Formal Proof for C-Like Programs

Formal Proof for C-Like Programs

Adam Chlipala

Associate Professor without Tenure

Computer Science and Artificial Intelligence Laboratory (CSAIL)

Massachusetts Institute of Technology

Trade-Offs in Choices of Programming Languages

In implementing some critical-infrastructure software, which programming language to use?

High-level language? (e.g., Java, Python, …)
- Many convenient abstractions to reduce programmer effort
- Generally lower performance, which can make a difference for popular systems

Low-level language? (e.g., C)
- Highest performance, with comprehensive control over low-level behavior
- Less convenient abstractions for programmers
- *Small programming mistakes can create serious security vulnerabilities!*

Example: **BIND** DNS server used to suffer from several new **buffer-overflow** attacks each year.
Has since been refactored so that vulnerabilities manifest as **assertion failures** instead of **segmentation faults**, surprising overwrites of live memory state, etc.
Could we hope to rule out even these assertion failures, in a principled way?

Formal Program Proofs (Hoare Logic)

Today, it is increasingly feasible to **prove mathematically** that programs avoid certain defects.
One venerable approach, applicable to C-like languages, is called **Hoare logic**.
(Named after *Tony Hoare*, a Turing Award winner who also invented quicksort, etc.)

In this CyberX topic: intro to the concepts behind Hoare-logic program proofs.
We will see how to do proofs that guarantee:
Invulnerability to buffer overflows and other low-level abstraction violations
Program-specific semantic correctness properties (e.g., no run-time assertion violations)

Segment 1: proving basic programs using integer variables
Segment 2: adding arrays
Segment 3: adding pointers
Segment 4: adding linked data structures
Segment 5: proving program-specific semantic properties
Coda: some pointers to further reading and open-source tools

Prerequisites

Some freshman-level material for college computer-science majors:

Basic programming in an imperative language

Including basics of working with C-style pointers

Basic discrete math and logic

Notations from first-order logic. Here's a quick refresher:

P, **Q**: logical variables standing for some *propositions*, each either true or false
$\neg$**P**: "not": **P** is false.
P $\wedge$ **Q**: "and": both **P** and **Q** are true.
P $\vee$ **Q**: "or": at least one of **P** and **Q** is true.
P $\rightarrow$ **Q**: "implies": if **P**, then **Q**.
$\forall x$. **P**(x): "for all": **P**(x) holds for any possible value of x to be substituted.
$\exists x$. **P**(x): "there exists": **P**(x) holds for *some* value of x.

To Keep in Mind:

The examples here will involve manual derivations and proofs about programs.
That sort of activity doesn't scale well to large, realistic programs!
Luckily, today it is well understood how to automate these derivations with software.
See the very end for pointers to some of those tools (which will make more sense after some practice doing manual proofs!).

Segment 1: proving basic programs using integer variables

A first example: swapping two variables

// Assuming: `int x, y`

$\{x = x_0 \wedge y = y_0\}$ ← Precondition

```
int tmp = x;
```

$\{x = x_0 \wedge y = y_0 \wedge tmp = x_0\}$ ← Intermediate invariants

```
x = y;
```

$\{x = y_0 \wedge y = y_0 \wedge tmp = x_0\}$ ← Intermediate invariants

```
y = tmp;
```

$\{x = y_0 \wedge y = x_0\}$ ← Postcondition

The game is:
Each {…} is an **assertion** that should hold each time we reach it.

Our basic units of proof look like:

{P} `stmt` **{Q}**

That means:

Assume **P** in starting state.
Prove that **Q** holds
in ending state,
with no memory errors
while running `stmt`.

Proving a single assignment

$\{x = x_0 \wedge y = y_0 \wedge \texttt{tmp} = x_0\}$

$\quad \texttt{x = y;}$

$\{x = y_0 \wedge y = y_0 \wedge \texttt{tmp} = x_0\}$

A simple assignment rule applies for assertions in this simple form:

A sequence of "and"s connecting formulas $\mathbf{P}_i$ where, for each $\mathbf{P}_i$, either:

$\mathbf{P}_i$ looks like "x = e", for some program variable x, and where e mentions no program variables; or

$\mathbf{P}_i$ doesn't mention any program variables. (An example of a non-program variable above is x_0.)

Then we have a rule for "x = e", starting from precondition $\mathbf{P}_0 \wedge \ldots \wedge \mathbf{P}_n$.

Condition: every program variable in e has an associated equation in some $\mathbf{P}_i$.

Postcondition is: precondition with the equation for x modified appropriately.

Strengthening preconditions, weakening postconditions

When we know:	We may conclude:	If:
{P}	{P'}	P' → P
`stmt;`	`stmt;`	and
{Q}	{Q'}	Q → Q'

Allows us to prove this:

$\{x = y_0 \land y = y_0 \land \texttt{tmp} = x_0\}$

```
    y = tmp;
```

$\{x = y_0 \land y = x_0 \land \texttt{tmp} = x_0\}$

And conclude this:

$\{x = y_0 \land y = y_0 \land \texttt{tmp} = x_0\}$

```
    y = tmp;
```

$\{x = y_0 \land y = x_0\}$

Handling conditionals

An example involving absolute value:

$\{x = x_0 \wedge y = y_0\}$

```
    if (x < y) {
        r = y - x;
    } else {
        r = x - y;
    }
```

$\{r = |x_0 - y_0|\}$

General "`if`" rule:

To establish:

$\{\mathbf{P}\}$ `if (e) { s1 } else { s2 }` $\{\mathbf{Q}\}$

Must show:

$\{\mathbf{P} \wedge \mathrm{e}\}\ \mathrm{s1}\ \{\mathbf{Q}\}$

$\{\mathbf{P} \wedge \neg \mathrm{e}\}\ \mathrm{s2}\ \{\mathbf{Q}\}$

Handling loops

Summing integers from 1 to n:

$\{n = n_0\}$

```
    i = 0;
    sum = 0;
```

$\{n = n_0 \wedge sum = \Sigma_{j < i}\, j\}$

```
    while (i < n) {
        sum += i;
        i += 1;
    }
```

$\{sum = \Sigma_{j < n0}\, j\}$

General "while" rule:

To establish:

$\{\mathbf{P}\}$ while (e) { s } $\{\mathbf{Q}\}$

Must show:

$\{\mathbf{P} \wedge e\}\ s\ \{\mathbf{P}\}$

$(\mathbf{P} \wedge \neg e) \rightarrow \mathbf{Q}$

Zooming in on subcases for the example

From precondition, easy to see that final value of sum is:

$(\Sigma_{j < \mathtt{i}}\, j) + \mathtt{i}$

Algebra shows equivalence to postcondition version:

$\Sigma_{j < \mathtt{i}+1}\, j$

case for going once around the loop

$\{\mathtt{n} = n_0 \wedge \mathtt{i} \le \mathtt{n} \wedge \mathtt{sum} = \Sigma_{j < \mathtt{i}}\, j \wedge \mathtt{i} < \mathtt{n}\}$

```
sum += i;
i += 1;
```

$\{\mathtt{n} = n_0 \wedge \mathtt{i} \le \mathtt{n} \wedge \mathtt{sum} = \Sigma_{j < \mathtt{i}}\, j\}$

$\{\mathtt{n} = n_0\}$

```
i = 0;
sum = 0;
```

$\{\mathtt{n} = n_0 \wedge \mathtt{i} \le \mathtt{n} \wedge \mathtt{sum} = \Sigma_{j < \mathtt{i}}\, j\}$

```
while (i < n) {
    sum += i;
    i += 1;
}
```

$\{\mathtt{sum} = \Sigma_{j < n0}\, j\}$

case for exiting the loop

$(\mathtt{n} = n_0 \wedge \mathtt{i} \le \mathtt{n} \wedge \mathtt{sum} = \Sigma_{j < \mathtt{i}}\, j \wedge \mathtt{i} \ge \mathtt{n}) \rightarrow \mathtt{sum} = \Sigma_{j < n0}\, j$

Implication easy to see, after noticing:

i ≤ n and i ≥ n imply i = n.

i = n and n = n_0 imply i = n_0.

So we can replace i with n_0 to get RHS.

i=n

i=n_0

Segment 2: adding arrays

First example revisited

// Assuming: `int x, y`

$\{x = x_0 \wedge y = y_0\}$ Precondition

```
    int tmp = x;
```

$\{x = x_0 \wedge y = y_0 \wedge tmp = x_0\}$

```
    x = y;
```

$\{x = y_0 \wedge y = y_0 \wedge tmp = x_0\}$

Intermediate invariants

```
    y = tmp;
```

$\{x = y_0 \wedge y = x_0\}$ Postcondition

The game is:
Each {…} is an **assertion** that should hold each time we reach it.

Our basic units of proof look like:
{**P**} `stmt` {**Q**}
That means:
Assume **P** in starting state.
Prove that **Q** holds
in ending state,
with no memory errors
while running `stmt`.

Swapping array cells

// Assuming: `int a[42], i, j`

$\{\underbrace{0 \leq \mathtt{i} < 42}\ \wedge\ 0 \leq \mathtt{j} < 42 \wedge \mathtt{a[i]} = x_0 \wedge \mathtt{a[j]} = y_0\}$

```
    int tmp = a[i];
```

$\{0 \leq \mathtt{i} < 42 \wedge 0 \leq \mathtt{j} < 42 \wedge \mathtt{a[i]} = x_0 \wedge \mathtt{a[j]} = y_0 \wedge \mathtt{tmp} = x_0\}$

```
    a[i] = a[j];
```

$\{0 \leq \mathtt{i} < 42 \wedge 0 \leq \mathtt{j} < 42 \wedge \mathtt{a[i]} = y_0 \wedge \mathtt{a[j]} = y_0 \wedge \mathtt{tmp} = x_0\}$

```
    a[j] = tmp;
```

$\{\mathtt{a[i]} = y_0 \wedge \mathtt{a[j]} = x_0\}$

This derivation is easy to carry out, where, in assertions, we treat each `a[...]` cell as a separate variable, additionally requiring that every index used with `a` in the code is between 0 and 41.

Too easy?

// Assuming: `int a[42], i, j`

$\{0 \leq i < 42 \wedge 0 \leq j < 42 \wedge a[i] = x_0 \wedge a[j] = y_0\}$

```
    a[i] = 1;
```

$\{0 \leq i < 42 \wedge 0 \leq j < 42 \wedge a[i] = 1 \wedge a[j] = y_0\}$

```
    a[j] = 2;
```

$\{a[i] = 1 \wedge a[j] = 2\}$

The postcondition we've "proved" here is not guaranteed to hold!

Can you see why?

Aliasing!

The spec doesn't hold when `i` = `j`.
In general, need to watch out for *different ways of writing the same array cell reference*.

One sound solution: treat an array as a single first-class value.

Sound array reasoning

We write **sel**(A, *i*) for looking up the *i*th value in an array value A,
and **upd**(A, *i*, *v*) for computing a new array that is like A, but with cell *i* overwritten with value *v*.

$\{0 \leq \mathtt{i} < 42 \wedge 0 \leq \mathtt{j} < 42 \wedge \mathtt{a} = A_0 \wedge \mathbf{sel}(A_0, \mathtt{i}) = x_0 \wedge \mathbf{sel}(A_0, \mathtt{j}) = y_0\}$

```
int tmp = a[i];
```

$\{0 \leq \mathtt{i} < 42 \wedge 0 \leq \mathtt{j} < 42 \wedge \mathtt{a} = A_0 \wedge \mathbf{sel}(A_0, \mathtt{i}) = x_0 \wedge \mathbf{sel}(A_0, \mathtt{j}) = y_0 \wedge \mathtt{tmp} = \mathbf{sel}(A_0, \mathtt{i})\}$

```
a[i] = a[j];
```

$\{0 \leq \mathtt{i} < 42 \wedge 0 \leq \mathtt{j} < 42 \wedge \mathtt{a} = \mathbf{upd}(A_0, i, \mathbf{sel}(A_0, \mathtt{j})) \wedge \mathbf{sel}(A_0, \mathtt{i}) = x_0 \wedge \mathbf{sel}(A_0, \mathtt{j}) = y_0 \wedge \mathtt{tmp} = x_0\}$

$\{0 \leq \mathtt{i} < 42 \wedge 0 \leq \mathtt{j} < 42 \wedge \mathbf{sel}(\mathtt{a}, \mathtt{i}) = y_0 \wedge \mathbf{sel}(\mathtt{a}, \mathtt{j}) = y_0\}$

Using 2 key algebraic properties:
sel(**upd**(A, *i*, *v*), *i*) = *v*
sel(**upd**(A, *i*, *v*), *j*) = **sel**(A, *j*) [when $i \neq j$]

A quick example mixing arrays and loops

// Assuming: int a[42]

$\{a = A_0\}$

```
   int i = 0;
```

$\{0 \le i \le 42 \wedge (\forall j.\ 0 \le j < i \rightarrow a[j] = A_0[j] + 1) \wedge (\forall j.\ i \le j < 42 \rightarrow a[j] = A_0[j])\}$

```
   while (i < 42) {
      a[i] = a[i] + 1;
   }
```

$\{(\Sigma_{j<42}\ a[j]) = 42 + \Sigma_{j<42}\ A_0[j]\}$

This spec isn't too hard to verify,
working carefully through the code,
modeling an array assignment like an integer assignment,
using **upd**, and simplifying using the **sel-upd** laws.

Segment 3: adding pointers

Swapping again (with pointers)

// Assuming: `int *x, *y`

{x ☞ x_0 * y ☞ y_0}

```
    int tmp = *x;
```

{x ☞ x_0 * y ☞ y_0 ∧ tmp = x_0}

```
    *x = *y;
```

{x ☞ y_0 * y ☞ y_0 ∧ tmp = x_0}

```
    *y = tmp;
```

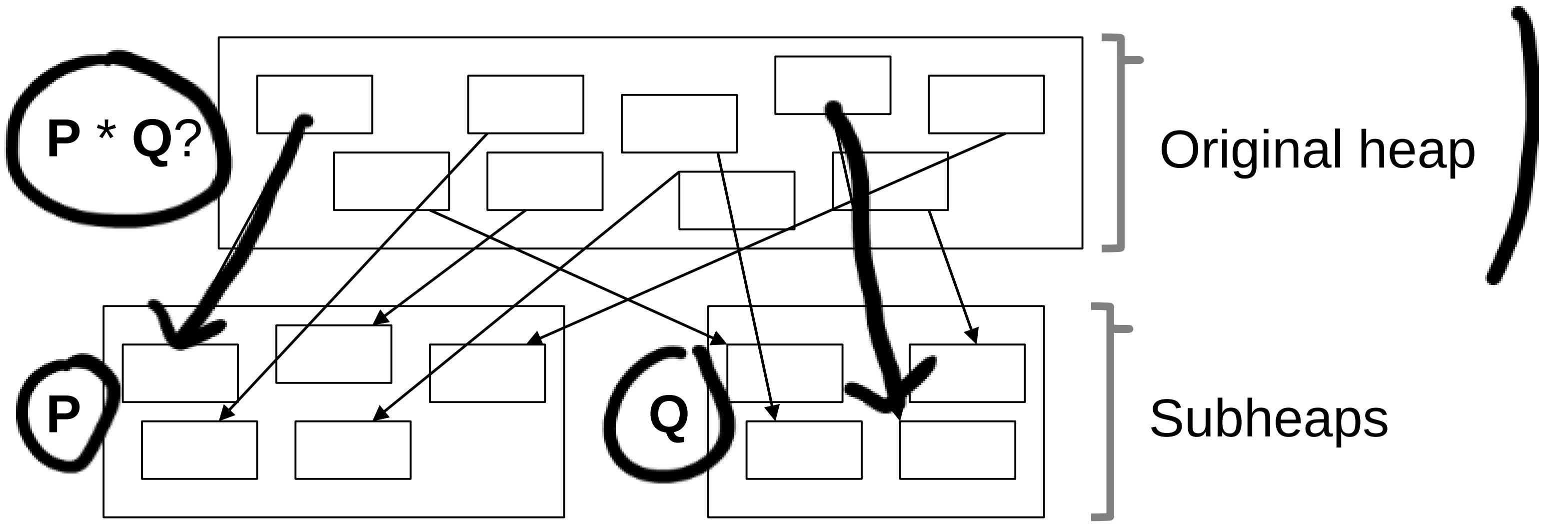
{x ☞ y_0 * y ☞ x_0}

An assertion *p* ☞ *v* says that *p* is a valid pointer, pointing to *v* in the current memory.

We also write **P * Q** for a separating conjunction, originated in a Hoare-logic extension called **separation logic**.

Meaning of **P** * **Q**:
Heap can be partitioned into two disjoint subheaps, one satisfying **P** and the other **Q**.

Separating conjunction “*” explained pictorially



Meaning of **P** * **Q**:
Heap can be partitioned into two disjoint subheaps, one satisfying **P** and the other **Q**.

Sound rules for reading and writing pointers

General "read" rule:

{x ☞ *v*}
y = *x;
{x ☞ *v* ∧ y = *v*}

Find a "points-to" fact for the pointer and just use the value you see there.

General "write" rule:

{y ☞ v_0 ∧ x = *v*}
*y = x;
{y ☞ *v* ∧ x = *v*}

Find a "points-to" fact for the pointer and overwrite its value with the new one being written.

Important side condition: logical expression *v* may not mention the program variable y!
(Also, x & y must be distinct.)

It's not hard to generalize these rules to handle more complex operations, combining multiple reads and writes.

Modular reasoning: the frame rule

<u>If you proved:</u>

{**P**}

    stmt

{**Q**}

<u>You may automatically conclude:</u>

{**P** * **R**}

    stmt

{**Q** * **R**}

for any **R** that doesn't mention program variables changed by `stmt`.

Example: earlier, we effectively proved this spec for a "swap" function:

{x ☞ x_0 * y ☞ y_0}
    swap(x, y);
{x ☞ y_0 * y ☞ x_0}

Step 1: instantiate spec for two calls to swap:

{p ☞ p_0 * q ☞ q_0}
    swap(p, q);
{p ☞ q_0 * q ☞ p_0}

{p ☞ q_0 * r ☞ r_0}
    swap(p, r);
{p ☞ r_0 * r ☞ q_0}

Step 2: extend specs with frame rule:

{p ☞ p_0 * q ☞ q_0 * r ☞ r_0}
    swap(p, q);
{p ☞ q_0 * q ☞ p_0 * r ☞ r_0}

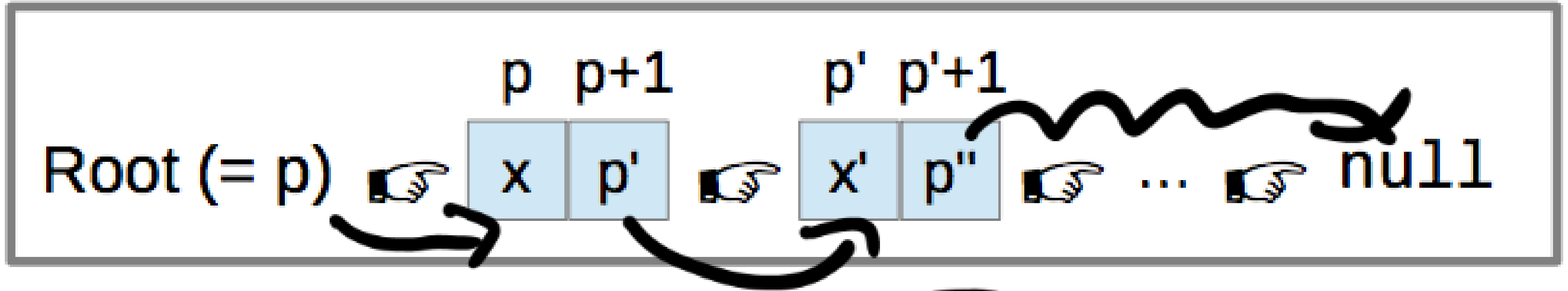
{p ☞ q_0 * r ☞ r_0 * q ☞ p_0}
    swap(p, r);
{p ☞ r_0 * r ☞ q_0 * q ☞ p_0}

Step 3: put the two calls in sequence:

{p ☞ p_0 * q ☞ q_0 * r ☞ r_0}
    swap(p, q);
{p ☞ q_0 * q ☞ p_0 * r ☞ r_0}
    swap(p, r);
{p ☞ r_0 * r ☞ q_0 * q ☞ p_0}

Segment 4: adding linked data structures

A recursive definition of linked lists



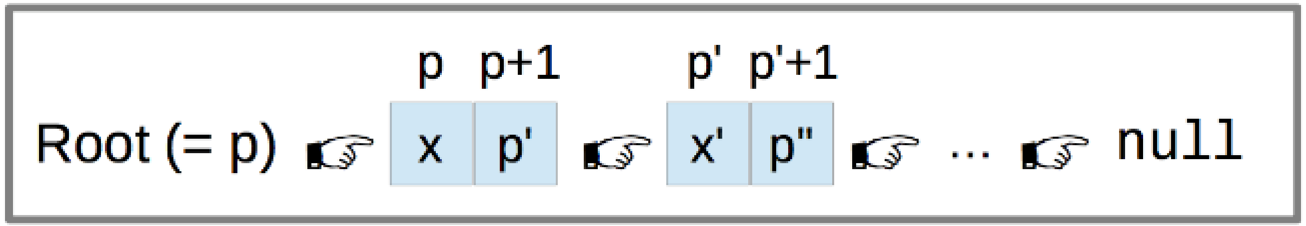
$$\mathbf{list}(p) \stackrel{\text{def}}{=} p = \mathtt{null} \lor (p \neq \mathtt{null} \land (\exists x.\ p \text{ ☞ } x) * (\exists p'.\ p{+}1 \text{ ☞ } p' * \mathbf{list}(p')))$$

Subtlety that we're steering clear of: what makes a recursive predicate definition legal?
E.g., consider the dubious definition $\mathbf{list}(p) \stackrel{\text{def}}{=} \neg\mathbf{list}(p)$.
Take my word for it that the other one here is OK. :)

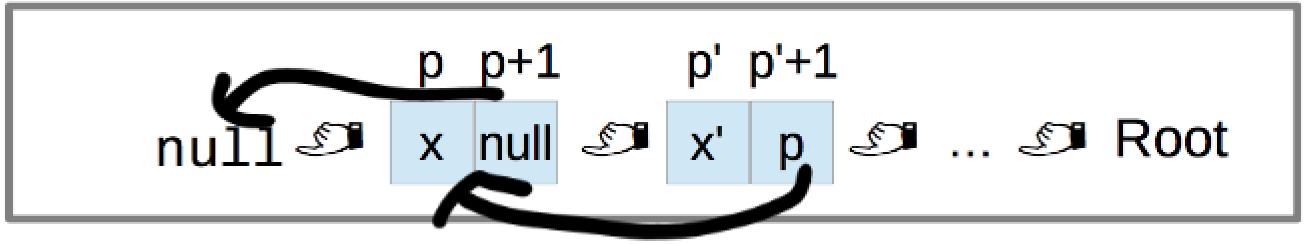
In-place reversal of a linked list

```
{list(p)}
    list *acc = null;
{list(p) * list(acc)}
    while (p != null) {
        list *tmp = p->next;
        p->next = acc;
        acc = p;
        p = tmp;
    }
{list(acc)}
```

Before:



After:



The assertions here are nice and straightforward, thanks to the power of separation logic!

Exercise for the reader: work through the proof, using the rules we've seen already, plus judicious "unfolding" of the **list** predicate, replacing certain uses with the definition.

More modularity, via the frame rule

We effectively just verified this function:

{**list**(p)}
```
  p' = reverse(p);
```
{**list**(p')}

Step 1: apply to two different lists:

{**list**(q)}
```
  q = reverse(q);
```
{**list**(q)}

{**list**(r)}
```
  r = reverse(r);
```
{**list**(r)}

Step 2: extend specs with frame rule:

{**list**(q) * **list**(r)}
```
  q = reverse(q);
```
{**list**(q) * **list**(r)}

{**list**(r) * **list**(q)}
```
  r = reverse(r);
```
{**list**(r) * **list**(q)}

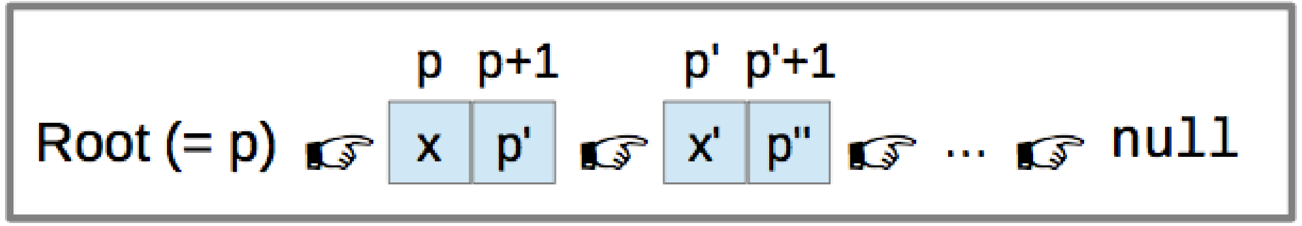
Step 3: put the two calls in sequence:

{**list**(q) * **list**(r)}
```
  q = reverse(q);
```
{**list**(q) * **list**(r)}
```
  r = reverse(r);
```
{**list**(r) * **list**(q)}

Free reasoning about (lack of) aliasing in linked structures!

Segment 5: proving program-specific semantic properties

Augmenting our recursive definition of linked lists



Use two operators for constructing mathematical lists (sequences):
nil for the empty list, $x :: L$ for the list that has x added to the front of L.

$$\mathbf{list}(p, L) \stackrel{\text{def}}{=} (L = \mathbf{nil} \wedge p = \mathtt{null})$$
$$\vee\ \exists x.\ \exists L'.\ \exists p'.\ L = x :: L' \wedge p \neq \mathtt{null} \wedge p \mapsto x * p{+}1 \mapsto p' * \mathbf{list}(p', L')$$

Now **list** predicate captures not just the idea,

"there is a linked list rooted at this address,"

but it also says,

"the linked list rooted here *encodes* this particular mathematical sequence."

In-place reversal of a linked list, with a stronger spec

{**list**(p, L_0)}

```
    list *acc = null;
```
{$\exists L_1$. $\exists L_2$. ($L_0 = L_1$ ++ L_2) ∧ **list**(p, L_2) * **list**(acc, **rev**(L_1))}

```
    while (p != null) {
        list *tmp = p->next;
        p->next = acc;
        acc = p;
        p = tmp;
    }
```
{**list**(acc, **rev**(L_0))}

Here we use two more mathematical list operators in our specs:
L1 ++ L2: list concatenation
rev(L): list reversal

(These operators are actually the same ones as appear in functional programming languages like Haskell and OCaml!)

Notice that the *program* is unchanged from our earlier example with a weaker spec.
The *proof* is just a little more involved,
with new reasoning about the algebraic properties of "++" and **rev**.

More modularity, via the frame rule

We effectively just verified this function:

{**list**(p, *L*)}
```
  p' = reverse(p);
```
{**list**(p', **rev**(*L*))}

Step 1: apply to two different lists:

{**list**(q, *Q*)}
```
  q = reverse(q);
```
{**list**(q, **rev**(*Q*))}

{**list**(r, *R*)}
```
  r = reverse(r);
```
{**list**(r, **rev**(*R*))}

Step 2: extend specs with frame rule:

{**list**(q, *Q*) * **list**(r, *R*)}
```
  q = reverse(q);
```
{**list**(q, **rev**(*Q*)) * **list**(r, *R*)}

{**list**(r, *R*) * **list**(q, **rev**(*Q*))}
```
  r = reverse(r);
```
{**list**(r, **rev**(*R*)) * **list**(q, **rev**(*Q*))}

Step 3: put the two calls in sequence:

{**list**(q, *Q*) * **list**(r, *R*)}
```
  q = reverse(q);
```
{**list**(q, **rev**(*Q*)) * **list**(r, *R*)}
```
  r = reverse(r);
```
{**list**(r, **rev**(*R*)) * **list**(q, **rev**(*Q*))}

Or reason about running `reverse` twice

Function spec:
{**list**(p, *L*)}
```
  p' = reverse(p);
```
{**list**(p', *L*)}

Step 1: instantiate spec twice:

{**list**(q, *Q*)}
```
  q = reverse(q);
```
{**list**(q, **rev**(*Q*))}

{**list**(q, **rev**(*Q*))}
```
  q = reverse(q);
```
{**list**(q, **rev**(**rev**(*Q*)))}

Step 2: put the two calls in sequence:

{**list**(q, *Q*)}
```
  q = reverse(q);
```
{**list**(q, **rev**(*Q*))}
```
  q = reverse(q);
```
{**list**(q, **rev**(**rev**(*Q*)))}

Step 3: use algebraic properties of **rev** to prove that postcondition implies:
{**list**(q, *Q*)}

(That is, we're back where we started.)

Coda: pointers to further reading and tools

Pointers to further reading & tools

Standalone, mostly automatic program verifiers:

Dafny, including in-browser demo with puzzles!

<http://research.microsoft.com/en-us/projects/dafny/>

VeriFast, based on separation logic

<http://people.cs.kuleuven.be/~bart.jacobs/verifast/>

Work done within proof assistants, general platforms for computerized proofs:

L4.verified, an operating system proved to meet a functional spec, using separation logic

<http://www.ertos.nicta.com/research/l4.verified/>

Bedrock, my own project that has applied separation logic to a Web server w/ dynamic content

<http://plv.csail.mit.edu/bedrock/>

To learn more about Hoare logic and the foundations of computerized program proof:

Software Foundations, a free online textbook

<http://www.cis.upenn.edu/~bcpierce/sf/>

THANK YOU

Adam Chlipala
Associate Professor without Tenure