15–819A3 — Class Notes
Mechanical Reasoning about Low-Level Programming Languages

David Walker
Department of Computer Science
Carnegie Mellon University

March 16, 2001

Background

- Insert picture of John, Samin, Peter and Cristiano reasoning about pointers in Hoare logic.

Meanwhile, on the other side of town ...

- Dave, Fred, Greg, and Karl are running up against some of the limitations of their research on typed assembly language.
- Again and again, these limitations revolve around reasoning about state, aliasing and memory reuse.
- Fundamental question: How do we mechanically check low-level programs that destructively alter program state in the presence of aliasing?

What we did

- A type system for reasoning about region-based memory management. (with Crary, Morrisett, popl 99)
  - possible to reason about shared and unshared regions
  - bounded quantification allowed us to move between shared and unshared views
  - no reasoning about the internal structure of regions
- A type system for reasoning about aliasing between individual objects. (with Smith, Morrisett, esop 00)
  - shared and unshared objects, no bounded quantification
  - we represent the internal structure of objects.
- Add recursive and existential types. (with Morrisett, tic 00)
  - Only unshared objects.
- We have implemented some of these features in TAL.

Notice: as the structure of objects becomes more sophisticated, the logic surrounding them decreases in complexity.

In part, this is because we were missing the beautiful and simple model proposed by Reynolds & Co.

Today

Goal: take a fragment of the logic of bunched implications and its model and incorporate it into our type system.

Benefits:

- Given a simple enough fragment, mechanical checking becomes possible. The type system provides a foundation for programming language design.
- Easily handles higher-order functions.
- Highlights the role of parametric polymorphism.
- Provides intuitive understanding of the relationships between Hoare-style reasoning and advanced type systems.

A Plan

Types for reasoning about aliasing.

- the storage model
- a simple fragment of BI for reasoning about the store
- a simple higher-order imperative language
- rules for typing instructions and examples
- rules for typing unshared trees and examples
- missing: a general account of recursive types

Mechanical Reasoning: Why not use a theorem prover?

Why prefer a typed programming language over static program analysis and theorem proving technology?

To use a theorem prover we must:

- synthesize loop invariants and function specifications
  - eg: $\{p \wedge \texttt{stree}\ \tau(i)\}$ `copytree` $(i; j)$ $\{p \wedge \texttt{stree}\ \tau(i)\}$
- prove logical implications
  - eg: $\texttt{list}\ \alpha \cdot \beta(i, k) \Leftrightarrow \exists j. \texttt{list}\ \alpha(i, j) * \texttt{list}\ \beta(j, k)$

But this is intrinsically extremely hard:

- theorem proving is necessarily incomplete because our logic is undecidable.
  - what happens when the theorem prover fails?
  - does the programmer have an intuitive model that suggests when the theorem prover will succeed/fail?
- theorem proving will not scale to large programs.

Mechanical Reasoning: Type Systems

When we use a decidable type system we must reduce our expectations. We must:

- give up on verifying complete specifications. Attempt to describe the shapes of data structures rather than their contents.
- focus on detecting and eliminating a wide class of common errors
  - dereferencing null or a dangling pointer
  - leaking memory
- be prepared to reject some programs because they use operations that are simply too hard to reason about.
  - eg: programs that xor their pointers!
- aim for the common case.

Some Advantages of Typed Programming Languages

- a type system ensures basic properties about the shapes of data structures and the behavior of code.
- programs can normally be checked one module at a time and therefore, type systems normally scale very well to large programs.
- type systems will reject sound programs that are too hard to reason about.
- the goal: develop an intuitive programming model so programmers understand why programs are rejected and how to fix them: let the programmer be your theorem prover.

The Storage Model

A store is a finite partial map from *locations* ρ to stored values s.

$\{\rho_1 \mapsto s_1, \ldots, \rho_n \mapsto s_n\}$

- locations are abstract values, with no assumed relation between them.
- each stored value is a "big thing."
- for now, we will consider tuples $\langle v_1, \ldots, v_n \rangle$

Small Values

Small (register-sized) values v fit into the fields of tuples.

- values of base type: integers i, booleans, ...
- pointers ρ
- recursive functions `fix` $f\ (\cdots).e$
  - functions are considered small
  - non-recursive functions are denoted $\lambda(\cdots).e$

Some Observations

We have raised the level of abstraction considerably.

- not all values are integers.
- different operations will apply to the different categories of values.
  - eg: integers can be added to each other, but not to pointers.
  - eg: pointers can be dereferenced, code cannot
- the model precludes certain forms of reasoning.
  - impossible to view a 4-tuple as two adjacent pairs.
  - no pointers into the middle of objects.
  - no pointer arithmetic
- garbage collectors can often make use of these invariants.
  - eg: BDW conservative collector for C
  - choose your level of abstraction carefully!

Types for Values

We define a typing judgment for values: $\Delta;\Gamma \vdash v : \tau$. The context Δ is a list of the locations that may be in use. The context Γ is a mapping from term variables x to types τ.

Base types are ordinary:

- $\Delta;\Gamma \vdash i : int$

To capture the shape of data structures, we will be more precise with pointers:

- $\Delta;\Gamma \vdash \rho : \rho \qquad (\text{if } \rho \in \Delta)$

The type ρ is a *singleton type*. The only value in that type is the value ρ.

[Read more about singleton types in Xi & Pfenning's DML]

Reasoning with Singletons

A useful lemma: If $\Delta; \Gamma \vdash v : \rho$ for some value v, then v must be ρ (or a variable x).

- By analyzing the *type*, we learn very precise information about the *value*.

A corollary: If $\Delta; \Gamma \vdash v : \rho$ and $\Delta; \Gamma \vdash v' : \rho$ and v and v' are not variables then v and v' must be *aliases*.

Singleton types provide a rudimentary form of equality information.

- no need to reason with sets of equations

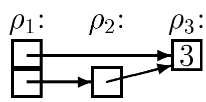
Store Types

We will equip our type system with a *decidable* logic for reasoning about the store.

We will concentrate on a very simple fragment of BI/LL

$$
\begin{array}{llll}
\textit{store types} & C & ::= & \epsilon \mid \mathbf{1} \mid C_1 * C_2 \mid \\
 & & & \texttt{true} \mid C_1 \& C_2 \mid \\
 & & & \{\rho \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}
\end{array}
$$

Example



$$
\begin{array}{l}
\{\rho_1 \mapsto \langle \rho_3, \rho_2 \rangle\} * \\
\{\rho_2 \mapsto \langle \rho_3 \rangle\} * \\
\{\rho_3 \mapsto \langle int \rangle\}
\end{array}
$$

Singleton types help specify the shape of the store.

Well-formed Stores

The judgment $\Delta \vdash h : C$ states that a store h is well-formed with type C. We specify a nondeterministic merge of two stores h_1 and h_2 using the notation $h_1 \bowtie h_2$. It requires that the domains of the stores h_1 and h_2 be disjoint.

$$\frac{}{\Delta \vdash \{\,\} : \mathbf{1}}$$

$$\frac{\Delta \vdash h_1 : C_1 \quad \Delta \vdash h_2 : C_2}{\Delta \vdash h_1 \bowtie h_2 : C_1 * C_2}$$

$$\frac{}{\Delta \vdash h : \texttt{true}}$$

$$\frac{\Delta \vdash h : C_1 \quad \Delta \vdash h : C_2}{\Delta \vdash h : C_1 \& C_2}$$

$$\frac{\Delta; \cdot \vdash v_i : \tau_i \quad \text{for } 1 \leq i \leq n}{\Delta \vdash \{\rho \mapsto \langle v_1, \ldots, v_n \rangle\} : \{\rho \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}}$$

The Logic

In order to show that a program manipulates the store properly, the type-checker will have to prove judgments in this logic.

Judgments have the form:

$$\Lambda \vdash C$$

Where $\Lambda \ ::= \ \cdot \mid \Lambda, C$.

We often omit the initial "$\cdot$". For example, when we have a single hypothesis we write:

$$C_1 \vdash C_2$$

As before, we use the notation $\Lambda_1 \bowtie \Lambda_2$ to indicate a nondeterministic merge of the two contexts. It contains all of the formulas in Λ_1 and Λ_2 (each formula appearing exactly once) interleaved in an arbitrary order.

Sequent Calculus for $(\mathbf{1}, *, \mathtt{true}, \&)$

$$\frac{}{C \vdash C}$$

$$\frac{}{\cdot \vdash \mathbf{1}}$$

$$\frac{\Lambda_1, \Lambda_2 \vdash C}{\Lambda_1, \mathbf{1}, \Lambda_2 \vdash C}$$

$$\frac{\Lambda_1 \vdash C_1 \quad \Lambda_2 \vdash C_2}{\Lambda_1 \bowtie \Lambda_2 \vdash C_1 * C_2}$$

$$\frac{\Lambda_1, C_1, C_2, \Lambda_2 \vdash C}{\Lambda_1, C_1 * C_2, \Lambda_2 \vdash C}$$

$$\frac{}{\Lambda \vdash \mathtt{true}}$$

$$\frac{\Lambda \vdash C_1 \quad \Lambda \vdash C_2}{\Lambda \vdash C_1 \& C_2}$$

$$\frac{\Lambda_1, C_1, \Lambda_2 \vdash C_3}{\Lambda_1, C_1 \& C_2, \Lambda_2 \vdash C_3}$$

$$\frac{\Lambda_1, C_2, \Lambda_2 \vdash C_3}{\Lambda_1, C_1 \& C_2, \Lambda_2 \vdash C_3}$$

$$\frac{\vdash \tau_i = \tau_i' \qquad \text{for } 1 \leq i \leq n}{\{\rho \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \vdash \{\rho \mapsto \langle \tau_1', \ldots, \tau_n' \rangle\}}$$

Properties of the Logic

Soundness:

- If $\Delta \vdash h : C_1$ and $C_1 \vdash C_2$ then $\Delta \vdash h : C_2$.
- More generally, if $\Delta \vdash h_i : C_i$ for $i \in 1..n$, $h_i \bot h_j$ whenever $i \neq j$, and $C_1, \ldots, C_n \vdash C$ then $\Delta \vdash \bigcup_{i \in 1..n} h_i : C$.

Decideability:

- Given C_1 and C_2, it is possible to decide whether or not $C_1 \vdash C_2$.

Computability of the residual store type:

- To develop the type system, we also require a procedure which given C_1 and ρ, computes a residual store type C_2 and types $\tau_1, \ldots, \tau_n$ such that $C_1 \vdash C_2 * \{\rho \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}$.
- In general, we cannot necessarily compute a "best" store type C_2, but we can compute a finite set of formulae and try each one in the remaining typing derivation. There is an algorithm that generates a "best" store type for all examples in this talk and it will succeed whenever the region in question appears at most once on the left of a points-to proposition in the formula in question.

So Far

- types for small values (integers, pointers)
- types (specifications) for heaps
- a decidable logic for mechanical reasoning about heap types

Next: the programming language and its type checking rules

Continuation Passing Style

The language we will study requires programs be written in *continuation-passing style* (CPS).

CPS functions have the following structure:

- a linear sequence of primitive operations (eg: allocate, dispose, dereference, update, etc.)
- terminated by a control-flow transfer
  - a jump to another function
  - a branching construct (if or case statement)
  - a special `halt` instruction

CPS functions *do not* have conventional call and return operations.

To mimic call and return, we jump to the function, explicitly passing the return address as an argument.

An Example

The factorial function in CPS. $fact(n) = n \times (n-1) \times \cdots \times 1$

$$
\begin{array}{l}
\texttt{fix}\ fact\ (n,\ k). \\
\quad\quad \texttt{if}\ n = 0\ \texttt{then} \\
\quad\quad\quad\quad k(1) \\
\quad\quad \texttt{else} \\
\quad\quad\quad\quad \texttt{let}\ x = n - 1\ \texttt{in} \\
\quad\quad\quad\quad \texttt{let}\ cont = \lambda(y).\,\texttt{let}\ z = n \times y\ \texttt{in}\ k(z)\ \texttt{in} \\
\quad\quad\quad\quad fact(x, cont)
\end{array}
$$

To use the factorial function, we apply fact to an argument and an initial continuation.

$$fact(6, \lambda(n).\,\texttt{halt}\ n)$$

Note: we will disregard space required for closures in this lecture.

Types for CPS Functions

Every CPS function specifies the type of the store that it expects:

$$(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}$$

Think of the store as an extra argument to every function.

Example: a dereference function

$$(\{\rho \mapsto \langle\tau\rangle\}, \rho, \tau_{cont}) \rightarrow \mathbf{0}$$

Where $\tau_{cont} = (\{\rho \mapsto \langle\tau\rangle\}, \tau) \rightarrow \mathbf{0}$

Before calling a function, it is necessary to satisfy its typing precondition.

Example: before calling the dereference function, we must prove the store has the shape $\{\rho \mapsto \langle\tau\rangle\}$.

Polymorphism

Problem: the dereference function can only operate on one specific location (ρ).

Location polymorphism makes it possible to abstract away from the specifics of the particular location.

Example: the dereference function

$$\forall[\rho].(\{\rho \mapsto \langle\tau\rangle\}, \rho, \tau_{cont}) \rightarrow \mathbf{0}$$

Where $\tau_{cont} = (\{\rho \mapsto \langle\tau\rangle\}, \tau) \rightarrow \mathbf{0}$

Before calling the dereference function, we must:

1. instantiate the bound type variables
   - eg: $\texttt{deref}[\rho'] : (\{\rho' \mapsto \langle\tau\rangle\}, \rho', \ldots) \rightarrow \mathbf{0}$
2. prove the new typing precondition
   - eg: prove the current store satisfies $\{\rho' \mapsto \langle\tau\rangle\}$

More polymorphism

Store polymorphism makes it possible to preserve portions of the store across function calls.

Example: the dereference function

$$\forall[\epsilon, \rho].(\epsilon * \{\rho \mapsto \langle\tau\rangle\}, \rho, \tau_{cont}) \to \mathbf{0}$$

Where $\tau_{cont} = (\epsilon * \{\rho \mapsto \langle\tau\rangle\}, \tau) \to \mathbf{0}$

The variable ϵ may be instantiated with any store type.

Store Polymorphism vs. The Frame Axiom

Recall: the frame axiom

$$\frac{\{p\}f\{q\}}{\{p * r\}f\{q * r\}}$$

In our polymorphic, continuation passing style:

$$f : \forall[\epsilon].(p * \epsilon, (q * \epsilon) \to \mathbf{0}) \to \mathbf{0}$$

To apply the frame axiom, instantiate the polymorphic variable ϵ.

$$f[r] : (p * r, (q * r) \to \mathbf{0}) \to \mathbf{0}$$

Store polymorphism appears essential for higher-order programs and for preservation of store shape in other circumstances (see example later).

A Problem?

A potential problem:

$$\texttt{deref} : \forall[\epsilon, \rho].(\epsilon * \{\rho \mapsto \langle\tau\rangle\}, \ldots) \rightarrow \mathbf{0}$$

Apply `deref` to the "wrong" store type:

$$\texttt{deref}[\{\rho' \mapsto \langle\tau\rangle\}, \rho'] : (\{\rho' \mapsto \langle\tau\rangle\} * \{\rho' \mapsto \langle\tau\rangle\}, \ldots) \rightarrow \mathbf{0}$$

We have created a "bad" store type. No store can be described by this type.

That is okay!

- We will never be able to prove that the current store has that shape and we will never be able to call the function.
- The instantiated function is dead code.

Term Syntax

$$
\begin{array}{llcl}
\textit{type contexts} & \Delta & ::= & \cdot \mid \Delta, \epsilon \mid \Delta, \rho \\
\textit{store types} & C & ::= & \cdots \\
\textit{types} & \tau & ::= & \cdots \\
\textit{location vars} & \rho & & \\
 & c & ::= & C \mid \tau \mid \rho \\
 & & & \\
\textit{small values} & v & ::= & x \mid i \mid v[c] \mid \\
 & & & \texttt{fix}\ f\ [\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \\
 & & & \\
\textit{primitive operators} & \bullet & ::= & + \mid \times \mid \div \mid \cdots \\
 & & & \\
\textit{instructions} & e & ::= & \texttt{let}\, x = v_1 \bullet v_2\, \texttt{in}\, e \\
 & & & \texttt{let}\, \rho, x = \texttt{new}\ (i)\ \texttt{in}\ \ e \mid \\
 & & & \texttt{dispose}\ v; e \mid \\
 & & & \texttt{let}\ \ x = v.i\, \texttt{in}\, e \mid \\
 & & & v_1.i{:=}v_2; e \mid \\
 & & & \texttt{if}\ v = 0\ \texttt{else}\ e_1\ \texttt{then}\ e_2 \mid \\
 & & & v(v_1, \ldots, v_n) \mid \\
 & & & \texttt{halt}\ v
\end{array}
$$

Typing Rules for Instructions

The typing rules have the form: $\Delta; C; \Gamma \vdash e$ where:

- the free type, location and store variables in C, Γ and e are contained in Δ.
- C describes the store before executing e
- Γ describes the types of free value variables in e
- Assume that all variables in Γ or Δ are distinct (alpha-vary bound variables where necessary)

These rules rely on the judgment for well-formed types $\Delta \vdash c$, which states that $\mathrm{FV}(c) \subseteq \Delta$.

Store Invariant Rules

The rule for `let` :

$$\frac{\Delta; \Gamma \vdash v : \tau \quad \Delta; C; \Gamma, x{:}\tau \vdash e}{\Delta; C; \Gamma \vdash \texttt{let}\, x = v\, \texttt{in}\, e}$$

The rule for primitives:

$$\frac{\Delta; \Gamma \vdash v_1 : int \quad \Delta; \Gamma \vdash v_2 : int \quad \Delta; C; \Gamma, x{:}int \vdash e}{\Delta; C; \Gamma \vdash \texttt{let}\, x = v_1 \bullet v_2\, \texttt{in}\, e}$$

The rule for if:

$$\frac{\Delta; \Gamma \vdash v : int \quad \Delta; C; \Gamma \vdash e_1 \quad \Delta; C; \Gamma \vdash e_2}{\Delta; C; \Gamma \vdash \texttt{if}\ v = 0\ \texttt{else}\ e_1\ \texttt{then}\ e_2}$$

New

$$\frac{\Delta, \rho; C * \{\rho \mapsto \overbrace{\langle junk, \ldots, junk \rangle}^{i}\}; \Gamma, x{:}\rho \vdash e}{\Delta; C; \Gamma \vdash \texttt{let}\, \rho, x = \texttt{new}\ (i)\ \texttt{in}\ \ e}$$

The Hoare rule:

C
$x := \texttt{cons}(junk, \ldots, junk)$
$C * \{x \mapsto \langle junk, \ldots, junk \rangle\}$

where x is fresh

Dispose

$$\frac{\Delta; \Gamma \vdash v : \rho \quad C \vdash C' * \{\rho \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} \quad \Delta; C'; \Gamma \vdash e}{\Delta; C; \Gamma \vdash \texttt{dispose}\ v; e}$$

The Hoare rule for single-cell dispose:

$$\begin{array}{l} C' * \{e \mapsto -\} \\ \texttt{dispose}\ e \\ C' \end{array}$$

Notice we have built in the rule for "strengthening the precedent"

$$\frac{p \Rightarrow q \quad \{q\}c\{r\}}{\{p\}c\{r\}}$$

The Hoare rule for multiple-cell dispose:

$$\begin{array}{l} C' * \{e \mapsto -\} * \cdots * \{e + (n-1) \mapsto -\} \\ \texttt{dispose}\ e, n \\ C' \end{array}$$

Lookup

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \rho \\ C \vdash \{\rho \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} * \mathtt{true} \qquad \Delta; C; \Gamma, x{:}\tau_i \vdash e\end{array}}{\Delta; C; \Gamma \vdash \mathtt{let}\ x = v.i\ \mathtt{in}\, e}$$

The Hoare rule:

$$\begin{array}{l}\exists y.C[y/x]\ \&\ \{e + (i-1) \hookrightarrow y\} \quad \equiv \\ \exists y.C[y/x]\ \&\ (\{e + (i-1) \mapsto y\} * \mathtt{true}) \\ x := [e + (i-1)] \\ C\end{array}$$

Update

$$\frac{\begin{array}{cc} \Delta;\Gamma \vdash v_1 : \rho & C \vdash C' * \{\rho \mapsto \langle \tau_1, \ldots, \tau_i, \ldots, \tau_n \rangle\} \\ \Delta;\Gamma \vdash v_2 : \tau & \Delta; C' * \{\rho \mapsto \langle \tau_1, \ldots, \tau, \ldots, \tau_n \rangle\}; \Gamma \vdash e \end{array}}{\Delta; C; \Gamma \vdash v_1.i{:=}v_2; e}$$

The Hoare rule:

$$\begin{array}{l} C' * \{e + (i-1) \mapsto -\} \\ e + (i-1) := e' \\ C' * \{e + (i-1) \mapsto e'\} \end{array}$$

Specification-Invariant Update Rule

Another update rule (specification invariant):

$$\frac{\Delta;\Gamma \vdash v_1 : \rho \qquad C \vdash \{\rho \mapsto \langle \tau_1, \ldots, \tau_i, \ldots, \tau_n \rangle\} * \texttt{true} \qquad \Delta;\Gamma \vdash v_2 : \tau_i \qquad \Delta; C;\Gamma \vdash e}{\Delta; C; \Gamma \vdash v_1.i{:=}v_2; e}$$

Be careful: unsound in the presence of subtyping.

Is this rule an instance of the more general rule for update? No:

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v_1 : \rho \\ (\{\rho \mapsto \langle \tau \rangle\} * C_1) \ \&\ C_2 \vdash \{\rho \mapsto \langle \tau \rangle\} * \texttt{true} \\ \Delta;\Gamma \vdash v_2 : \tau \qquad \Delta; (\{\rho \mapsto \langle \tau \rangle\} * C_1) \ \&\ C_2; \Gamma \vdash e\end{array}}{\Delta; (\{\rho \mapsto \langle \tau \rangle\} * C_1) \ \&\ C_2; \Gamma \vdash v_1.1{:=}v_2; e}$$

In general, we will lose the information contained in C_2 if we use the previous update rule:

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v_1 : \rho \\ (\{\rho \mapsto \langle \tau \rangle\} * C_1) \ \&\ C_2 \vdash C_1 * \{\rho \mapsto \langle \tau \rangle\} \\ \Delta;\Gamma \vdash v_2 : \tau \qquad \Delta; C_1 * \{\rho \mapsto \langle \tau \rangle\}; \Gamma \vdash e\end{array}}{\Delta; (\{\rho \mapsto \langle \tau \rangle\} * C_1) \ \&\ C_2; \Gamma \vdash v_1.1{:=}v_2; e}$$

The Hoare rule?

Function Application and Termination

Function application

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \forall[\cdot].(C', \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \\ \Delta;\Gamma \vdash v_1 : \tau_1 \qquad \cdots \qquad \Delta;\Gamma \vdash v_n : \tau_n \qquad C \vdash C'\end{array}}{\Delta; C; \Gamma \vdash v(v_1, \ldots, v_n)}$$

Termination For "tight" specifications that require programs to collect their garbage:

$$\frac{\Delta;\Gamma \vdash v : int \quad C \vdash \mathbf{1}}{\Delta; C; \Gamma \vdash \texttt{halt}\ v}$$

For "loose" specifications that leave garbage lying around:

$$\frac{\Delta;\Gamma \vdash v : int \quad C \vdash \texttt{true}}{\Delta; C; \Gamma \vdash \texttt{halt}\ v}$$

Functions

The typing judgment for (possibly open) values has the form $\Delta; \Gamma \vdash v : \tau$

For function values:

$$\frac{\begin{array}{c}\Delta, \Delta'; C; \Gamma, f : \forall[\Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash e \\ \Delta \vdash \forall[\Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}\end{array}}{\Delta; \Gamma \vdash \mathtt{fix} f[\Delta'](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \forall[\Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}}$$

Type application:

$$\frac{\Delta; C; \Gamma \vdash v : \forall[\rho', \Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \quad \Delta \vdash \rho}{\Delta; C; \Gamma \vdash v[\rho] : \forall[\Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}[\rho/\rho']}$$

$$\frac{\Delta; C; \Gamma \vdash v : \forall[\epsilon, \Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \quad \Delta \vdash C'}{\Delta; C; \Gamma \vdash v[C'] : \forall[\Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}[C'/\epsilon]}$$

Example

Arguments: two references to integers.

Add contents of arguments, store result in first argument.

Preserve the structure of the store

```
let add = λ[ε, ρ1, ρ2]
          (C, x:ρ1, y:ρ2, k:(ε) → 0).
          let z = x.1 in
          let w = y.1 in
          let a = z + w in
          x.1 := a;            % by invariant update
          k( )                 % C ⊢ ε, by &-left, id
in ···
```

where $C = \epsilon \ \& \ (\{\rho_1 \mapsto \langle int \rangle\} * \texttt{true}) \ \& \ (\{\rho_2 \mapsto \langle int \rangle\} * \texttt{true})$

Example Continued

Using the add function:

$$
\begin{array}{ll}
\texttt{let}\ \rho_1, x = \texttt{new}(1)\ \texttt{in} & \texttt{\%}\ \{\rho_1 \mapsto \langle junk \rangle\} \\
x.1 := 17; & \texttt{\%}\ \{\rho_1 \mapsto \langle int \rangle\} \\
\texttt{let}\ \rho_2, y = \texttt{new}(1)\ \texttt{in} & \texttt{\%}\ \{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle junk \rangle\} \\
y.1 := 32; & \texttt{\%}\ \{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle int \rangle\} \\
\texttt{let}\ k = \lambda[\,] & \\
\quad\quad (\{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle int \rangle\}) & \\
\quad\quad \texttt{let}\ a = x.1\ \texttt{in} & \\
\quad\quad \texttt{dispose}(x); & \texttt{\%}\ \{\rho_2 \mapsto \langle int \rangle\} \\
\quad\quad \texttt{dispose}(y); & \texttt{\%}\ \mathbf{1} \quad \texttt{(*)} \\
\quad\quad \texttt{halt}\ a & \\
\texttt{in} & \\
add[\{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle int \rangle\}, \rho_1, \rho_2](x, y, k) & \quad \texttt{(**)}
\end{array}
$$

The dispose operation:

$$
\frac{\cdot \vdash \mathbf{1} \qquad \{\rho_2 \mapsto \langle int \rangle\} \vdash \{\rho_2 \mapsto \langle int \rangle\}}{\{\rho_2 \mapsto \langle int \rangle\} \vdash \mathbf{1} * \{\rho_2 \mapsto \langle int \rangle\}}\ (*)
$$

Justification for (**)

The function application:

$$add : \forall[\epsilon, \rho_1, \rho_2].(C, \rho_1, \rho_2, (\epsilon) \to \mathbf{0})) \to \mathbf{0}$$

where $C = \epsilon \mathbin{\&} (\{\rho_1 \mapsto \langle int \rangle\} * \texttt{true}) \mathbin{\&} (\{\rho_2 \mapsto \langle int \rangle\} * \texttt{true})$

$$add[C', \rho_1, \rho_2] : (C[C'/\epsilon], \rho_1, \rho_2, (C') \to \mathbf{0})) \to \mathbf{0}$$

where $C' = \{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle int \rangle\}$

Must prove that the current store C' satisfies the precondition for $add[\cdots]$:

$$\frac{\vdots}{\begin{array}{l}\{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle int \rangle\} \vdash \\ (\{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle int \rangle\}) \mathbin{\&} (\{\rho_1 \mapsto \langle int \rangle\} * \texttt{true}) \mathbin{\&} (\{\rho_2 \mapsto \langle int \rangle\} * \texttt{true})\end{array}}$$

Which reduces to proving the three conjuncts separately (by &-right).

$$\frac{}{\{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle int \rangle\} \vdash (\{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle int \rangle\})}$$

$$\dfrac{\dfrac{\{\rho_1 \mapsto \langle int \rangle\} \vdash \{\rho_1 \mapsto \langle int \rangle\} \quad \{\rho_2 \mapsto \langle int \rangle\} \vdash \texttt{true}}{\{\rho_1 \mapsto \langle int \rangle\}, \{\rho_2 \mapsto \langle int \rangle\} \vdash \{\rho_1 \mapsto \langle int \rangle\} * \texttt{true}}}{\{\rho_1 \mapsto \langle int \rangle\} * \{\rho_2 \mapsto \langle int \rangle\} \vdash \{\rho_1 \mapsto \langle int \rangle\} * \texttt{true}}$$

The last conjunct is similar.

Example Continued

Using the add function:

$\texttt{let}\ \rho_1, x = \texttt{new}(1)\ \texttt{in}$ % $\{\rho_1 \mapsto \langle junk \rangle\}$
$x.1 := 17;$ % $\{\rho_1 \mapsto \langle int \rangle\}$
$\texttt{let}\ k = \lambda[\,]$
    $(\{\rho_1 \mapsto \langle int \rangle\})$
    $\texttt{let}\ a = x.1\ \texttt{in}$
    $\texttt{dispose}(x);$ **% 1**
    $\texttt{halt}\ a$
$\texttt{in}$
$add[\{\rho_1 \mapsto \langle int \rangle\}, \rho_1, \rho_1](x, x, k)$ `(**)`

We must prove the following at the function application `(**)`.

$$\frac{\vdots}{\begin{array}{l}\{\rho_1 \mapsto \langle int \rangle\} \vdash \\ \{\rho_1 \mapsto \langle int \rangle\} \& (\{\rho_1 \mapsto \langle int \rangle\} * \texttt{true}) \& (\{\rho_1 \mapsto \langle int \rangle\} * \texttt{true})\end{array}}$$

Main points:

- *add* is sharing insensitive
- *add* preserves store shape exactly
- store polymorphism is essential

An Aside

In previous work, I (with Crary and Morrisett, popl 99) solved the same problem with a slightly different logic and bounded polymorphism:

$$\forall[\epsilon \leq C].\tau$$

Recently, Crary (icfp 00) has shown that bounded polymorphism can be compiled into a language with ordinary parametric polymorphism and intersection types:

$$\forall[\epsilon \leq C].\tau \leadsto \forall[\epsilon].\tau[C\&\epsilon/\epsilon]$$

Our add function in previous work:

$$\forall[\epsilon \leq C, \ldots].(\epsilon, \ldots) \rightarrow \mathbf{0}$$

Our add function today:

$$\forall[\epsilon, \ldots].(C\&\epsilon, \ldots) \rightarrow \mathbf{0}$$

I had hoped that this insight would lead to a formal connection between our "capability logic" and this fragment of BI. Still working on it.

Unshared Trees

We introduce an abstract type for unshared trees.

$$
\begin{array}{llll}
\textit{store types} & C & ::= & \cdots \mid \texttt{tree}(\rho) \\
\textit{types} & \tau & ::= & \cdots \mid \texttt{L} \mid \texttt{N} \\
\textit{values} & v & ::= & \cdots \mid \texttt{L} \mid \texttt{N} \\
\textit{instructions} & e & ::= & \cdots \mid \texttt{case}\ v\ (\texttt{L}\ \Rightarrow e_1 \mid \texttt{N}(\rho_1, \rho_2)\ \Rightarrow e_2)
\end{array}
$$

We give the atoms N and L singleton types.

$$\frac{}{\Delta; \Gamma \vdash \texttt{L} : \texttt{L}}$$

$$\frac{}{\Delta; \Gamma \vdash \texttt{N} : \texttt{N}}$$

Whenever we need to express some data dependence, we use singleton types.

Store Typing for Trees

In addition to previous rules, we have:

$$\frac{}{\Delta \vdash \{\rho \mapsto \langle \mathtt{L} \rangle\} : \mathtt{tree}(\rho)}$$

$$\frac{\begin{array}{c}\vdash h_1 : \mathtt{tree}(\rho_1) \qquad \vdash h_2 : \mathtt{tree}(\rho_2) \\ \vdash h_3 : \{\rho \mapsto \langle \mathtt{N}, \rho_1, \rho_2 \rangle\}\end{array}}{\Delta \vdash h_1 \bowtie h_2 \bowtie h_3 : \mathtt{tree}(\rho)}$$

A leaf node can be given two types: $\mathtt{tree}(\rho)$ or $\{\rho \mapsto \langle \mathtt{L} \rangle\}$.

Therefore, we extend the logic:

$$\frac{\Lambda \vdash \{\rho \mapsto \langle \mathtt{L} \rangle\}}{\Lambda \vdash \mathtt{tree}(\rho)}$$

$$\frac{\Lambda \vdash \{\rho \mapsto \langle \mathtt{N}, \rho_1, \rho_2 \rangle\} * \mathtt{tree}(\rho_1) * \mathtt{tree}(\rho_2)}{\Lambda \vdash \mathtt{tree}(\rho)}$$

The logic remains easily decidable.

Case Statement

The elimination form for trees is a case on the first component.

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \rho' \qquad C \vdash \texttt{tree}(\rho) * \texttt{true} \\ \Delta; C[\{\rho' \mapsto \langle \texttt{L} \rangle\}]; \Gamma \vdash e_1 \\ \Delta, \rho_1, \rho_2; C[\{\rho' \mapsto \langle \texttt{N}, \rho_1, \rho_2 \rangle\} * \texttt{tree}(\rho_1) * \texttt{tree}(\rho_2)]; \Gamma \vdash e_2\end{array}}{\Delta; C; \Gamma \vdash \texttt{case}\ v\ (\texttt{L} \ \Rightarrow e_1 \mid \texttt{N}(\rho_1, \rho_2) \ \Rightarrow e_2)}$$

The notation $C[\{\rho \mapsto \langle \cdots \rangle\}]$ replaces all occurrences of $\texttt{tree}(\rho)$ in C with $\{\rho \mapsto \langle \cdots \rangle\}$.
Unlike the case in ML, this case does not automatically project the two components of the node in the second branch.

In-place Tree Traversal

It is possible to traverse a tree without using a stack by reusing the tree nodes to store control information. (Deutsch, Schorr, Waite)

This technique was initially used in the mark phase of a garbage collector.

Traversal overview:

1. Begin with a tree T, a pointer to T's parent and a continuation.

2. Store the pointer to T's parent in the position normally used for T's left subtree. Store the continuation in the position normally used for the tag. Traverse the left subtree.

3. Restore the left subtree. Store the pointer to T's parent in the position normally used for T's right subtree. Traverse the right subtree.

4. Restore the right subtree and the tag. Call the initial continuation.

In-place Tree Traversal

```
% Traverse a tree ρ1 with parent ρ2
```

$\texttt{fix}\, walk[\epsilon, \rho_1, \rho_2 \mid \epsilon * \texttt{tree}(\rho_1)](t : \rho_1, up : \rho_2, cont : \tau_c[\epsilon, \rho_1, \rho_2]).$

$\quad \texttt{case}\, t,$

$\qquad (\ \texttt{L} \Longrightarrow cont(t, up)$

$\qquad \mid \texttt{N}(\rho_L, \rho_R) \ \Longrightarrow$

$\qquad\quad$ `%` $\epsilon * \{\rho_1 \mapsto \langle \texttt{N}, \rho_L, \rho_R \rangle\} * \texttt{tree}(\rho_L) * \texttt{tree}(\rho_R)$

$\qquad\quad t.1{:=}cont;$ `% store` $cont$ `in tag position`

$\qquad\quad \texttt{let}\ left = t.2\, \texttt{in}$

$\qquad\quad t.2{:=}up;$ `% store parent in left subtree`

$\qquad\quad$ `%` $\epsilon * \{\rho_1 \mapsto \langle \tau_c[\epsilon, \rho_1, \rho_2], \rho_2, \rho_R \rangle\} * \texttt{tree}(\rho_L) * \texttt{tree}(\rho_R)$

$\qquad\quad walk[\epsilon * \{\rho_1 \mapsto \langle \tau_c[\epsilon, \rho_1, \rho_2], \rho_2, \rho_R \rangle\} * \texttt{tree}(\rho_R), \rho_L, \rho_1]$

$\qquad\qquad (left, t, rwalk[\epsilon, \rho_1, \rho_2, \rho_L, \rho_R]))$

$\quad$ where $\tau_c[\epsilon, \rho_1, \rho_2] =$

$\qquad \forall[\cdot].(\epsilon * \texttt{tree}(\rho_1), \rho_1, \rho_2) \to \mathbf{0}$

Traversal Continued

```
% Walk the right-hand subtree ρ_R
```

and $rwalk[\epsilon, \rho_1, \rho_2, \rho_L, \rho_R \mid \epsilon * \{\rho_1 \mapsto \langle \tau_c[\epsilon, \rho_1, \rho_2], \rho_2, \rho_R\rangle\} *$
$\quad\quad \mathtt{tree}(\rho_L) * \mathtt{tree}(\rho_R)]$
$\quad\quad (left : \rho_L, t : \rho_1).$
$\quad$ let $up = t.2$ in
$\quad t.2{:=}left;$ `% restore left subtree`
$\quad$ let $right = t.3$ in
$\quad t.3{:=}up;$ `% store parent in right subtree`
$\quad walk[\epsilon * \{\rho_1 \mapsto \langle \tau_c[\epsilon, \rho_1, \rho_2], \rho_L, \rho_2\rangle\} * \mathtt{tree}(\rho_L), \rho_R, \rho_1]$
$\quad\quad (right, t, finish[\epsilon, \rho_1, \rho_2, \rho_L, \rho_R])$

where $\tau_c[\epsilon, \rho_1, \rho_2] =$
$$\forall[\cdot].(\epsilon * \mathtt{tree}(\rho_1), \rho_1, \rho_2) \to \mathbf{0}$$

Traversal Continued

```
% Reconstruct tree node and return
```

and $finish[\epsilon, \rho_1, \rho_2, \rho_L, \rho_R \mid \epsilon * \{\rho_1 \mapsto \langle \tau_c[\epsilon, \rho_1, \rho_2], \rho_L, \rho_2 \rangle\}$
$\quad\quad \texttt{tree}(\rho_L) * \texttt{tree}(\rho_R)]$
$\quad\quad (right : \rho_R, t : \rho_1).$
$\quad$ let $up = t.3$ in
$\quad t.3{:=}right;$ `% restore right subtree`
$\quad$ let $cont = t.1$ in
$\quad t.1{:=}\texttt{N};$ `% restore tag`
$\quad cont(t, up)$

where $\tau_c[\epsilon, \rho_1, \rho_2] =$
$\quad \forall[\cdot].(\epsilon * \texttt{tree}(\rho_1), \rho_1, \rho_2) \to \mathbf{0}$

General Recursive Types

For an explanation of how to define trees and other recursive datatypes in the multiplicative fragment of the logic, see my paper with Morrisett.

One difference between the Reynolds/O'Hearn approach and our approach:

- Reynolds and O'Hearn use induction over the structure of the store: $\mu\epsilon.C$
- We use induction over types: $\mu\alpha.\tau$.

Our technique requires that we find a way to include specifications of the store within a type τ.

- Can be achieved through a special form of existential type.
- These ideas led to techniques for combining regions with linear data structures in a recent paper with Kevin Watkins.

Conclusions and Future Work

- Integration of substructural logics with dependent typing is a rich area for further research.
- This simple fragment of BI (multiplicative and additive conjunctions and units) seems like it has great potential in the domain of memory management.
- To incorporate this type system into a practical source programming language, we will need to resolve many issues with respect to syntax and type inference.
- In may be possible to use this framework to control other program effects including concurrency, access control and security-relevant effects.