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

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

buggy

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

ideal
Modularity: making checking tractable

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

Modularity drawbacks
- Rigid
- Have to think ahead
- Tedious

Monolithic

Modular
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
  - \text{lookup} : \text{int[]} \; ! \; \text{int} \; ! \; \text{int}
  - \text{update} : \text{int[]} \; ! \; \text{int} \; ! \; \text{int} \; ! \; \text{int[]}
- Problem: to update an array, it must be copied so as to leave original unchanged
Conventional functional array update

```ocaml
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 \( A ::= x : \tau \mid A,A \)

\[ \frac{}{A, x : \tau \vdash x : \tau} \text{[var]} \]
\[ \frac{A \vdash e_1 : \tau_1}{A, x : \tau_1 \vdash e_2 : \tau_2} \text{[let]} \]
\[ A \vdash \text{let } x = e_1 \text{ in } e_2 \]

- Key feature:
  Assumption \( x : \tau \) can be used 0, 1, or many times
Enter linear types

- Linear types: \( \tau ::= .. \ j \tau \ - \ tau \ j \text{int}[^2] \ j \tau \ ( \tau \)

- Judgments: \( A ` e : \tau \)

- Key feature: Each assumption \( x : \tau \) used exactly once
Linear arrays

- Array functions
  
  **lookup** : `int[]² → int ( int - int[]²`
  
  **update** : `int[]² → int ( int ( int[]²`

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

\[
\frac{x : \tau_1 \vdash e : \tau_2}{\vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \quad \text{[lambda-nl]}
\]
Linear functional array update

let $x_0 = \text{new} \ int[1]$ in
let $x_1 = \text{update} \ x_0 \ 0 \ 9$ in
let $y = \text{update} \ x_1 \ 0 \ 8$ in
let $(a, x_2) = \text{lookup} \ x_1 \ 0$ in

... 

- Does not type check. Why?

\[
x_1 : \text{int[]} \vdash \text{update}...
\]
\[
y : \text{int[]} \vdash \text{let} \ (a, x_2) = ...
\]
\[
x_1 : \text{int[]} \vdash \text{let} \ y = \text{update} \ x_1 \ 0 \ 8 \text{ in let} \ (a, x_2) = ...
\]

- No need to copy if everything is used once only!
- \textit{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 ! T²
  - use : T² ! T²
  - free : T² ! unit

- **File protocol**
  - open : string ! File²
  - read : File² ! File²
  - close : File² ! unit

More complicated protocols?

![Diagram of resource protocol and file protocol interactions]

Stateful Software Interfaces
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 and ways to recover type information through dynamic tests
Summary so far

- Problem of checking
  - Resource management
  - Type state rules

...reduced to type checking.

\[ \phi \ ` p : \tau \]
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 \((x, y) = \text{copy } e_1 \text{ in } e_2\)
  - 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: \(\text{int}[^2] \triangleleft \text{int}[^2]\)
  - 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 \(e_2\)

\[
\begin{align*}
A_1, x : \!\tau \vdash e_1 : \tau_1 \\
A_2, x : \tau, y : \tau_1 \vdash e_2 : \tau_2
\end{align*}
\]

\[
A_1, A_2, x : \tau \vdash \text{let} (x) \ y = e_1 \ in \ e_2 : \tau_2
\]

**Example:**

- Operator stripping of \([2]\) sides (recursively)

- Assume: \(\text{lookup} : \text{Int}[] : \text{int} \ in \text{int}\)

  \[
  \begin{align*}
  \text{let!} \ (x) \ a = \text{lookup} \ x \ 0 \ in \\
  \text{let!} \ (x) \ b = \text{lookup} \ x \ 1 \ in...
  \end{align*}
  \]
Modeling with linear types

- Allocation/Deallocation
- Memory initialization
- Locks
- Events
- Type states
- Object states
- Regions
- Reference counting
- Sharing
- Channels
- Deadlock freedom
\[ \tau ::= \text{Lock} T^i \quad \text{non-linear type} \]

- **create** : \( T^2 ! \text{Lock} h T^i \)
- **acquire** : \( \text{Lock} h T^i ! T^2 \)
- **release** : \( \text{Lock} h T^i ! T^2 ! \text{unit} \)

- **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

\[ \tau ::= \ldots j \text{ RToken}^2 \]

- 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

- `acquire`: Lockh T^2i ! T^2 - RToken^2
- `release`: Lockh T^2i ! T^2 - RToken^2 ! unit
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 \( r_0 : t \)

- intermediate steps
  1. \( r_0 : \langle s(0), s(0) \rangle \)
  2. \( r_0 : \langle s(CTag), s(0) \rangle \)
  3. \( r_0 : \langle s(CTag), s(5) \rangle \)
  4. \( r_0 : \langle s(CTag), \text{int} \rangle \)
Singleton Type Aside

- A type denoting a single value.
- $\tau ::= s(i) \ j \ldots$
- $i ::= n \ constant \ int$
  \[ j \rho \ symbolic \ int \]
- Given $x : s(i)$, we know that $\ll x \rr = \ll i \rr$ 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
  
  \[
  \begin{align*}
  E : \text{Var}!\text{Loc} & \quad \text{Environment} \\
  M : \text{Loc}!\text{Val} & \quad \text{Store}
  \end{align*}
  \]

- 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
  \[ \tau ::= \text{pt}(i) \ j \ \text{int} \ j \ldots \]
  \[ h ::= h_{\sigma_1..\sigma_{n}} \ j \ \tau[\ ] j \ 9[\Delta \mid C].h \]
  \[ \sigma ::= 9[\Delta \mid C].\sigma \ j \ldots \ j \tau \text{linear types} \]

- Use explicit heap: \( A; C \ ` e : \sigma; C' \)
  “In environment A, given a heap described by capabilities C, e evaluates to some value v, such that v : \tau, and the final heap is described by C’”

\[ A ::= \emptyset \ j \ x : \tau, A \]
\[ C ::= \emptyset \ j \ \{ \ i \mapsto h \} - C \ j \ldots \]
\[ \Delta ::= \emptyset \ j \ \rho, \Delta \]


**Capability Type Rules**

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

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

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

\[
\frac{A; C_1 \vdash e : \text{pt}(i); C_2}{C_2 = \{ i \mapsto \langle \tau_1 \ldots \tau_n \rangle \} \otimes C_3} \quad [\text{free}]
\]

\[
\frac{A; C_1 \vdash \text{free } e : \text{unit}; C_3}{C_2 = \{ \rho \mapsto \langle s(0) \ldots s(0) \rangle \} \otimes C_3} \quad [\text{free}]
\]
Spatial Conjunction

- $H : \text{heaps}$
- $H \sqcup C$  Heap $H$ is described by $C$

\[
\begin{align*}
H &= H_1 \sqcup H_2 \\
H_1 &\models C_1 \\
H_2 &\models C_2 \\
\hline
H &\models C_1 \otimes C_2
\end{align*}
\]

\[
C = C_1 \otimes C_2 \\
C_1 = \{i \mapsto \langle \text{int}, \text{pt}(j) \rangle \} \\
C_2 = \{j \mapsto \langle \text{int}, \text{pt}(i) \rangle \}
\]
Capability Type Rules (2)

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

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

\[ r_0 = \text{alloc } 8; \]
\[ r_0 : \text{pt}(\rho) \quad C = \{\rho \mapsto \langle s(0), s(0) \rangle\} \]

\[ r_{0.1} = \text{CTag}; \]
\[ r_0 : \text{pt}(\rho) \quad C = \{\rho \mapsto \langle s(\text{CTag}), s(0) \rangle\} \]

\[ r_{0.2} = 5; \]
\[ r_0 : \text{pt}(\rho) \quad C = \{\rho \mapsto \langle s(\text{CTag}), \text{int} \rangle\} \]
\[ r_0 : \text{pt}(\rho) \quad C = \{\rho \mapsto \text{t}\} \]
Observations

- Capability rules look similar to Hoare triples
  \[ A; C \ ` e : \sigma; C' \]

  \{ 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