Mostly Automated Formal Verification of Loop Dependencies

ENS DE LYON

CSAIL

Thomas Grégoire,
Adam Chlipala

August 21, 2016

A bit of motivation

- Consider a PDE in dimension ≤ 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.

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

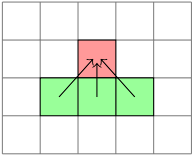
$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t, x + \Delta x, y) - 2u(t, x, y) + u(t, x - \Delta x, y)}{\Delta x^2}$$

A bit of motivation (2)

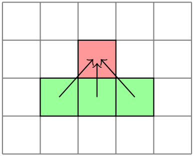
- Yields an equation of the form:

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

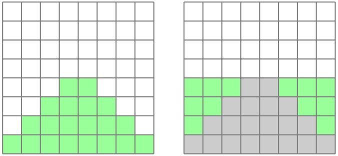
- Graphically:



Implementing stencil code (1)



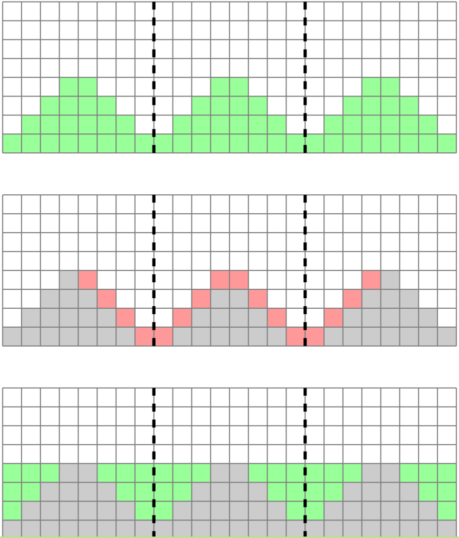
- Naive algorithm: left-to-right, bottom-up traversal.
- We can do better: **cache-oblivious** implementation!



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



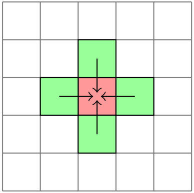
Goals of this talk

- 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

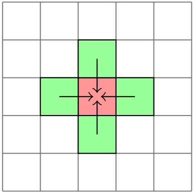
```
Parameters T I J : Z.

Module Jacobi2D <: (PROBLEM Z3).
  Local Open Scope aexpr.

  Definition space := [[0, T]]×[[0, I]]×[[0, J]].
  Definition target := [[0, T]]×[[0, I]]×[[0, J]].
  Definition dep c :=
    match c with
      | (t,i,j) ⇒ [(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.
End 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:

```
for t=0 to T do
  for i=0 to I do
    for j=0 to J do
      flag u_t[i,j]
```



Correctness of stencil code

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

```
for t=0 to T do
  for i=0 to I do
    for j=0 to J do
      flag u_t[i,j]
```

$\Rightarrow$

```
for t=0 to T do
  for i=0 to I do
    for j=0 to J do
      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.
- **flag** c marks cell c as computed.
- **assert** c checks that c has been computed and fails if not.
- Syntax:

$$
\begin{array}{lcl}
p & ::= & \textbf{nop} \mid p; p \mid \textbf{flag}\ c \mid \textbf{assert}\ c \mid \textbf{if}\ b\ \textbf{then}\ p\ \textbf{else}\ p \mid \textbf{for}\ v = e\ \textbf{to}\ e\ \textbf{do}\ p \\
e & ::= & k \mid v \mid e + e \mid e - e \mid e \times e \mid e/e \mid e \bmod e \\
b & ::= & \epsilon \mid \textbf{not}\ b \mid b\ \textbf{or}\ b \mid b\ \textbf{and}\ b \mid e = e \mid e \neq e \mid e \leq e \mid e \geq e \mid e < e \mid e > e
\end{array}
$$

$k \in \mathbb{Z}$, $\epsilon \in \{\top, \bot\}$

Semantics

- Given by a judgment $\rho \vdash (C_1, p) \Downarrow C_2$.
- "If I execute p in environment ρ 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:

$$\text{Assert:}\ \frac{[\![c]\!]_\rho \in C \vee [\![c]\!]_\rho \notin \texttt{space}}{\rho \vdash (C, \textbf{assert } c) \Downarrow C} \qquad \text{Flag:}\ \frac{}{\rho \vdash (C, \textbf{flag } c) \Downarrow C \cup \{[\![c]\!]_\rho\}}$$

- 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, ρ an environment, and C a set of cells. If $\mathsf{VC}_{\rho,C}(p)$ holds, then $\rho \vdash (C, p) \Downarrow (C \cup \mathsf{Shape}_\rho(p))$".

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

$\mathsf{Shape}_\rho \ldots$

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

$$
\begin{aligned}
\mathsf{Shape}_\rho(\mathbf{nop}) &:= \emptyset, \quad \mathsf{Shape}_\rho(\mathbf{flag}\ c) := \{[\![c]\!]_\rho\} \\
\mathsf{Shape}_\rho(\mathbf{if}\ b\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2) &:= \begin{cases} \mathsf{Shape}_\rho(p_1) & \text{if } [\![b]\!]_\rho = \top \\ \mathsf{Shape}_\rho(p_2) & \text{otherwise} \end{cases} \\
\mathsf{Shape}_\rho(p_1; p_2) &:= \mathsf{Shape}_\rho(p_1) \cup \mathsf{Shape}_\rho(p_2) \\
\mathsf{Shape}_\rho(\mathbf{assert}\ c) &:= \emptyset \\
\mathsf{Shape}_\rho(\mathbf{for}\ x = a\ \mathbf{to}\ b\ \mathbf{do}\ p) &:= \bigcup_{k \in [\![A,B]\!]} \mathsf{Shape}_{\rho[x \leftarrow k]}(p), \quad A = [\![a]\!]_\rho, B = [\![b]\!]_\rho
\end{aligned}
$$

… and the verification conditions

$$\mathrm{VC}_{\rho,C}(\textbf{nop}) := \top, \quad \mathrm{VC}_{\rho,C}(\textbf{flag}\ c) := \top$$

$$\mathrm{VC}_{\rho,C}(\textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2) := \begin{cases} \mathrm{VC}_{\rho,C}(p_1) & \text{if } [\![b]\!]_\rho = \top \\ \mathrm{VC}_{\rho,C}(p_2) & \text{otherwise} \end{cases}$$

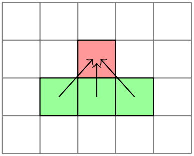
$$\mathrm{VC}_{\rho,C}(p_1; p_2) := \mathrm{VC}_{\rho,C}(p_1) \wedge \mathrm{VC}_{\rho,C\cup\mathrm{Shape}_\rho(p_1)}(p_2)$$

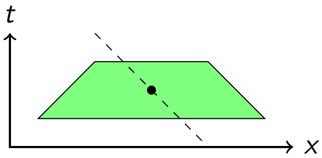
$$\mathrm{VC}_{\rho,C}(\textbf{assert}\ c) := [\![c]\!]_\rho \in C \vee [\![c]\!]_\rho \notin \mathtt{space}$$

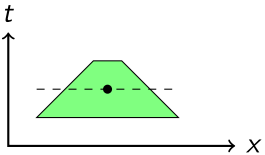
$$\mathrm{VC}_{\rho,C}(\textbf{for}\ x = a\ \textbf{to}\ b\ \textbf{do}\ p) := \forall A \le i \le B.\ \mathrm{VC}_{\rho[x\leftarrow i],D}(p)$$

$$A = [\![a]\!]_\rho,\ B = [\![b]\!]_\rho,\ D = C \cup \mathrm{Shape}_\rho(\textbf{for}\ x = a\ \textbf{to}\ i-1\ \textbf{do}\ p)$$

An example: optimal three-point stencil








```
Definition Walk1 : { Tp : trapezoid | WF Tp } -> prog.
  refine (Fix Vol_order_wf (fun _ ⇒
prog) (fun Tp self ⇒ _)).

  destruct Tp as [[[[[ t0 t1] x0] v0] x1] v1] ].

  refine (let h := t1 − t0 in
    if h =? 1 then
      For "x" From x0 To (x1 − 1) Do
        Fire (t0 : aexpr, "x" : aexpr)
    else
      if (h ∗ 4) <? ((x1 − x0) ∗ 2 + (v1 − v0) ∗ h) then
        let xm := ((x0 + x1) ∗ 2 + (v0 + v1 + 2) ∗ h) / 4 in
        Call self (t0, t1, x0, v0, xm, −1);;
        Call self (t0, t1, xm, −1, x1, v1)
      else
        let s := h / 2 in
        Call self (t0, t0 + s, x0, v0, x1, v1);;
        Call self (t0 + s, t1, x0 + v0 ∗ s, v0, x1 + v1 ∗ s, v1))%prog;
    clear self; abstract (substs; prove_Vol || prove_WF).
Defined.
```

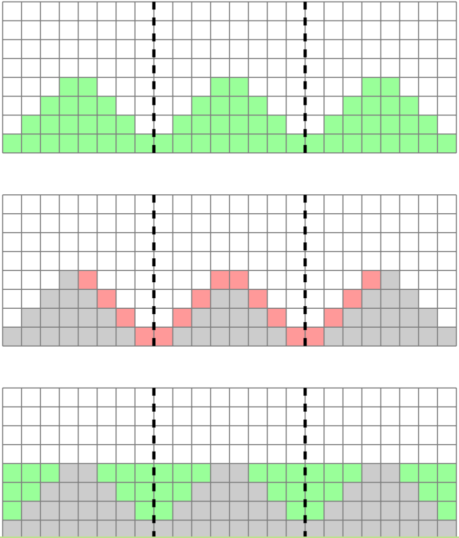
A word about automation

- 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**.
- *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

Computation step	Communication step

if T=0 **then**
  for t=0 **to** 3 **do**
    for $i = t$ **to** $7 - t$ **do**
      fire $(8 \times \mathrm{id} + i, t)$
else *(* T=1 *)*
  for t=1 **to** 3 **do**
    for $i = -t$ **to** $t - 1$ **do**
      fire $(8 \times \mathrm{id} + i, t)$
    for $i = -t$ **to** $t - 1$ **do**
      fire $(8 \times \mathrm{id} + 8 + i, t)$
	if T=0 **then**
	  if $\mathrm{to} = \mathrm{id} - 1$ **then**
	    for t=0 **to** 3 **do**
	      fire $(8 \times \mathrm{id} + t, t)$
	  else if $\mathrm{to} = \mathrm{id} + 1$ **then**
	    for t=0 **to** 3 **do**
	      fire $(8 \times \mathrm{id} + 4 + t, 3 - t)$

Semantics

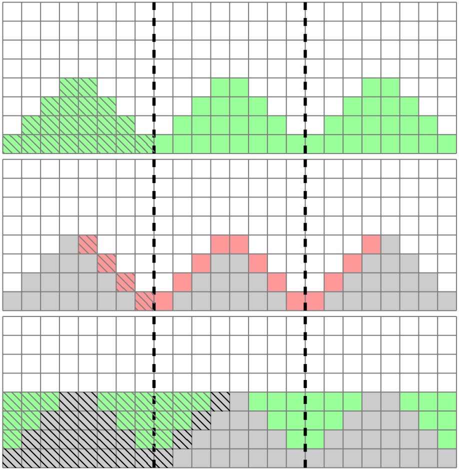
- Two semantics for **fire** c: check that c is known (**communication**), or check that its dependencies are satisfied and flag it (**computation**).

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

$$\mathtt{beforeComp}, \mathtt{afterComp} : \mathtt{time} \times \mathtt{thread} \to \mathtt{set\ cell},$$

$$\mathtt{sends} : \mathtt{time} \times \mathtt{thread} \times \mathtt{thread} \to \mathtt{set\ cell}.$$

Some examples



$\texttt{beforeComp}(T = 0, i = 0) = \emptyset$,

$\texttt{afterComp}(T = 0, i = 0) = \text{hatched}$

$\texttt{sends}(T = 0, i = 0, j = 1) = \text{hatched}$,

$\texttt{sends}(T = 0, i = 0, j = 2) = \emptyset$,

$\texttt{beforeComp}(T = 1, i = 0) = \text{hatched \& gray}$,

$\texttt{afterComp}(T = 1, i = 0) = \text{hatched}$,

Correctness (1)

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

- Computation steps happen as in the sequential case:

$$\rho_{t,i} \vdash (\texttt{beforeComp}(t,i), p_{\mathsf{comp}}) \Downarrow \texttt{afterComp}(t,i).$$

- The threads start from no knowledge:

$$\texttt{beforeComp}(t=0) = \emptyset.$$

Correctness (2)

- The computation program describes the pattern sent to other threads:

$$\rho_{t,i,j} \vdash (\emptyset, p_{\text{comm}}) \Downarrow \texttt{sends}(t, i, j).$$

- A thread cannot send a value it does not know:

$$\texttt{sends}(t, i, j) \subseteq \texttt{afterComp}(t, i).$$

- What a thread knows at time $t + 1$, comes from its knowledge at time t or was just received:

$$\texttt{beforeComp}(t + 1, i) = \texttt{afterComp}(t, i) \cup \bigcup_{j} \texttt{sends}(t, j, i).$$

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 $\approx$ 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!

	Type	Lines of Proof
Heat Equation, 2D	Naive	30
American Put Stock Options	Naive	25
American Put Stock Options	Optimized	25
Distributed American Put Stock Options	Naive	65
Distributed American Put Stock Options	Optimized	150
Pairwise Sequence Alignment	Dynamic programming	20
Distributed Three-Point Stencil	Naive	60
Distributed Three-Point Stencil	Optimized	160
Universal Three-Point Stencil Algorithm	Optimal	300

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