CS 565

Programming Languages
Spring 2025

Week 15

Course Review

Functional Programming

We'll start our investigation by considering a small functional language

- These languages tend to have a small core set of features
- Datatypes, functions, and their application
- Written in Gallina, the specification and programming language for Coq

```
Definition double (n : nat) : nat := n + n.
```

Week 1

Programming in Gallina

Functions

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume values, produce values

```
Definition double (n : nat) : nat := n + n.

Eval compute in (double 1). (* = 2 *)
```

Compound ADTs

- Can build new ADTs from existing ones:
  - A color is either black, white, or a primary color
  - Need to apply primary to something of type rgb
- ADTs are **algebraic** because they are built from a small set of operators (sums of product).

```
Inductive rgb : Type := | red | green | blue.

Inductive color := | black | white
  | primary (p : rgb).

Eval compute in (primary red). (* = primary red *)
```

Week 2

Induction

Nat Induction

Mathematical Induction for Natural Numbers:

For any predicate P on natural numbers, **if:**

1. P(0)
2. P(n) implies P(n+1)

Then:

for all n, P(n) holds.

Tree Induction

Works for trees too:

> For any number n, and tree t
> element (insert t n) n = true.

Proof: By induction on t.

Next, suppose t = node n' lt rt, where

element (insert lt n) n = true and element (insert rt n) n = true.

Induction Hypothesis

We must show: element (insert (node n' lt rt) n) n = true.

By definition, this is equivalent to:

element (**if** (cmp n n') **then** node n' (insert cmp lt n) rt
else node y lt (insert cmp rt n))

- Consider the case when cmp n n′ = true.

  We must show: element (node n' (insert cmp lt n) rt) n = true.

  This follows from the IH.

- Consider the case when cmp n n′ = false.

  We must show: element (node n' lt (insert cmp rt n)) n = true.

  This follows from the IH.

Week 3

Functional Programming and Polymorphism

Total Maps

Standard operations: higher-order functions:

```
Definition map : Type := string -> nat.

Definition lookup (m : map) (x : string) : nat := m x.
Definition empty : map := fun x => 0.

Definition update (m : map) (x : string) (v : nat) : map :=
  fun y => if (eqb_string x y) then v else m y.


Definition example : map := update (update empty “x” 1 ) “y” 2.
```

What is the behavior of m?

```
Definition m : map :=
      update (update (fun y => 42) “x” 7 ) “z” 10.
```

Generic Lists

Coq supports **type abstraction** in data type declarations via **type parameters:**

```
Inductive list (X : Type) : Type :=
  | nil
  | cons (x : X) (l : list X).
```

list is a function from types to types:

```
Check list.  (* : Type -> Type *)
```

Week 4

Inductive Propositions

Propositions

A **proposition** is a factual claim.

Have seen a couple of propositions (in Coq) so far:

- equalities: 0 + n = n
- implications: P -> Q
- universally quantified propositions: forall x, P

A **proof** is some evidence for the truth of a proposition

A **proof system** is a formalization of particular kinds of evidence.

Propositions

Can have polymorphic predicates:

```
Definition injective {A B} (f : A -> B) : Prop :=
  forall x y : A, f x = f y -> x = y.
Theorem plus1_inj : injective (plus 1).
Proof.
… (* unfold injective *)
```

Equality is a polymorphic binary predicate:

```
Check @eq. (* : ∀ A : Type, A → A → Prop *)
```

Judgement

A **judgement** is a claim of a proof system

The judgement $\Gamma \vdash A$ is read as:
"assuming the propositions in Γ are true, A is true".

We'll see other judgements over the course of the semester:

Inference Rules

Proof systems construct evidence of judgements via inference rules:

Axioms

$$\frac{}{\Gamma \vdash T}$$

$$\frac{A \in \Gamma}{\Gamma \vdash A}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \mathbf{I}\!\to$$

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \mathbf{E}\!\to$$

Inference Rules

Inductively Defined Propositions

- Goal:

  N-ary relation on natural numbers

  Form of evidence of membership in that relation
- Step 0: Name the ~~relation~~ type:
- Step 1: Give the ~~relation~~ type a ~~signature~~ type:
- Step 2: Enumerate ~~evidence~~ constructors:

```
Inductive even : nat -> Prop :=
  | ev_O : even O
  | even_2 : forall n : nat, even n -> even (S (S n)).
```

Week 5

Curry-Howard Isomorphism

Observation

Two ways of thinking about $\rightarrow$:

- As a type constructor:

  f: $A \rightarrow B$ denotes the type of a function that transforms elements of A into elements of B

- As a logical implication:

  $A \rightarrow B$ establishes the validity of proposition B given the validity of proposition A

How are these notions related?

Observation

They are exactly the same!

Logical implication models the type of functions that transforms evidence (aka proofs):

$A \rightarrow B$ represents the type of all functions that given evidence for the validity of A, returns a proof (aka evidence) for the validity of B

As a Relation

Key Idea: Define evaluation as a Inductive Relation

aevalR: total_map → A → ℕ → Proposition

- ★ Ternary relation on states, expressions and values
- ★ Read ‘σ, a ⇓ n’ as ‘a evaluates to n in state σ’
- ★ Relation precisely spells out what values program can evaluate to
- ★ Put another way, rules define an ‘abstract machine’ for executing expression

Week 6

Big-Step Semantics and IMP

Semantics

cevalR: (Id → ℕ) → C → (Id → ℕ) → Proposition

- Ternary relation on initial states, commands and final state
- Read ‘σ, c ⇓ σ’ as ‘when run in initial state σ, c produces (i.e. evaluates to) final state σ’

Semantics

Inference Rules for ⇓ (commands)

EWHILET

$$\frac{\sigma_1, b \Downarrow \textbf{true} \qquad \sigma_1, c \Downarrow \sigma_2 \qquad \sigma_2, \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end} \Downarrow \sigma_3}{\sigma_1, \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end} \Downarrow \sigma_3}$$

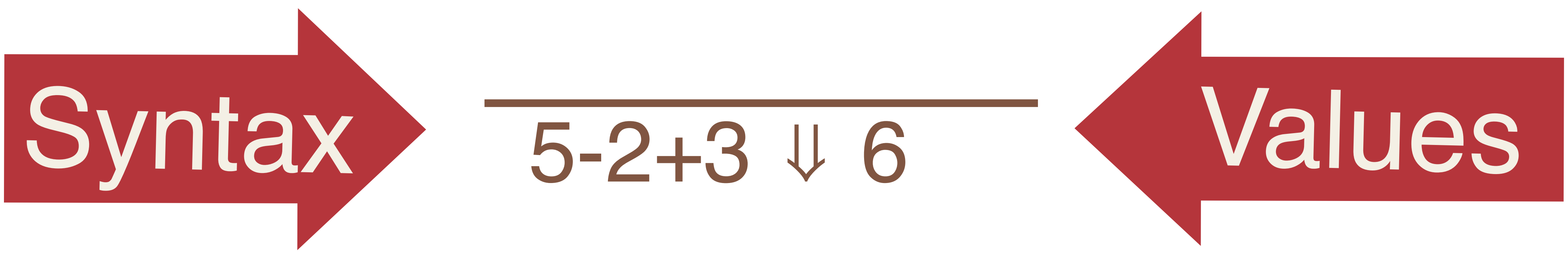
EWHILEF

$$\frac{\sigma, b \Downarrow \textbf{false}}{\sigma, \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end} \Downarrow \sigma}$$

Why is this a better formulation than the definition of ceval?

Big-Step Semantics

- Binary relation on pairs of syntax and values
- Read '⇓' as 'evaluates to'
- Specifies what values program can map to



- Good for whole program reasoning
  - Compiler Correctness; program equivalence;
- Bad for talking about intermediate states
  - Concurrent programs; errors

Week 7

Smallstep Operational Semantics and Denotational Semantics

Step Size

Big Step Semantics

$$\frac{e_n \Downarrow n \qquad e_m \Downarrow m}{e_n +_E e_m \Downarrow n + m}$$

$$\frac{}{C\ n \Downarrow n}$$

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

Big-Step reduction relation is from syntax, to values.

Small-Step reduction relation is from syntax, to syntax.

Small-Step Termination

- How to tell when we're 'done' evaluating?
- Define a class of syntactic values:

$$\frac{}{\textbf{value}\ C\ n}$$

Now we can talk about making progress

Theorem [STRONG PROGRESS]:

For any term t, either t is a value or there exists a term t' such that t $\longrightarrow$ t'.

Normal Form

A term e that isn't reducible is in normal form.

$$\neg \exists e'.\, e \longrightarrow e'$$

How is this different from a value?

Syntactic versus semantic.

Do not need to coincide!

Semantics Recap

- We've considered several flavors of Operational Semantics:
  - Abstract machine specifies *how* an expression is executed:
- σ, c $\Downarrow$ σ' reads as 'when run in initial state σ, c produces (i.e. evaluates to) final state σ'
- $e_1 \longrightarrow e_2$ reads as 'e_1 reduces to e_2 in a single step'
- $e_1 \longrightarrow^* e_2$ reads as 'e_1 reduces to e_2 in zero or more steps'

Recap (Denotational Semantics)

- Key Idea: define semantics via translation to a well-understood **semantic domain:**
  - Using sets, we can model partial and total functions on state
  - Can also represent nondeterministic semantics
- Can relate different kinds of semantics
- Denotational semantics are designed to be **compositional**
- Denotational semantics are useful for reasoning about program equivalence

Week 8

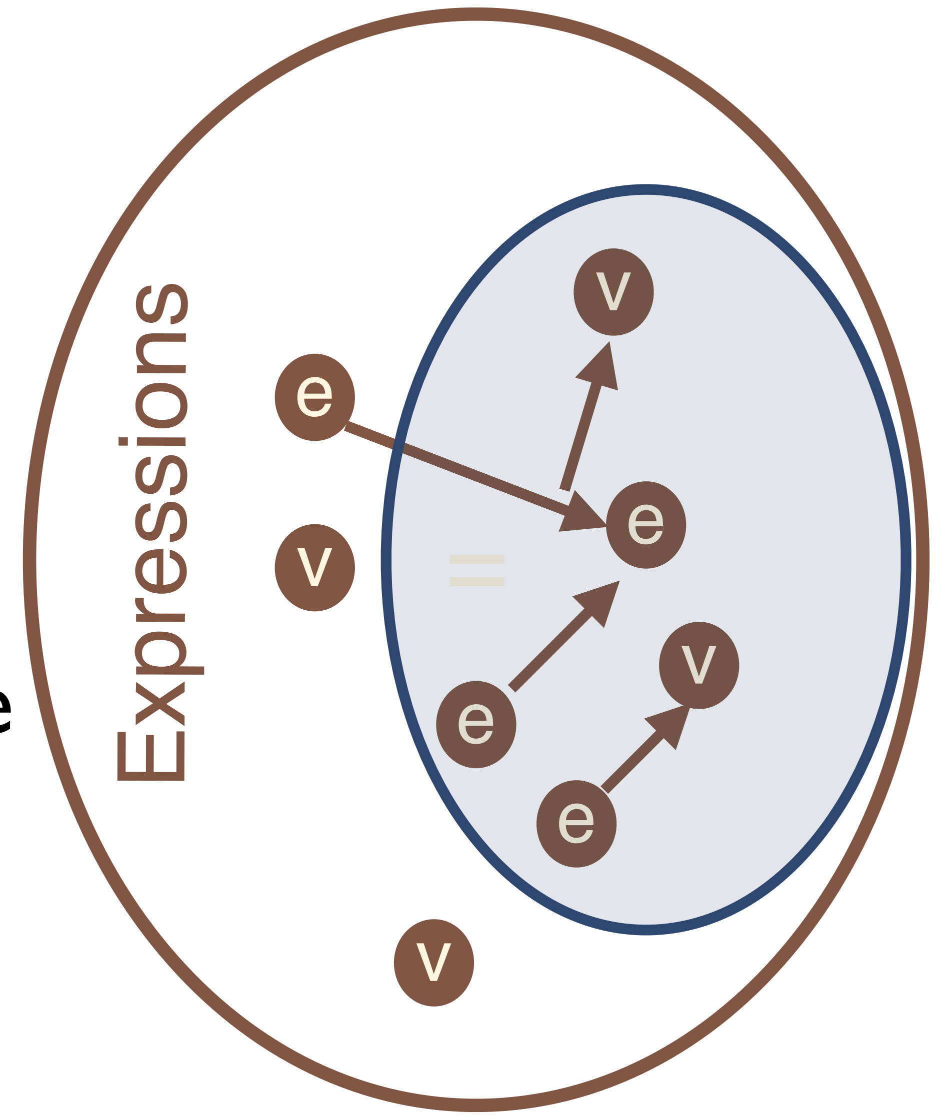
Type Systems and Simply-Typed Lambda Calculus

Static Semantics

A recipe for defining a language:

1. Syntax:
   - What are the valid expressions?
2. Semantics (Dynamic Semantics):
   - How do I evaluate valid expressions?
3. Sanity Checks (Static Semantics):
   - What expressions are "good", i.e have meaningful evaluations?

Type systems identify a subset of good expressions

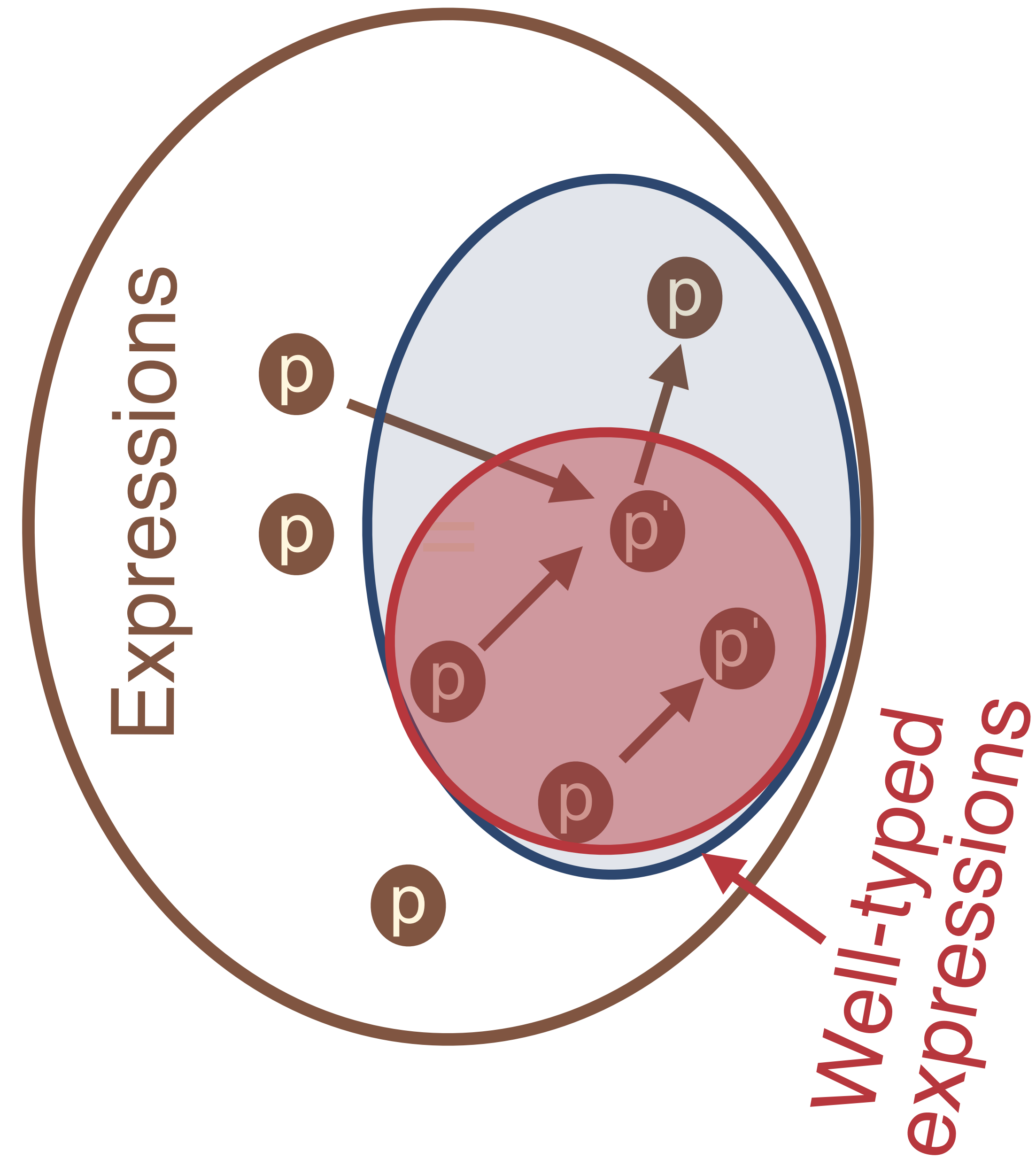


Type Safety

- When is a type system correct?
  - ⋆ Need to show this classification is sound. i.e. no false positives:

$$\vdash e : T \quad \rightarrow \quad \sim e \text{ is bad!}$$

- If the a language's type system is sound, it is said to be type-safe.
- Soundness relates provable claims to semantic property



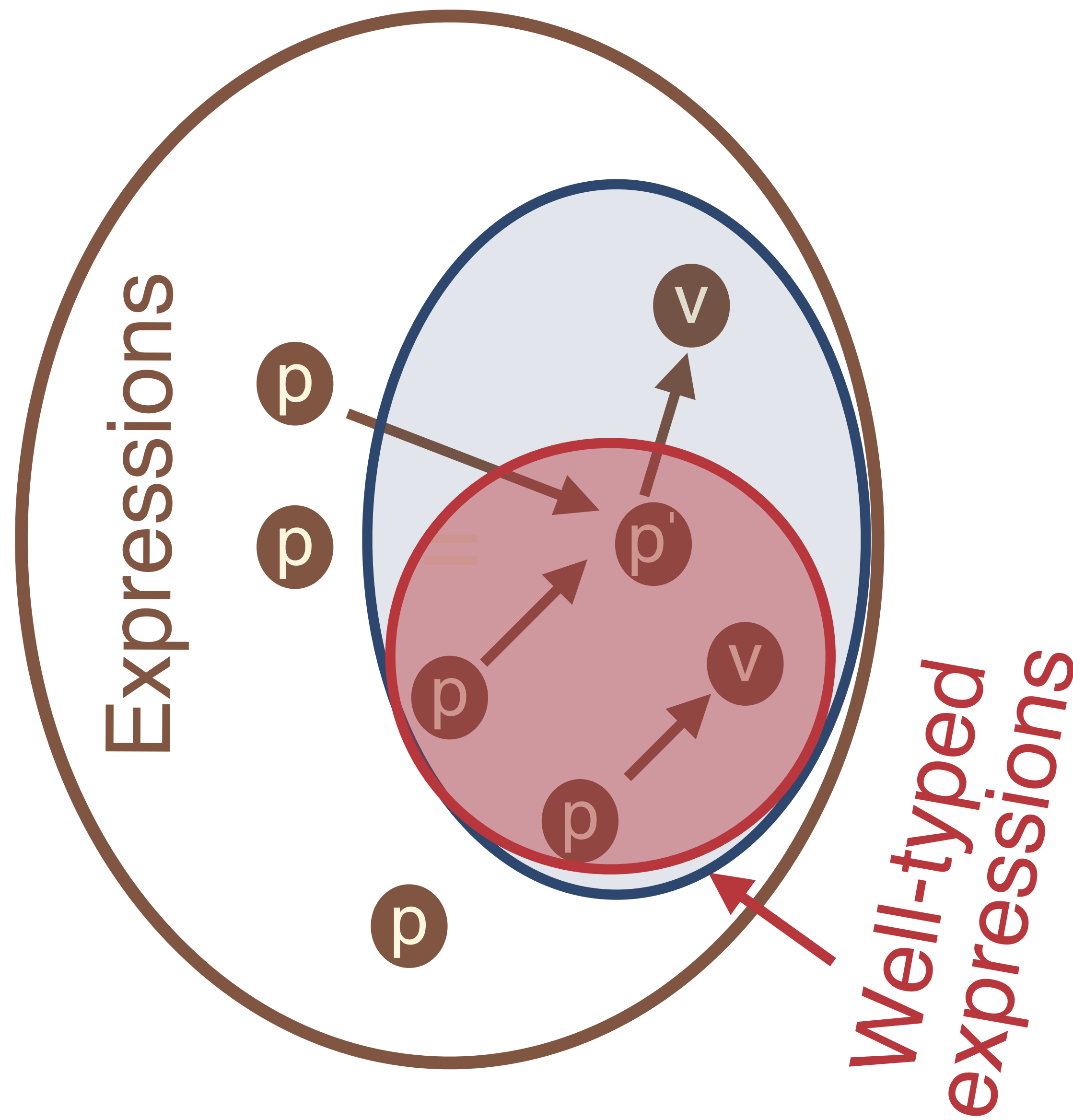
Progress

<u>Theorem [Progress]</u>: Suppose e is a well-typed expression ($\vdash e : T$). Then either e is a value or there exists some e' such that e evaluates to e' ($\sigma, e \longrightarrow e'$).

Values:

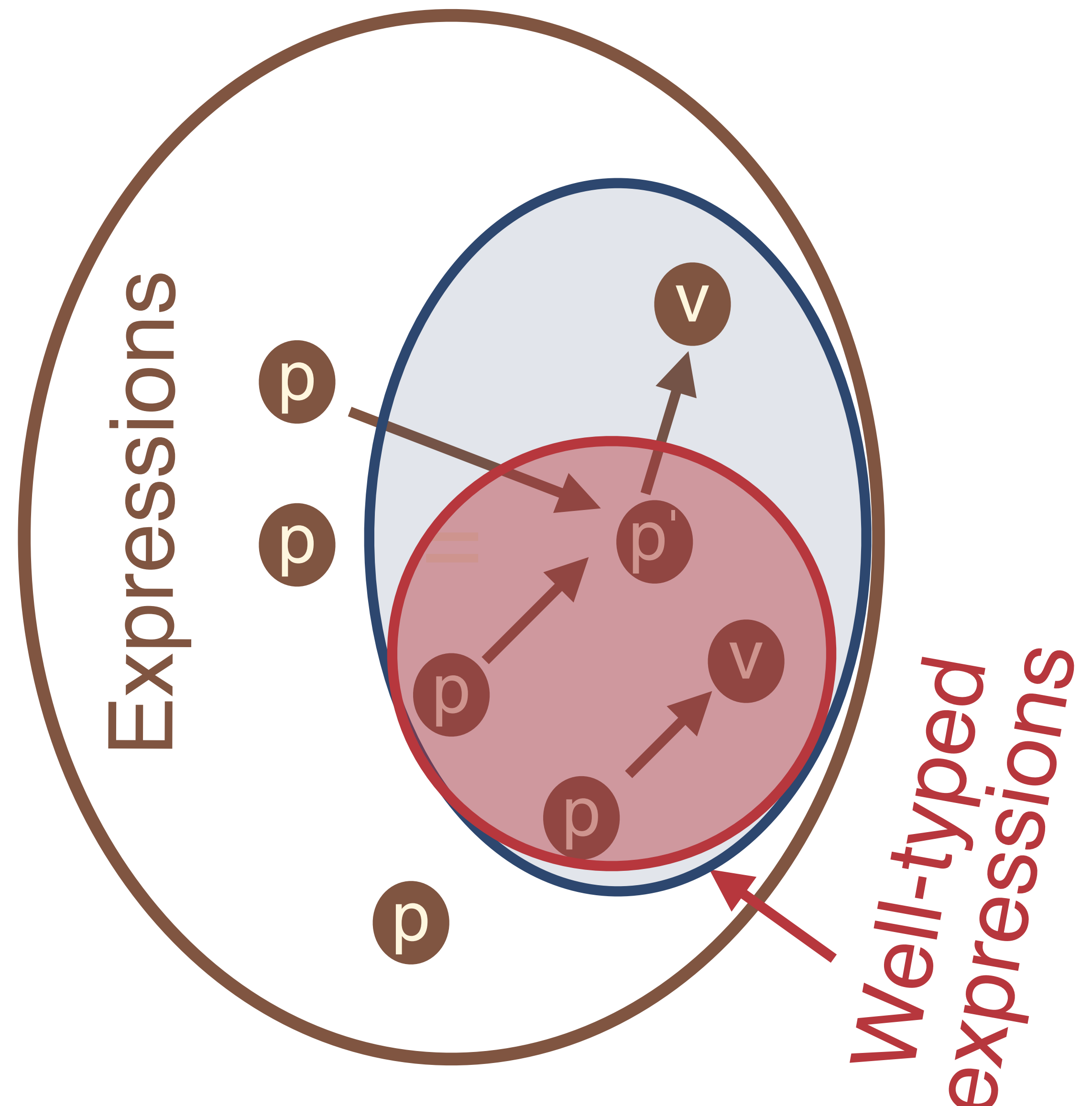
$$\frac{}{\text{value true}}\ \textsc{TValue}$$

$$\frac{n \in N}{\text{value } n}\ \textsc{NumValue}$$



Preservation

- **<u>Theorem</u>** <u>[PRESERVATION]</u>: Suppose e is a well-typed term ($\vdash$ e : T). Then, if e evaluates to e', e' is also a well-typed term under the empty context, with the same type as e ($\vdash$ e' : T).



Type Soundness

Theorem [Type Soundness]: If an expression e has type T, and e reduces to e' in zero or more steps, then e' is not a stuck term.

Proof.

By induction on $\sigma, e \longrightarrow^* e'$…

Qed.

- Corollary [Normalization]: If an expression e has type T, e reduces to a value in zero or more steps.

Typing STLC

$$\Gamma \vdash t : T$$

Γ maps bound variables to their types

- Here are the typing rules:

$$\frac{}{\Gamma \vdash n : nat} \text{ TNUM}$$

$$\frac{\Gamma[x \mapsto T_1] \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \rightarrow T_2} \text{ TABS}$$

$$\frac{\Gamma \vdash t : nat}{\Gamma \vdash t{+}1 : nat} \text{ TINC}$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \text{ TAPP}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \text{ TVAR}$$

Normalization

- **Theorem** [NORMALIZATION]: If an expression e has type T in the empty context, e reduces to a value in zero or more steps.

Why is STLC normalizing but not IMP?

STLC+Fix

- Updated Syntax:

  t ::= … | fix t

- Updated Semantics:

$$\frac{t_1 \longrightarrow t_1'}{\text{fix } t_1 \longrightarrow \text{fix } t_1'}$$

$$\frac{}{\text{fix } (\lambda x{:}T.t_1) \longrightarrow [x{:=}\text{fix } (\lambda x{:}T.t_1)]t_1}$$

Week 9

Subtyping

Subsumption

Would like this to typecheck:

Dist ⟨x=2, y=2, R=0, G=140, B=255⟩

$$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 <: T_2}{\Gamma \vdash t_1 : T_2} \text{ TSUB}$$

How to define T1 <: T2?

Substitutability: If T1 <: T2, then any value of type T1 must be usable in every way a T2 is.

The difficulty is ensuring this is safe (i.e. doesn't break type safety)!

Variance

Variance is a property on the arguments of type constructors like function types $(A \to B)$, tuples $(A \times B)$, and record types

F(A) is **covariant** over A if $A <: A'$ implies that $F(A) <: F(A')$

F(B) is **contravariant** over B if $B' <: B$ implies that $F(B) <: F(B')$

F(T) is **invariant** over T otherwise

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2} \quad \text{SB-TUPLE}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \quad \text{SB-ARROW}$$

Week 10

Axiomatic Semantics and Hoare Logic

Hoare Triple

- Step 1B: Define a judgement for claims about programs involving assertions
- Partial Correctness Triple:

$$\{P\}\ c\ \{Q\}$$

Validity

We can now precisely define wh[en] a partial Hoare Triple is valid:

$$\{P\}\ c\ \{Q\} \equiv \forall \sigma.\ \sigma \models P \rightarrow \forall \sigma'.\ \sigma, c \Downarrow \sigma' \rightarrow \sigma' \models Q$$

If we start in a state satisfying P

And c terminates in a state.

then that final state satisfies Q

$\sigma \models P$

VALIDITY

The rule admits the possibility that there is no such σ'

Hoare While!

I is a *loop invariant*:

- Holds before loop
- Holds after each loop iteration
- Holds when the loop exits

$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \{I\}\ \text{while } b \text{ do } c \text{ end}\ \{I \wedge \neg b\}}\ \textsc{HLWhile}$$

Rule Review

$$\frac{}{\vdash \{Q[X:=a]\}\, X:=a\, \{Q\}}\ \text{HLAssign} \qquad \frac{}{\vdash \{Q\}\ \text{skip}\ \{Q\}}\ \text{HLSkip}$$

$$\frac{\vdash \{P\}\ c_1\ \{R\} \qquad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1;c_2\ \{Q\}}\ \text{HLSeq}$$

$$\frac{\vdash \{P \land b\}\ c_1\ \{Q\} \qquad \vdash \{P \land \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}\ \text{HLIf}$$

$$\frac{\vdash \{I \land b\}\ c\ \{I\}}{\vdash \{I\}\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end}\ \{I \land \neg b\}}\ \text{HLWhile}$$

Loop Invariants

Hoare Logic is a structural model-theoretic proof system

- Rules characterize a set of states consistent with the requirements imposed by the pre- and post-conditions
- Highly mechanical: intermediate states can almost always be automatically constructed
- One major exception:

$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \{I\}\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end}\ \{I \wedge \neg b\}}\ \text{HLW}{\small\text{HILE}}$$

The invariant must:

- be weak enough to be implied by the precondition
- hold across each iteration
- be strong enough to imply the postcondition

Loops


```
{{ True }} ->
      {{  min a b = min a b   }}
   X := a;
      {{  min X b = min a b   }}
   Y := b;
      {{  min X Y = min a b   }}
   Z := 0;
      {{  Inv                 }}
   while X <> 0 && Y <> 0 do
      {{ Inv /\ (X <> 0) /\ Y <> 0) }} ->
      {{ Z + 1 + min (X - 1) (Y - 1) = min a b }}
    X := X - 1;
      {{ Z + 1 + min X (Y - 1) = min a b }}
    Y := Y - 1;
      {{ Z + 1 + min X Y = min a b }}
    Z := Z + 1;
      {{ Inv  }}
   end
{{ ~(X <> 0 /\ Y <> 0) /\ Inv) }} ->
{{ Z = min a b }}
```

This style of proof construction is known as weakest precondition inference

Identify a precondition that satisfies the largest set of states that still enable verification of the postcondition

Can automate this inference once we know the loop invariant

Week 11 - 13

Dafny

Dafny

- Applies Hoare reasoning to programs
- User provides specifications in the form of pre- and postconditions, along with other assertions
- Dafny verifies that the program meets the specification
  - When successful, Dafny guarantees (total) functional correctness of the program

Correctness:

- Reflects base-level semantic properties (no runtime errors (e.g., divide-by-zero, null pointer dereferences, etc.)
- But, also justifies higher-level application-specific properties (e.g., correctness of distributed systems, …)

Specifications

- Specifications are meant to capture salient behavior of an application, eliding issues of efficiency and low-level representation.

    forall k:int :: 0 <= k < a.Length ==> 0 < a[k]

- Specifications in Dafny can be arbitrarily sophisticated.

- We can think of Dafny as being two smaller languages rolled into one:

    - An imperative core that has methods, loops, arrays, if statements… and other features found in realistic programming languages. This core can be compiled and executed.

    - A pure (functional) specification language that supports functions, sets, predicates, algebraic datatypes, etc. This language is used by the prover but is not compiled.

Invariants


```
method loopEx (n : nat)
{
    var i : int := 0;
    while (i < n)
        invariant 0 <= i
        {
            i := i + 1;
    }
    assert i == n;
}
```

Dafny will not verify this program. Why?

Need invariants to be inductive!
- hold in the initial state
- hold in every state reachable from the initial state
- strong enough to imply the postcondition

```
method loopExCheckFixed (n : nat)
{
    var i : int := 0;
    while (i < n)
        invariant 0 <= i <= n
        {
            i := i + 1;
    }
    assert i == n;
}
```

Decreases clause


```
function seqSum (s : seq<int>, lo : int, hi : int) : int
    requires 0 <= lo <= hi <= |s|
{
    if (lo == hi) then 0 else s[lo] + seqSum(s, lo+1, hi)
}
```

Dafny complains that it cannot prove the recursive call terminates - it is unable to identify a termination metric that signals every recursive call gets "smaller"

```
function seqSum (s : seq<int>, lo : int, hi : int) : int
    requires 0 <= lo <= hi <= |s|
     decreases hi - lo
{
    if (lo == hi) then 0 else s[lo] + seqSum(s, lo+1, hi)
}
```

What about using -lo as a decreases clause?

Lemmas

Sometimes, the property we wish to prove cannot be automatically verified. To help Dafny, we can provide *lemmas*, theorems that exist in service of proving some other property.

```
method FindZero(a: array<int>) returns (index: int)
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
{
}
```

Precondition restricts input array such that all elements are greater than or equal to zero and each successive element in the array can decrease by at most one from the previous element.

We can take advantage of this observation in searching for the first zero in the array, by skipping elements. E.g., if a[j] = 7, then index of next possible zero cannot be before a[j + a[j]], i.e., if j = 3, then first possible zero can only be at a[10]

Lemmas and Induction

Express this inductive property:

```
assert count(a + b) == count([a[0]]) + count(a[1..] + b);
```

using recursion

```
lemma DistributiveLemma(a: seq<bool>, b: seq<bool>)
  ensures count(a + b) == count(a) + count(b)
{
  if a == [] {
    assert a + b == b;
  } else {
    DistributiveLemma(a[1..], b);
    assert a + b == [a[0]] + (a[1..] + b);
  }
}
function count(a: seq<bool>): nat
{
  if |a| == 0 then 0 else
    (if a[0] then 1 else 0) + count(a[1..])
}
```

Proof Calculations

Proof that Nil is idempotent over list appends

```
lemma prop_app_Nil(xs: list)
  ensures app(xs, Nil) == xs;
{
    match xs {
        case Nil =>
        case Cons(y,ys) =>
            calc { app(xs,Nil) ;
            == app(Cons(y,ys), Nil);
            == Cons(y, app(ys,Nil));
            == { prop_app_Nil(app(ys,Nil)); }  // proof hint
           xs;
           }
    }
}
```

Proof Calculations and Induction


```
lemma {:induction false} MirrorMirror<T>(t: Tree)
    ensures mirror(mirror(t)) == t
    {
        match t
            case Leaf(_) =>
            case Node(left,right) =>
                calc
                {
                    mirror(mirror(Node(left,right)));
                    ==
                    mirror(Node(mirror(right),mirror(left)));
                    ==
                    Node(mirror(mirror(left)),mirror(mirror(right)));
                    ==  // IH
                   { MirrorMirror(left);    MirrorMirror(right); }
                  Node(left, right);
                }
    }
```

Proofs by Contradiction

General shape:

```
!Q -> (R /\ !R)
_______________
       Q
```

```
lemma Lem(args)
   requires P(x)
   ensures Q(x)
 {
    if !Q(x)                    // property is false
      {
        assert !P(x)            // contradiction: precondition is
        assert false            // true and false
    }
   assert Q(x)
}
```

Functional data structures

- In addition to support for inductive datatypes, Dafny also has specialized support for certain kinds of functional data structures, specifically sets and sequences.
- A set is an *order-less immutable* collection of *distinct* elements

  {2, 3, 3, 2} == {2, 2, 2, 3, 3, 3} == {2, 3}

  {2, 4, 4, 3, 5} == {5, 3, 4, 4, 2} == { 2, 3, 4, 5}
- Sets can be used in both specification and code

```
method Main()
     {
       var a: set<int> := {1,2,3,4};
       var b: set<int> := {4,3,2,1,1,2,3,4};
       assert |a| == |b|== 4;  // same length
       assert a - b == {};     // same sets
       print a, b;             // can print them
}
```

Sequences

A sequence is an ordered immutable list of (possibly non-unique) elements

```
method SeqsAreOrdered() {
   var s: seq<int> := [2,1,3];
   var t: seq<int> := [1,2,3];
   assert s != t;
}
```

```
method CheckLength() {
  var a:array<int> := new int[][1,2,3,4];
  var s:set<int> := {1,2,3,4,4,3,2,1,1,1,1,1};
  var t:seq<int> := [1,2,3,4];
  assert a.Length == |s| == |t| == 4;
}
```

Two-State Predicates

- Specifications for imperative programs often need to relate the value of a structure in the pre-state (before the method executes) and the post-state (after the method completes).
- Use old(E) to refer to the value of E in the prestate
  - old tracks heap dereferences

```
method Increment(a : array<int>, i: int)
   requires 0 <= i < a.Length
   modifies a
   ensures a[i] == old(a)[i] + 1
{
   a[i] := a[i] + 1;
}
```

vs.

```
method Increment(a : array<int>, i: int)
   requires 0 <= i < a.Length
   modifies a
   ensures a[i] == old(a[i]) + 1
{
   a[i] := a[i] + 1;
}
```

Week 14

Separation Logic

Motivation

- Hoare Logic is defined in terms of assertions on states:
    - states are maps from variables to their values
    - most programming languages also support the notion of a heap:
        - variables map to addresses
        - the contents at a given address can be shared and aliased
    - Embedding notions of sharing and mutation into the logic is problematic
- Separation Logic enables local reasoning about memory
    - It is a *substructural* logic that controls how memory (heaps) are constructed and used
    - In classical logical systems (e.g., Hoare logic) can:
        - add (weaken) or contract (duplicate) assumptions. Here, think of assumptions as claims we can make about resources (aka states or memory)
        - Substructural logics restrict how assumptions can be introduced:
            - can't invent extra memory to satisfy predicates
            - can't duplicate memory

Separation Logic

Rather than trying to explicitly reason about heap structure and aliasing within Hoare Logic, introduce new logical operators to reason about how heaps (aka resources) are used

- emp: empty heap
- x $\mapsto$ v: heap has a cell at x with value v
- P * Q: separating conjunction (disjoint parts of the heap)
- P $-*$ Q: separating implication (hypothetical heap extension)

Assertions now describe heap and variable conditions

- Conjunction (*), implication (-*)
- Emp and points-to relation

Example:

$$h \models P * Q$$

means: the heap can be divided into disjoint parts, one which satisfies P ($h \models P$)and the other which satisfies Q ($h \models Q$)

Frame Rule

Note that in the rule:

$$\{\mathrm{x} \mapsto v_1 * \mathrm{y} \mapsto v_2\}[\mathrm{x}] := \mathrm{v}_3\{\mathrm{x} \mapsto v_3 * \mathrm{y} \mapsto v_2\}$$

the assertion on y is unused and provides no meaningful information relevant to the proof

$$\frac{\{\psi\} \quad \mathsf{c} \quad \{\phi\}}{\{\psi * \mathsf{F}\} \quad \mathsf{c} \quad \{\phi * \mathsf{F}\}}$$

Importantly, the following is not valid:

$$\frac{\{\psi\} \quad \mathsf{c} \quad \{\phi\}}{\{\psi \wedge \mathsf{F}\} \quad \mathsf{c} \quad \{\phi \wedge \mathsf{F}\}}$$

Magic Wand (Separating Implication)

$$P \quad -\!\!* \quad Q$$

reads:

Extending a heap *h* with another (disjoint) heap that satisfies *P*, results in a new heap that satisfies *Q*

$$\frac{\forall h'.h' \perp h, h' \models P \rightarrow \ h \oplus h' \models Q}{\langle h, \gamma \rangle \models P -\!\!* Q}$$

$$\frac{\langle h, \gamma \rangle \models P * (P -\!\!* Q)}{\langle h, \gamma \rangle \models Q}$$