarrow k](p), \quad A = [a]_\rho, B = [b]_\rho
\end{align*}
\]
... and the verification conditions

\[
\begin{align*}
VC_{\rho,C}(\text{nop}) & := \top, \quad VC_{\rho,C}(\text{flag } c) := \top \\
VC_{\rho,C}(\text{if } b \text{ then } p_1 \text{ else } p_2) & := \begin{cases} 
VC_{\rho,C}(p_1) & \text{if } \llbracket b \rrbracket_\rho = \top \\
VC_{\rho,C}(p_2) & \text{otherwise}
\end{cases} \\
VC_{\rho,C}(p_1; p_2) & := VC_{\rho,C}(p_1) \land VC_{\rho,C \cup \text{Shape}_\rho(p_1)}(p_2) \\
VC_{\rho,C}(\text{assert } c) & := \llbracket c \rrbracket_\rho \in C \lor \llbracket c \rrbracket_\rho \not\in \text{space} \\
VC_{\rho,C}(\text{for } x = a \text{ to } b \text{ do } p) & := \forall A \leq i \leq B. \ VC_{\rho[x \leftarrow i],D}(p)
\end{align*}
\]

\[A = \llbracket a \rrbracket_\rho, \quad B = \llbracket b \rrbracket_\rho, \quad D = C \cup \text{Shape}_\rho(\text{for } x = a \text{ to } i - 1 \text{ do } p)\]
An example: optimal three-point stencil

\[
\text{Definition Walk1} : \{ Tp : \text{trapezoid} \mid WF Tp \} \rightarrow \text{prog.}
\]

\[
\text{refine (Fix Vol\_order\_wf (fun _ ⇒ prog) (fun Tp self ⇒ _)).}
\]

\[
\text{destruct Tp as [[[[ t0 t1] x0] v0] x1] v1].}
\]

\[
\text{refine (let h := t1 − t0 in}
\]

\[
\text{if h =? 1 then}
\]

\[
\text{For "x" From x0 To (x1 − 1) Do}
\]

\[
\text{Fire (t0 : aexpr, "x" : aexpr)}
\]

\[
\text{else}
\]

\[
\text{if (h * 4) <? ((x1 − x0) * 2 + (v1 − v0) * h) then}
\]

\[
\text{let xm := ((x0 + x1) * 2 + (v0 + v1 + 2) * h) / 4 in}
\]

\[
\text{Call self (t0, t1, x0, v0, xm, −1);}\]

\[
\text{Call self (t0, t1, xm, −1, x1, v1)}\]

\[
\text{else}
\]

\[
\text{let s := h / 2 in}
\]

\[
\text{Call self (t0, t0 + s, x0, v0, x1, v1);}\]

\[
\text{Call self (t0 + s, t1, x0 + v0 * s, v0, x1 + v1 * s, v1))%prog;}
\]

\[
\text{clear self; abstract (substs; prove_Vol || prove_WF).}
\]

\[
\text{Defined.}
\]
A word about automation

- We added automation to clean up the result. Leaves us with goals of the form:

\[ c \in K, \]

where \( c \) is a cell and \( K \) a set of cells.

- Stencils are usually defined on \( \mathbb{Z}^n \). Resulting goals/hypotheses are systems of non-linear integer equations.

- \textit{nia} can solve them, but slow to fail, hence tough for branching.

- Still makes the user experience way nicer.
Back to distributed stencil code
Syntax for distributed programs

<table>
<thead>
<tr>
<th>Computation step</th>
<th>Communication step</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if</strong> $T=0$ <strong>then</strong></td>
<td><strong>if</strong> $T=0$ <strong>then</strong></td>
</tr>
<tr>
<td><strong>for</strong> $t=0$ <strong>to</strong> 3 <strong>do</strong></td>
<td><strong>if</strong> to = id − 1 <strong>then</strong></td>
</tr>
<tr>
<td><strong>for</strong> $i = t$ <strong>to</strong> 7 − $t$ <strong>do</strong></td>
<td><strong>for</strong> $t=0$ <strong>to</strong> 3 <strong>do</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>fire</strong> ($8 \times id + i, t$)</td>
<td><strong>fire</strong> ($8 \times id + t, t$)</td>
</tr>
<tr>
<td><strong>else</strong> */ $T=1$ */</td>
<td><strong>else if</strong> to = id + 1 <strong>then</strong></td>
</tr>
<tr>
<td><strong>for</strong> $t=1$ <strong>to</strong> 3 <strong>do</strong></td>
<td><strong>for</strong> $t=0$ <strong>to</strong> 3 <strong>do</strong></td>
</tr>
<tr>
<td><strong>for</strong> $i = −t$ <strong>to</strong> $t − 1$ <strong>do</strong></td>
<td><strong>fire</strong> ($8 \times id + 4 + t, 3 − t$)</td>
</tr>
<tr>
<td><strong>fire</strong> ($8 \times id + i, t$)</td>
<td></td>
</tr>
<tr>
<td><strong>for</strong> $i = −t$ <strong>to</strong> $t − 1$ <strong>do</strong></td>
<td></td>
</tr>
<tr>
<td><strong>fire</strong> ($8 \times id + 8 + i, t$)</td>
<td></td>
</tr>
</tbody>
</table>
Semantics

- Two semantics for fire $c$: check that $c$ is known (communication), or check that its dependencies are satisfied and flag it (computation).

- An execution trace is a triple $(\text{beforeComp}, \text{afterComp}, \text{sends})$ with

  \[
  \text{beforeComp, afterComp} : \text{time} \times \text{thread} \rightarrow \text{set cell},
  \]

  \[
  \text{sends} : \text{time} \times \text{thread} \times \text{thread} \rightarrow \text{set cell}.
  \]
Some examples

\[
\text{beforeComp}(T = 0, i = 0) = \emptyset, \\
\text{afterComp}(T = 0, i = 0) = \text{hatched}
\]

\[
\text{sends}(T = 0, i = 0, j = 1) = \text{hatched}, \\
\text{sends}(T = 0, i = 0, j = 2) = \emptyset,
\]

\[
\text{beforeComp}(T = 1, i = 0) = \text{hatched & gray}, \\
\text{afterComp}(T = 1, i = 0) = \text{hatched},
\]
Correctness (1)

A distributed stencil algorithm is correct if there exists a trace satisfying:

- Computation steps happen as in the sequential case:
  \[ \rho_{t,i} \vdash (before\text{Comp}(t, i), p_{\text{comp}}) \Downarrow after\text{Comp}(t, i). \]

- The threads start from no knowledge:
  \[ before\text{Comp}(t = 0) = \emptyset. \]
Correctness (2)

- The computation program describes the pattern sent to other threads:
  \[ \rho_{t,i,j} \vdash (\emptyset, \rho_{\text{comm}}) \downarrow \text{sends}(t, i, j). \]

- A thread cannot send a value it does not know:
  \[ \text{sends}(t, i, j) \subseteq \text{afterComp}(t, i). \]

- What a thread knows at time \( t + 1 \), comes from its knowledge at time \( t \) or was just received:
  \[ \text{beforeComp}(t + 1, i) = \text{afterComp}(t, i) \cup \bigcup_{j} \text{sends}(t, j, i). \]
Verifying distributed stencil code

- We derived the “trace that the program would follow if it did not fail.”
- We also designed a VC generator.
- Theorem similar to the one for sequential code.
- With automation, optimized distributed three-point stencil ≈ 160LoC.
Summary

- Definition of stencils, as well as sequential and distributed stencil algorithms.
- A simple solution that leads to the proof of an optimal algorithm.
- In this “synchronous” framework, verifying distributed algorithms boils down to verifying some sequential algorithms.
- Some more work needed in terms of automation.
- Interesting direction: synthesis!
Thank you for your attention!

<table>
<thead>
<tr>
<th>Type</th>
<th>Lines of Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat Equation, 2D</td>
<td>Naive</td>
</tr>
<tr>
<td>American Put Stock Options</td>
<td>Naive</td>
</tr>
<tr>
<td>American Put Stock Options</td>
<td>Optimized</td>
</tr>
<tr>
<td>Distributed American Put Stock Options</td>
<td>Naive</td>
</tr>
<tr>
<td>Distributed American Put Stock Options</td>
<td>Optimized</td>
</tr>
<tr>
<td>Pairwise Sequence Alignment</td>
<td>Dynamic programming</td>
</tr>
<tr>
<td>Distributed Three-Point Stencil</td>
<td>Naive</td>
</tr>
<tr>
<td>Distributed Three-Point Stencil</td>
<td>Optimized</td>
</tr>
<tr>
<td>Universal Three-Point Stencil Algorithm</td>
<td>Optimal</td>
</tr>
</tbody>
</table>

https://github.com/mit-plv/stencils