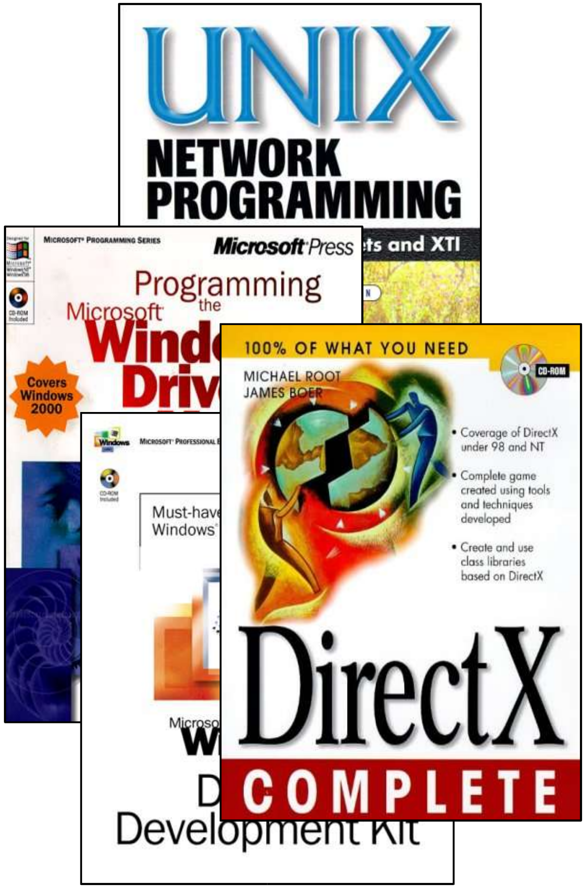
Specifying and Checking Stateful Software Interfaces

Manuel Fähndrich maf@microsoft.com
Microsoft Research

2005 Summer School on Reliable Computing
Eugene, Oregon

The world is stateful!

- API documentation is full of rules
  - Governing order of operations & data access
  - Explaining resource management
- Disobeying a rule causes bad behavior
  - Unexpected exceptions
  - Failed runtime checks
  - Leaked resources
- Rules are informal
  - Usually incomplete (bad examples, good examples)
  - Not enforced



The state of the world

Existing languages too permissive

- Compilers do not catch enough bad programs (why?)
- Cannot specify stricter usage rules

Programmers overwhelmed with complexity

- Did I cover all cases?
  - Do I even know all possible cases?
- Did I think through all paths?
- Did I consider all aliasing combinations?
- Did I consider all thread interactions?
- Did I handle all messages?

Language-based approach

- Methodology not after-the-fact analysis
  - Language provides a programming model for correct usage
  - Language makes failures explicit
  - Make programmers deal with failures
  - Guide programmer from the beginning
- Modularity
  - Programmer has to write interface specifications
  - Specifications of interfaces for components, data, and functions are part of the program
- Compiler or checkers enforce the correct usage rules
  - Trade-off between expressiveness and automation
  - Approach from tractable end; grow expressiveness

Specifications reduce Complexity

Every pre-condition/invariant rules out one or more cases/paths in a procedure

- Specify sub-ranges:
  [A | B | C]   possibly-null(T) vs. non-null(T)
- Make impossible paths obvious
- State (non-) aliasing assumptions
- Specify legal thread interactions

```
void f(T *x) {
  if (x == NULL) {
    // now what?
  }
  T y = *x;
  ...
}
```

```
void f(T *x) {
  T y = *x;
  ...
}
```

buggy

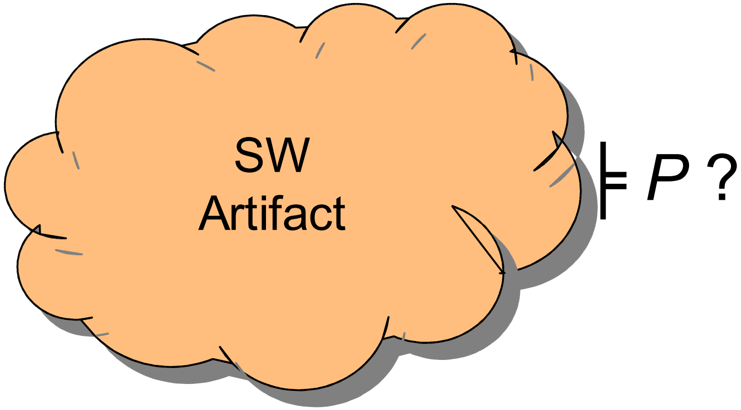
```
void f(T *!x) {
  T y = *x;
  ...
}
```

ideal

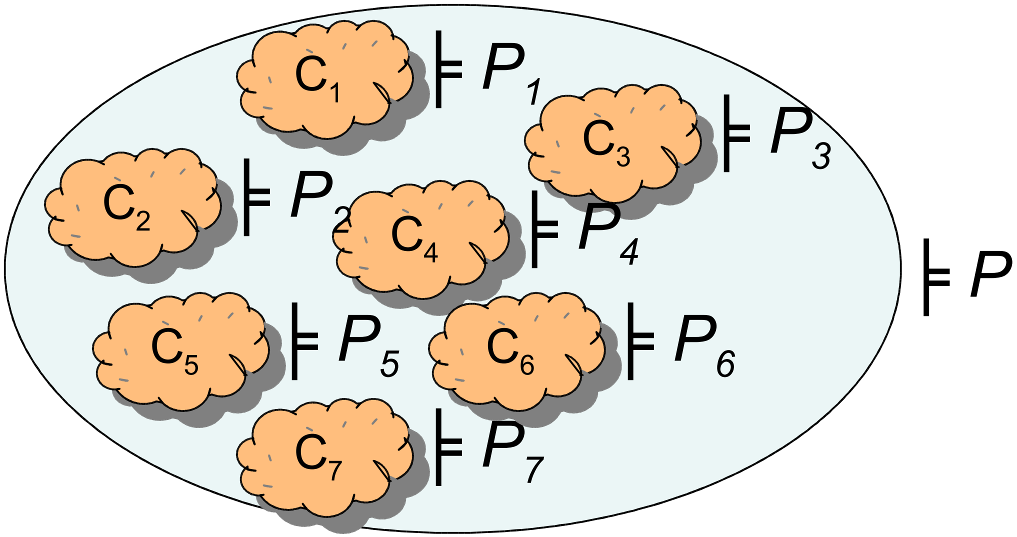
defensive

Modularity: making checking tractable

Monolithic



Modular



Modularity advantages

- More powerful
- Early error detection
- Robustness
- Incremental (open)

Modularity drawbacks

- Rigid
- Have to think ahead
- Tedious

Lecture Outline

- Motivation and Context
  - Reason about imperative programs
  - Specify behavior
  - Check code against specification
- Lecture approach
  - Start with a specification problem
  - Bring in technical background
  - Point out limitations
  - Relate to practical experience

What would you like to specify today?

- Allocation/Deallocation
- Memory initialization
- Locks
- Events
- Type states
- Regions
- Reference counting
- Sharing
- Communication channels
- Deadlock freedom

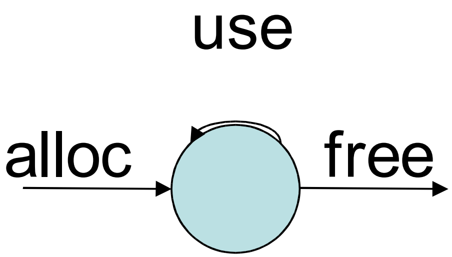
Technical material

- Type systems with state
  - linear types
  - capability-based systems
- Programming models
- Object type-states

Demo?

Allocation/Deallocation

- Familiar protocol
- Rules
  - free when done
  - don’t use after free
  - don’t free twice



- Although the world is stateful, …

  …all I ever needed to know I learned from the functional programming community!

Linear Types can Change the World

- Paper by Wadler 1990
  - From linear logic to linear types
  - Purely functional setting

    $e ::= n\ j\ (e, e)\ j\ \text{let } p = e_1 \text{ in } e_2\ j\ e_1\ e_2\ j\ \lambda x.e$
  - Conventional types

    $\tau ::= \text{int}\ j\ \tau\ £\ \tau\ j\ \text{int[]}\ j\ \tau\ !\ \tau\ j\ \text{unit}$
  - Array functions

    lookup : int[] ! int ! int

    update : int[] ! int ! int ! int[]
  - Problem: to update an array, it must be copied so as to leave original unchanged

Conventional functional array update

```
let x = new int[1] in
let x = update x 0 9 in
let y = update x 0 8 in
let a = lookup x 0 in
let b = lookup y 0 in
assert (a == 9) in
assert (b == 8) in
()
```

- Often, original array no longer needed.
- Would like to eliminate copy in those cases.

Conventional type rules

Rules of the form: $A \vdash e : \tau$

where $A ::= x : \tau \mid A,A$

$$\frac{}{A, x : \tau \vdash x : \tau}\,[\mathsf{var}]$$

$$\frac{\begin{array}{c} A \vdash e_1 : \tau_1 \\ A, x : \tau_1 \vdash e_2 : \tau_2 \end{array}}{A \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2}\,[\mathsf{let}]$$

- Key feature:

  Assumption x:τ can be used 0, 1, or many times

Enter linear types

- Linear types: τ ::= .. j τ - τ j int[]^2 j τ (τ
- Judgments: A ` e : τ
- Key feature: Each assumption x:τ used exactly once

$$\frac{}{x : \tau \vdash x : \tau}\,[\text{var}]$$

$$\frac{\begin{array}{c} A_1 \vdash e_1 : \tau_1 \\ A_2, x : \tau_1 \vdash e_2 : \tau_2 \end{array}}{A_1, A_2 \vdash \text{let}\, x = e_1 \text{in}\, e_2 : \tau_2}\,[\text{let}]$$

$$\frac{\begin{array}{l} A_1 \vdash e_1 : \tau_2 \multimap \tau_r \\ A_2 \vdash e_2 : \tau_2 \end{array}}{A_1, A_2 \vdash e_1\ \ e_2 : \tau_r}\,[\text{app}]$$

$$\frac{\begin{array}{l} A_1 \vdash e_1 : \tau_1 \otimes \tau_2 \\ A_2, x : \tau_1, y : \tau_2 \vdash e_2 : \tau_3 \end{array}}{A_1, A_2 \vdash \text{let}\,(x, y) = e_1 \text{in}\, e_2 : \tau_3}\,[\text{letp}]$$

Linear arrays

- Array functions

lookup : int[]^2 ! int (int - int[]^2

update : int[]^2 ! int (int (int[]^2

$$\frac{A, x : \tau_1 \vdash e : \tau_2}{A \vdash \lambda x.e : \tau_1 \multimap \tau_2}\text{[lambda]}$$

$$\frac{x : \tau_1 \vdash e : \tau_2}{\cdot \ \vdash \lambda x.e : \tau_1 \to \tau_2}\text{[lambda-nl]}$$

Linear functional array update

let x_0 = **new** int[1] in
let x_1 = update x_0 0 9 in
let y = update x_1 0 8 in
let (a,x_2) = lookup x_1 0 in
…

- Does not type check. Why?

$$\frac{\begin{array}{r} x_1 : \mathsf{int}[]^\bullet \vdash \mathsf{update}\ldots \\ y : \mathsf{int}[]^\bullet \vdash \mathsf{let}\,(a, x_2) = \ldots \end{array}}{x_1 : \mathsf{int}[]^\bullet \vdash \mathsf{let}\ y = \mathsf{update}\,x_1\,0\,8\,\mathsf{in}\ \mathsf{let}\,(a, x_2) = \ldots}$$

- No need to copy if everything is used once only!
- *update* function can actually update array in place.

Observations on Linearity

- Value of linear type is like a coin
  - You can spend it, but you can spend it only once
- Single threading of arrays
  - Similar to store threading in denotational semantics
- Advantages
  - No leaks : if program type check, no left-overs
  - Memory can be reused
- Does it address our resource management specification problem?

Modeling with linear types

- Resource protocol

  alloc : unit ! T^2

  use : T^2 ! T^2

  free : T^2 ! unit

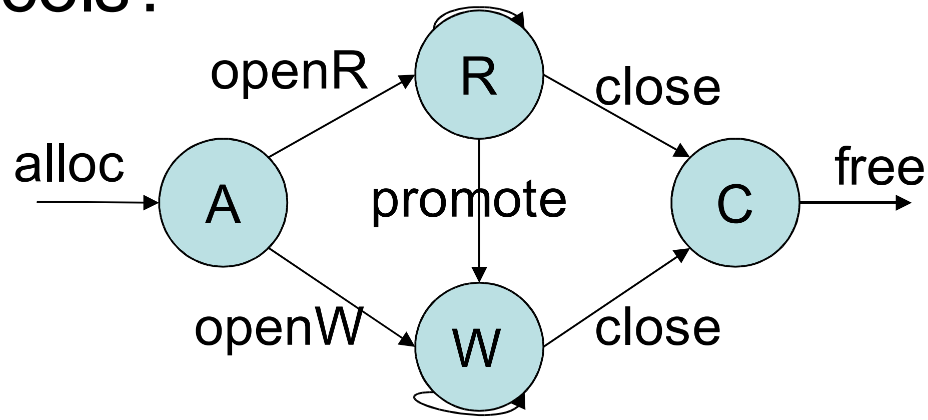
- File protocol

  open : string ! File^2

  read : File^2 ! File^2

  close : File^2 ! unit

More complicated protocols?



Type-state modeling

- Complex file protocol

  alloc : unit ! AFile²

  openR : AFile² ! RFile²

  openW: AFile² ! WFile²

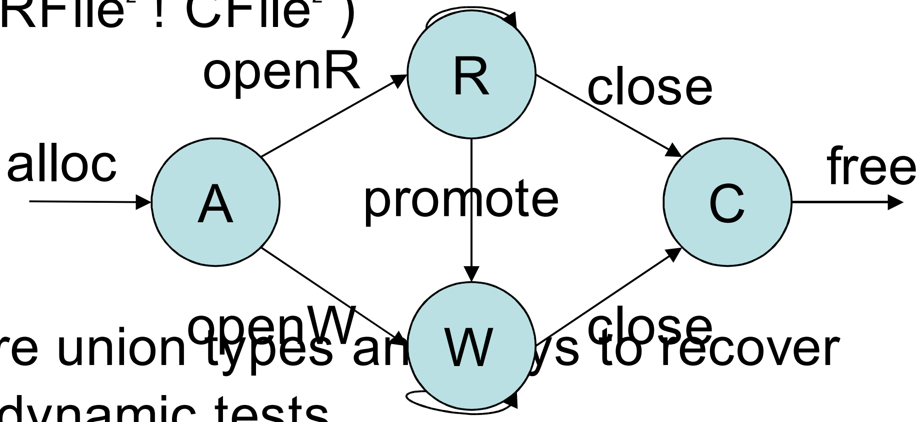
  read: RFile² ! RFile²

  write : WFile² ! WFile²

  close: (WFile² ! CFile²) Æ (RFile² ! CFile²)

  free : CFile² ! unit

- Observations
  - One type per type-state
  - DFA’s easy, NFA’s require union types an ys to recover type information through dynamic tests



Summary so far

- Problem of checking
  - Resource management
  - Type state rules

…reduced to type checking.

$$\Gamma \vdash p : \tau$$

Problems with Linear Types

- Use = Consume
- Style (single threading)
- What if I do want to use things multiple times?
  - explicit copy
- Single pointer invariant
- Can we use linear types for all data structures?
  - As long as they are trees!

Non-linear types for non-trees

- Two environments or explicit copy and destroy
- Assume explicit duplication
  - Non-linear values can be duplicated for free (no runtime cost)
  - let (x,y) = **copy** e_1 in e_2
  - Requires that e_1 has non-linear type
- When is a type non-linear?
  - Wadler says: if top-level constructor is non-linear
- What about a type like: int[]^2 £ int[]^2
  - a non-linear pair of linear arrays
  - cannot be duplicated without actual runtime copy
  - linear type systems disallow such types

Temporary non-linear access

- **let**! (x) y = e_1 **in** e_2
- Like **let** y = e_1 **in** e_2, but x given non-linear type in e_1, then reverts to linear type in e_2

$$\frac{\begin{array}{c}A_1, x :!\tau \vdash e_1 : \tau_1 \\ A_2, x : \tau, y : \tau_1 \vdash e_2 : \tau_2\end{array}}{A_1, A_2, x : \tau \vdash \mathsf{let}\,(x)\;\; y = e_1 \mathsf{in}\, e_2 : \tau_2}[\mathsf{let!}]\quad \textit{ugly side}$$

- $!\tau$: operator stripping of circles (recursively)
- Assume: lookup : int[] ! int ! int

- Example: **let** x = new int[2] **in**

  let! (x) a = lookup x 0 **in**

  let! (x) b = lookup x 1 **in** …

Modeling with linear types

- Allocation/Deallocation ➢
- Memory initialization
- Locks
- Events
- Type states ➢
- Object states
- Regions
- Reference counting
- Sharing
- Channels
- Deadlock freedom

Locking (1)

τ ::= Lockh T^{2}i non-linear type

create : T^{2} ! LockhT^{2}i

acquire : Lockh T^{2}i ! T^{2}

release : Lockh T^{2}i ! T^{2} ! unit

- Model
  - Lock contains and protects some linear data T^{2}
  - Acquire blocks until lock is available and returns T^{2}
  - Release releases lock and specifies new data
- Lingering errors?
  - Double acquire
  - Never release
  - Double release

Locking (2)

- Avoid forgetting to release

τ ::= … j RToken^{2}

acquire : Lockh T^{2}i ! T^{2} - RToken^{2}

release : Lockh T^{2}i ! T^{2}- RToken^{2} ! unit

- Model
  - Can only release as many times as we acquired
  - Won’t forget to release
    (no other way to get rid of RToken)
  - Can still double release though
  - Release wrong lock

Summary of Linear type systems

- Linearity controls the creation and uses of aliases
  - Each type assumption used exactly once
- Can express
  - resource management
  - type state protocols
  - some locking
- Good for
  - purely functional contexts
  - single pointer invariant
  - single-threading style

Where are the Programming Languages?

- Simple, empowering technique
- Programming language: Concurrent Clean
- Problems
  - Style
  - To overcome, things get messy
  - Dichotomy between non-linear and linear data
    Linear vs. non-linear choice at birth, fixed, except for **let!**
  - **let!** has problems
  - No linear data in non-linear data
  - No correlations (e.g., lock and release token)
  - No control over non-linear data

- **Big problem**: World is still imperative

Specification tasks

- Allocation/Deallocation ⊳
- Memory initialization
- Locks (⊳)
- Events
- Type states ⊳
- Object states
- Regions
- Reference counting
- Sharing
- Channels
- Deadlock freedom

Initialization is imperative!

- TAL allocation problem (Morrisett et.al.)
- How to allocate C(5) ?
    - datatype t = C of int | D

```
ld.w r0 = alloc 8;
st.w r0[0] = CTag;
st.w r0[4] = 5;
```

- at this point: need to prove that $r_0 : \mathsf{t}$
- intermediate steps

1. $r_0 : \langle s(0), s(0) \rangle$
2. $r_0 : \langle s(\mathsf{CTag}), s(0) \rangle$
3. $r_0 : \langle s(\mathsf{CTag}), s(5) \rangle$
4. $r_0 : \langle s(\mathsf{CTag}), \mathsf{int} \rangle$

Singleton Type Aside

- A type denoting a single value.
- τ ::= s(i) j …
- i ::= n *constant int*
  j ρ *symbolic int*
- Given x : s(i), we know that «x¬ = «i¬ in all evaluations.

TAL allocation problem

- Allocation happens in many small steps
- Must be able to type each intermediate configuration
- Updates must be *strong,* i.e., they change the type
- Key insight: model after dynamic semantics

E : Var ! Loc	Environment
M : Loc ! Val	Store

- At type level
  - Separate pointers from permissions
  - Split environment assumptions into
    - Non-linear type assumptions
    - Linear capabilities
  - Make explicit which operations
    - require capabilities
    - consume capabilities

Alias Types and Capabilities

- Use singleton types for pointers

τ::= pt(i) j int j … *non-linear types*

h::= h$\sigma_1..\sigma_n$i j τ[] j 9[Δ | C].h *heap block types*

σ ::= 9[Δ | C].σ j … j ́τ*linear types*

- Use explicit heap: A; C ` e : σ; C’

“In environment A, given a heap described by capabilities C, e evaluates to some value v, such that v : τ, and the final heap is described by C’ ”

A ::= ¢ j x : τ, A

C ::= ¢ j { i ↦ h } - C j …

Δ ::= ¢ j ρ,Δ

Capability Type Rules

$$\frac{}{A, x : \tau; C \vdash x : \tau; C}\text{[var]}$$

$$\frac{A; C_1 \vdash e_1 : \tau_1; C_2 \qquad A, x : \tau_1; C_2 \vdash e_2 : \tau_2; C_3}{A; C_1 \vdash \mathsf{let}\, x = e_1 \mathsf{in}\, e_2 : \tau_2; C_3}\text{[let]}$$

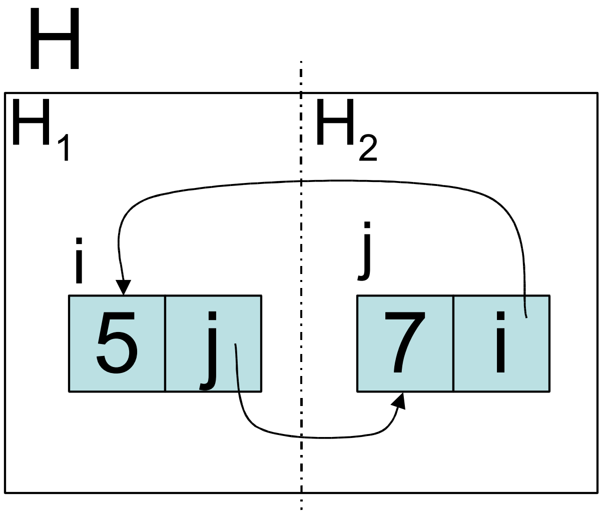
$$\frac{C_2 = \{\rho \mapsto \langle s(0)..s(0)\rangle\} \otimes C_1}{A; C_1 \vdash \mathsf{alloc}(n) : \mathsf{pt}(\rho); C_2}\text{[alloc]}$$

$$\frac{A; C_1 \vdash e : \mathsf{pt}(i); C_2 \qquad C_2 = \{i \mapsto \langle \tau_1..\tau_n\rangle\} \otimes C_3}{A; C_1 \vdash \mathsf{free}\, e : \mathsf{unit}; C_3}\text{[free]}$$

Spatial Conjunction

- H : heaps
- H ² C Heap H is described by C

$$\frac{\begin{array}{l} H = H_1 \uplus H_2 \\ H_1 \models C_1 \\ H_2 \models C_2 \end{array}}{H \models C_1 \otimes C_2}$$



$$\begin{aligned} C &= C_1 \otimes C_2 \\ C_1 &= \{i \mapsto \langle \mathsf{int}, \mathsf{pt}(j) \rangle\} \\ C_2 &= \{j \mapsto \langle \mathsf{int}, \mathsf{pt}(i) \rangle\} \end{aligned}$$

Capability Type Rules (2)

$$\frac{A; C_1 \vdash e : \mathsf{pt}(i); C_2 \qquad C_2 = \{i \mapsto \langle \tau_1..\tau_n \rangle\} \otimes C_3}{A; C_1 \vdash e.k : \tau_k; C_2}\text{[load]}$$

$$\frac{A; C_1 \vdash e : \mathsf{pt}(i); C_2 \qquad A; C_2 \vdash e' : \tau; C_3 \qquad C_3 = \{i \mapsto \langle \tau_1..\tau_n \rangle\} \otimes C_4}{A; C_1 \vdash e.k{:=}e' : \mathsf{unit}; \{i \mapsto \langle \tau_1..\tau_{k-1}, \tau, \tau_{k+1}..\tau_n \rangle\} \otimes C_4}\text{[store]}$$

Allocation revisited

r_0 = alloc 8;

$r_0 : \mathsf{pt}(\rho) \qquad C = \{\rho \mapsto \langle s(0), s(0)\rangle\}$

r_0.1 = CTag;

$r_0 : \mathsf{pt}(\rho) \qquad C = \{\rho \mapsto \langle s(CTag), s(0)\rangle\}$

r_0.2 = 5;

$r_0 : \mathsf{pt}(\rho) \qquad C = \{\rho \mapsto \langle s(CTag), \mathsf{int}\rangle\}$

$r_0 : \mathsf{pt}(\rho) \qquad C = \{\rho \mapsto \mathsf{t}\}$

Observations

- Capability rules look similar to Hoare triples

  $A; C \vdash e : \sigma; C'$

  $\{ P \}\ e\ \{ Q \}$

- Logic of capabilities is not first order logic, but a specialized logic for heaps
  - separation logic, logic of bunched implications
  - usually restricted to be tractable

End of Lecture 1