CS 456

Programming Languages
Fall 2024

Week 15

Course Review

Defining a Language (Week 2)

A "recipe" for defining a language:

1. Syntax:
   - What are the valid expressions?
2. Semantics (Dynamic Semantics):
   - What is the meaning of valid expressions?
3. Sanity Checks (Static Semantics):
   - What expressions have meaningful evaluations?

Lambda Calculus

- Lambda calculus was developed by Alonzo Church in the 30s
  - A core language in which *everything* is a function
- Syntax of Lambda terms:

```
t ::= x          Variable
    | λx.t       Lambda abstraction
    | t t        Application
```

Variable Scopes


```
t ::= x
    | λx.t
    | t t
    | n
    | t + t
x ∈ Var
n ∈ N
```

1. A variable x is **bound** when it occurs in the body t of a lambda abstraction `λx.t`:

2. A variable x is **free** if it is not bound by an enclosing lambda expression:

3. A **closed** term has no free variables

Inference Rules

To describe the meaning of lambda-calculus expressions, we will use a notation called *inference (or reduction) rules*.

Informally, a rule of the form:

$$\frac{A_1, A_2, \ldots, A_n}{\mathtt{t}_1 \to \mathtt{t}_2}$$

reads:

Expression $\mathtt{t}_1$ evaluates to (or "reduces" to) $\mathtt{t}_2$
if the constraints defined by $A_1, A_2, \ldots, A_n$ hold

We'll delve into a more formal characterization of what these rules signify later in the course …

Semantics

PLUS NUMBERS

Reduction Rules

$$\frac{\text{value } t_1 \qquad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$\frac{\text{value } t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x \coloneqq t_2]t_1}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1 + t_2 \longrightarrow t_1 + t_2}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}$$

$$\frac{n \in \mathbb{Z} \qquad m \in \mathbb{Z}}{n + m \longrightarrow n +_{\mathbb{Z}} m}$$

Value Rules

$$\frac{}{\text{value } (\lambda x.t)}$$

$$\frac{n \in \mathbb{Z}}{\text{value } n}$$

Evaluation Strategies

CALL-BY-VALUE
AKA STRICT

Recall that lambda abstractions and numbers are values:

The lambda calculus' values are the functions:

$$\frac{}{\text{value } \lambda x.t}$$

This is called a *call-by-value* semantics: redexes are always the top-most function that is applied to a value:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \frac{\text{value } t_1 \qquad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{\text{value } t_2}{(\lambda x.t_1)\ t_2 \longrightarrow [x{:=}t_2]t_1}$$

Normalization

- If *every* program in a language is guaranteed to always evaluate to a normal term, we say the language is *strongly normalizing*.
  - Formally:
  - **<u>Statement of Strong Normalization:</u>**
  - For any term `t`, all sequences of reduction steps starting from `t` eventually reaches a normal form `t'`.
- Every program in a strongly normalizing language terminates.

Evaluation Strategies

Call-By-Name
Aka Lazy

An alternative: beta-reductions are performed as soon as possible:

$$\frac{}{(\lambda x.t_1)\ t_2 \longrightarrow [x \coloneqq t_2]t_1} \qquad \frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

```
(λx.λy.y x)(5+2)λx.x+1
⟶(λy.y (5+2)) λx.x+1
⟶(λx.x+1) (5+2)
⟶(5 + 2) + 1
⟶ 7 + 1
⟶ 8
```

```
(λf.f 7)((λx.x x) λy.y)
⟶ ((λy. y) (λy. y)) 7
⟶ (λy. y) 7
⟶ 7
```

term duplicated!

Evaluation Strategies

Call-By-Name

$(\lambda x.x + x)(5 + 6)$
$\longrightarrow (5 + 6) + (5 + 6)$
$\longrightarrow 11 + (5 + 6)$
$\longrightarrow 11 + 11$
$\longrightarrow 22$

Laziness can lead to duplicated work!

Call-By-Value

$(\lambda x\ y.x + x)\ 5\ (5 + 6)$
$\longrightarrow (\lambda y.5 + 5)\ (5 + 6)$
$\longrightarrow (\lambda y.5 + 5)\ 11$
$\longrightarrow 5 + 5$
$\longrightarrow 10$

Strictness can lead to unnecessary work!

Fixpoints/Recursion (Week 3)

Define `Z = λ f. Z`$_{\texttt{f}}$

Now, Z defines a fixpoint for any f:

```
Z ≡ λ f. (λ y. ((λ x. (f (λ y. (x x y))))
                (λ x. (f (λ y. (x x y))))
                y))
```

Z computes the *least fixpoint* of a function.

Continuation-Passing Style

Is a technique that can translate any procedure into a tail recursive one.

More generally, it makes explicit the "linearization" of control that is otherwise implicit in a program

Example:

4 * 3 * 2 * fact(1)

Define the *context* of fact(1) to be

```
fn v => 4 * 3 * 2 * v
```

Here, the context is a function that given the value produced by `fact(1)` returns the result of `fact(4)`

CPS Translation

$C\,[\,x\,]\,k \;= k\;x$

Returning the value of a variable simply passes that value to the current continuation.

$C[\,\lambda x.e\,]\,k = k\,(\lambda x\;k'\,.\,C\,[\,e\,]\,k')$

A function takes an extra argument which represents the continuation(s) of its call point(s), and its body is evaluated in this context.

$C[\,e1(e2)\,]\,k = C\,[\,e1\,]\,\lambda v.\,C\,[\,e2\,]\,\lambda v'.v\,(v', k)$

An application evaluates its first argument in the context of a continuation that evaluates its second argument in the context of a continuation that performs the application and supplies the result to its context.

A-Normal Form

Consider a language with the following grammar:

```
M ::= v                                  [RETURN]
    | let x = V in M                     [BIND]
    | if V then M else M                 [BRANCH]
    | V(V1, … ,VN)                         [TAIL-CALL]
    | let x = (V V1… VN) in M            [NON-TAIL]
    | P(V1 … Vn)                         [PRIMITIVE CALL]
v ::= c | x | λx1…xn.M                   [VALUES]
```

A-Normal Form

- All continuations are implicit.
  - But, like CPS all intermediate expressions are named
  - And, control-flow is apparent from syntactic structure of the program
  - Tail calls distinguished from non-tail calls. Recall that a tail call is a function call that occurs as the last statement in the calling function.

Evaluation Contexts

- How do we think of continuations without an explicit lambda term to capture control-flow?
- An *evaluation context* is a term with a "hole" corresponding to the next expression to be evaluated. (The context surrounding the "hole" is an implicit representation of the continuation for any term substituted for the hole.)

```
E ::= [ ]
    | let x = E  in M
    | if E then M else M
    | F(V … V E M … M) (where F = V or F = O)
```

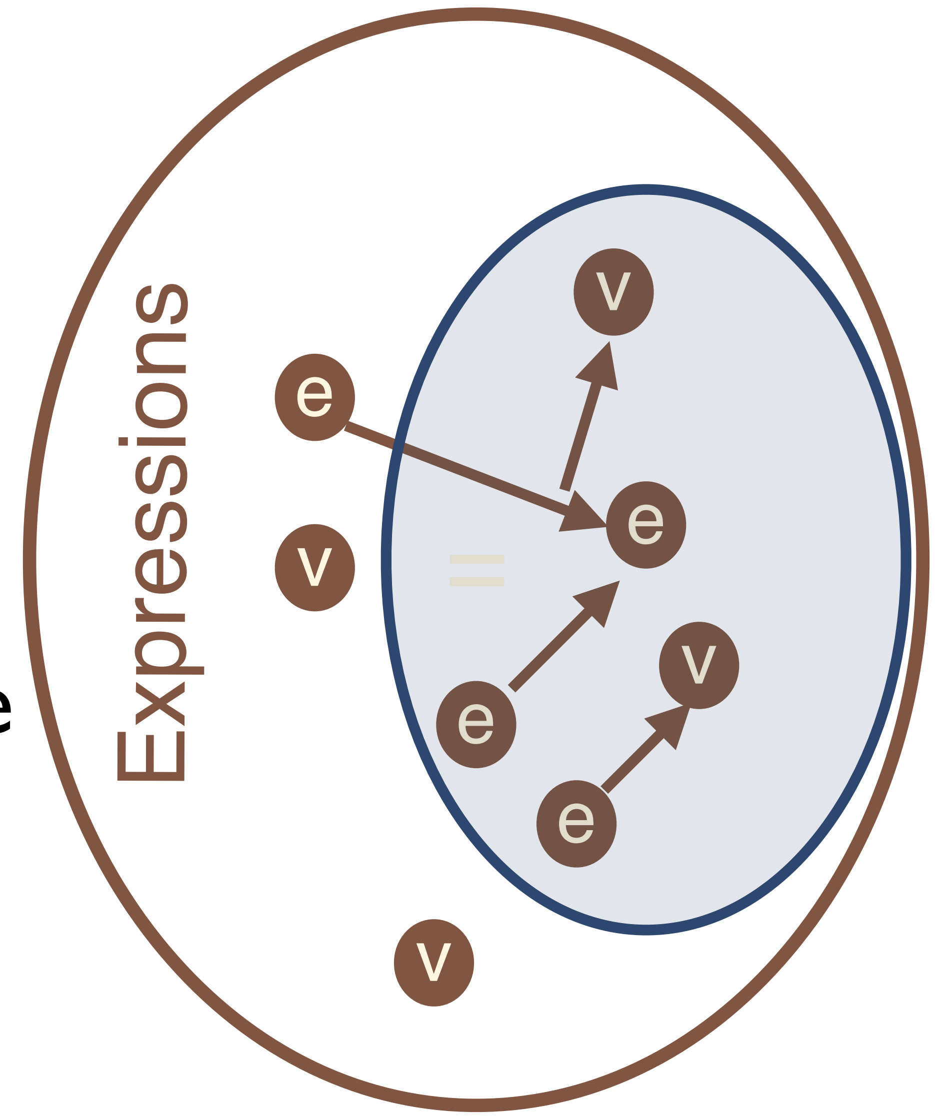
`M` is a term and `V` is a value as defined earlier; neither contain "holes." Thus, the structure of this grammar forces a left-to-right evaluation.

Static Semantics (Week 4)

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



Typing

- First step is to define badness:
  - Needs to be broad ... ties
  - Some
    annota
- What are

“Well-typed programs cannot go wrong”

A Theory of Type Polymorphism in Programming (Milner 78)

true + 3

x * ((y > 3) ? 3 : y)

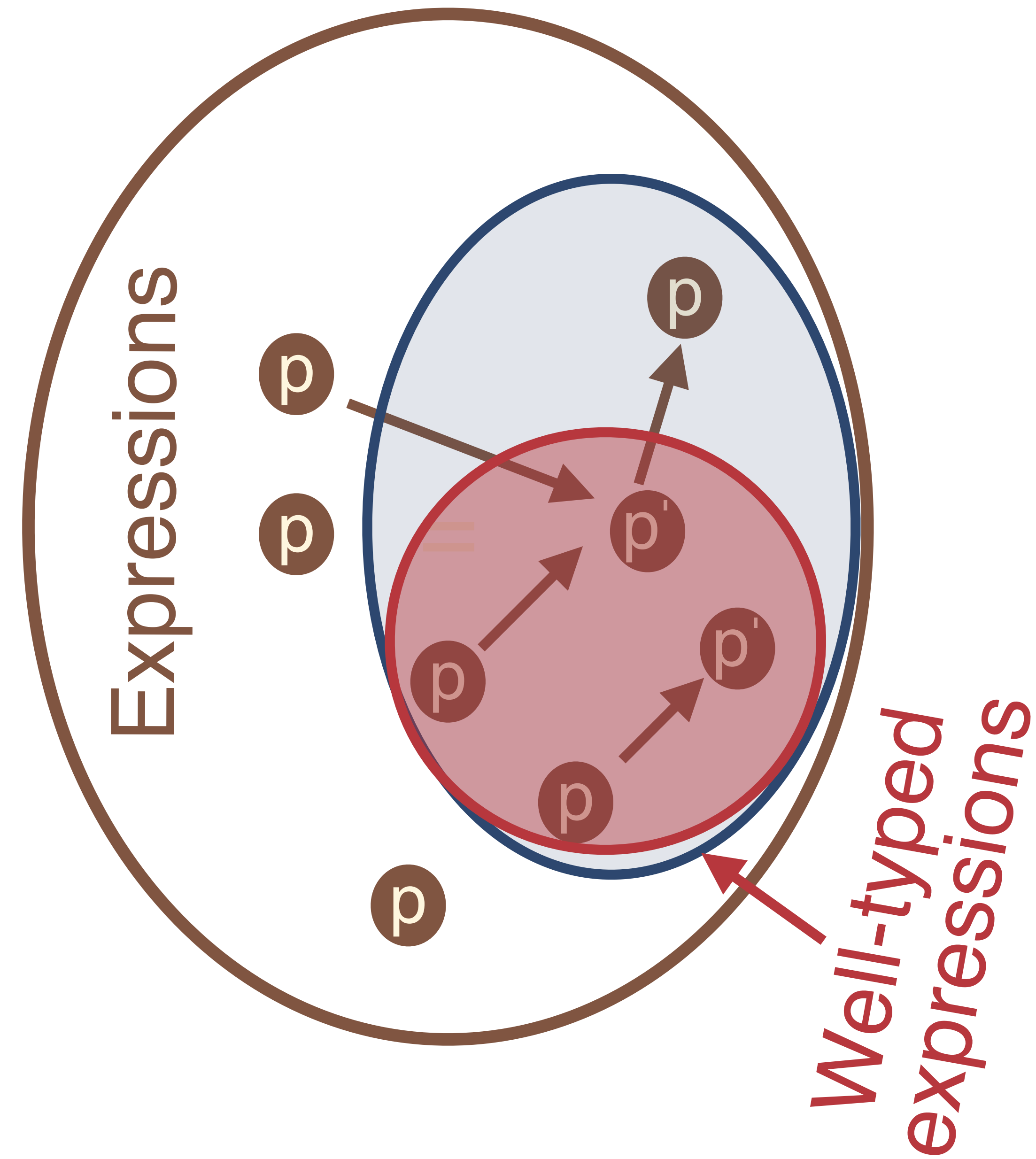
- Those that evaluate to a stuck expression: a normal form that isn’t a value

Type Safety

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

$$\vdash e : T \quad \rightarrow \quad v \in [\![e]\!]$$

- The set of values an expression can yield is non-empty (ie inhabited)
- If the a language's type system is sound, it is said to be type-safe.
- Soundness relates provable claims to semantic property



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.

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

Typing λ

- We first extend the syntax of terms to include type annotations
- Updated Syntax:

$$\texttt{T ::= T} \rightarrow \texttt{T | nat}$$

$$\texttt{n} \in \mathbb{N}$$

$$\texttt{t ::= x | } \lambda\texttt{x : T. t | t t | n | t + 1}$$

$$\frac{\text{value } t_1 \quad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

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

$$\frac{\text{value } t_2}{(\lambda x{:}T.\ t_1)\ t_2 \longrightarrow [x{:=}t_2]t_1}$$

. . .

$$\frac{}{\text{value } (\lambda x{:}T.t)}$$

Typing λ

- Need to refine our typing judgement:
  - We have two kinds of variables now
  - Variables can be unbound

$$\Gamma \vdash t : T$$

- Here are the typing rules:

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

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

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

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

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

System F (Week 5)

- ★ Here is the syntax of **pure System F**, with new bits **highlighted**.

$t ::= x \mid \lambda x{:}T.t \mid t\ t$

$\mid \Lambda X.t$ ⇐ Type Abstraction

$\mid t\ [T]$ ⇐ Type Application

$v ::= \lambda x{:}T.t \mid \Lambda X.t$

$T ::= T \to T$

$\mid \forall X.T$ ⇐ Universal Type

$\mid X$ ⇐ Type Variable

System F

★ Here are the new bits of the operational semantics

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{EAPP}_1$$

$$\frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'}\ \text{EAPP}_2$$

$$\frac{}{(\lambda x{:}T.e)\ v \longrightarrow e_1\ [x \mapsto v]}\ \text{EAPPABS}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ [T_2] \longrightarrow e_1'\ [T_2]}\ \text{ETAPP}$$

Where is ETAPP$_2$?

$$\frac{}{(\Lambda X.e_1)\ [T] \longrightarrow e_1\ [X := T]}\ \text{ETAPPTABS}$$

System F

★ Here are the **new bits** of the typing rules

$$\frac{\Gamma, [x \mapsto T_1] \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \rightarrow T_2} \quad \text{TABS} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \text{TVAR}$$

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

$$\frac{\Gamma \vdash t : T_2}{\Gamma \vdash \Lambda X.t : \forall X.T_2} \quad \text{TTABS}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_2}{\Gamma \vdash t_1\ [T_1] : T_2[X \coloneqq T_1]} \quad \text{TTAPP}$$

System F Metatheory

- ★ OTOH, the metatheory of System F diverges from STLC in key ways with respect to type inference:

  $\lceil x \rceil = x$

  $\lceil \lambda x{:}T.M \rceil = \lambda x.\lceil M \rceil$

  $\lceil M_1\ M_2 \rceil = \lceil M_1 \rceil\ \lceil M_2 \rceil$

  $\lceil \Lambda X.t \rceil = \lceil t \rceil$

  $\lceil t_1\ [T_2] \rceil = \lceil t_1 \rceil$

- **Theorem** [TYPE INFERENCE IS UNDECIDABLE]: Suppose m is a closed term in the untyped lambda calculus. Then it is undecidable if there exists some well-typed term system F term, t , such that $\lceil t \rceil = m$.

★ Bummer!

Prenex Predicative Polymorphism

- **Key Idea**: Restrict uses of polymorphism in types to enable type reconstruction.
- Can you think of one?
  - **Quantifiers only appear at the start of a formula and can only be instantiated with monomorphic types**

-

- This restriction can be expressed syntactically

$$\tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t$$

$$\sigma ::= \tau \mid \forall t.\ \sigma$$

$$e ::= x \mid e_1\ e_2 \mid \lambda x{:}\tau.\ e \mid \Lambda t.e \mid e\ [\tau]$$

- Type application is restricted to mono types

  $(\forall t.\, t \rightarrow t) \rightarrow (\forall t.\, t \rightarrow t)$ is <u>not</u> a valid type

- Abstraction only on mono types
- Cannot apply "id" to itself anymore
- Simple semantics and termination proof

ML's Polymorphic Let

ML solution: slight extension

Introduce "`let x : σ = e1 in e2`"

- With the semantics of "$(\lambda x : \sigma . e_2)\ e_1$"
- And typed as "$[e_1/x]\ e_2$"

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x : \sigma\ = e_1\ \texttt{in}\ e_2 : \tau}$$

This lets us write the polymorphic sort as

```
let
   s : ∀t.τ = Λt. ... code for  polymorphic sort ...
in
  ... s [nat] x .... s [bool] y
```

Type Inference (Week 6)

- More interesting question is how to avoid annotations if possible?
- **Today**: A **type inference** algorithm *infers* the **principal type** of a term missing some type annotations.
  - Such algorithms are key to OCaml's type system:

```
fold f acc [ ] = acc
fold f acc (x :: xs) = f x (fold f acc xs)

map (fun x -> x + 4) [1; 2]
```

Type Variables

- First step: extend STLC with Type Variables:

  $n \in \mathbb{N}$ $\qquad X_? \in \mathit{TypeVariables}$

  $T ::= \mathrm{Nat} \mid \mathrm{Bool} \mid T \to T \mid X_?$

  $t ::= x \mid \lambda x : T.\ t \mid t\ t \mid \mathtt{n} \mid t + t$
  $\quad \mid \mathrm{true} \mid \mathrm{false} \mid \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t$

- Typing rules and Operational Semantics are **same** as before:

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \textsc{TApp} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \textsc{TVar}$$

$$\frac{\Gamma[x \mapsto T_1] \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2}\ \textsc{TAbs}$$

...

Type Inference

Algorithm InferType(Γ, e_{in})

Input: Typing Context Γ, Untyped Lambda term e_{in}

Output: Well-typed STLC term or ill-typed

1. e_1 ← **annotate** all lambda abstractions in e_{in} with fresh Type Variables;
2. (T, ξ) ← **calculate type and constraints** that *any* solution for Γ and e_1 must satisfy
3. γ ← **find solution** to ξ, **or** report none exists (⊥)
4. **if** γ == ⊥ **then return** ill-typed
5. **return** $\gamma(\Gamma) \vdash \gamma(e_1) : \gamma(T)$

Type Inference

Algorithm InferType(Γ, e_{in})

Input: Typing Cont…

e_{in}

…resh

… **constraints** that *any*

… e_1 must satisfy

… **find solution** to ξ, **or** report none exists ($\perp$)

if $\gamma == \perp$ **then return** ill-typed

Since typing does not affect dynamic behavior, e_{in} is guaranteed to not get stuck if InferType returns a well-typed term!

Constraint-Based Typing

- **Key Idea$_1$**: record a set of ***constraints*** about how variables are used, and figure out how to solve them **later**
- Types constrain how things can be used:
  - The condition of an **if** expression must have type bool
  - Only expressions of type nat can be added together
- Formally, we define a new typing algorithm with the following judgement:

$$\Gamma \vdash e : T \mid \varnothing$$

Sensibility of Approach

- Let's take a step back and ask when this makes sense.
  - How does this relate to the original type system?

- **Theorem**: Constraint typing is sound. That is, if $\Gamma \vdash e: T \mid C$, then any solution S and γ must also be a solution for Γ and e.

- **Theorem**: Constraint typing is complete. That is, if S and γ are a solution for e and Γ and $\Gamma \vdash e: T \mid C$, then if γ and the type variables in C do not overlap, there must exist some solution for the original typing derivation, γ_2 and S'.

- **Theorem**: Constraint typing is sane: there is a solution to $\Gamma \vdash e: T \mid C$ if and only if there is a solution to Γ and e.

A Calculus for Subtyping (Week 7))

- Begin with a simple language for studying subtyping
- Key extension is Records (labeled products)

$t ::= x \mid \lambda x{:}T.t \mid t\ t \mid n \in \mathbb{N} \mid i \in \mathbb{R} \mid t + t \mid \ldots$

$\quad \mid \langle l1{=}t1, \ldots, ln = tn\rangle$ ⇐ Records

$\quad \mid t.l$ ⇐ Projection

$v ::= \lambda x{:}T.t \mid n \in \mathbb{N} \mid i \in \mathbb{R} \mid \langle l1{=}v1, \ldots, ln = vn\rangle$

can be empty $\langle\ \rangle$

$T ::= T \rightarrow T \mid \mathbb{R} \mid \mathbb{N}$

$\quad \mid \langle l1{:}T1, \ldots, ln : Tn\rangle$ ⇐ Record Types

$\quad \mid Top$

Subsumption

Would like this to typecheck:

$\text{Dist}\ \langle x=2, y=2, R=0, G=140, B=255 \rangle$

$$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 <: T_2}{\Gamma \vdash t_1 : T_2}\ \textsc{TSub}$$

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 $(A \rightarrow B)$, 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

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

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

Monads (Week 8)

- Conversion from ordinary to/from option types is tedious
- Would like to wrap (i.e, amplify) computed values with the option they are associated with
- Build a type constructor for this purpose:

```
module type Monad = sig
        type 'a t
        val return : 'a -> 'a t
        val bind : 'a t -> ('a -> 'b t) -> 'b t
    end
let (>>=) m f = bind m f
```

- A monad defines a container
- `return` puts a value in that container
- `bind` takes a container that contains a value of type 'a, a function that takes a value of type 'a and returns a container containing values of type 'b and returns that container

The State Monad


```
module State : Monad = struct
  type state     (* the record {s1; s2} *)
  type 'a t = state -> ‘a * state
   (* a state monad is a container over a state transition function *)
   (* in our example, these are the functions g, h, and i after they have
      been applied to an initial value. *)

  val return: ‘a -> ‘a t
  let return x = fun s -> (x, s)

  val bind: ‘a t -> (‘a -> ‘b t) -> ‘b t
  let bind s f =
    fun state ->
      (* apply the supplied state transition function *)
      let (a, s’) = s state in
      (* generate a new state transition function and value *)
      let (b, s’’) = f a s’ in
      (b, s’’)
end
```

IMP and Operational Semantics (Week 10)

(of Imp Commands)

```
C := skip
  | X := A
  | C ; C
  | if B then C
        else C end
  | while B do C end
```

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

Semantics

As a Relation

Key Idea: Define evaluation as a Inductive Relation ($\Downarrow$)

Inference Rules for $\Downarrow$

$$\frac{}{\sigma, n \Downarrow n}\ \text{ENum}$$

$$\frac{}{\sigma, x \Downarrow \sigma(x)}\ \text{EVar}$$

$$\frac{\sigma, e_n \Downarrow v_n \qquad \sigma, e_m \Downarrow v_m}{\sigma, e_n + e_m \Downarrow v_n +_{\mathbb{N}} v_m}\ \text{EAdd}$$

$$\frac{\sigma, e_n \Downarrow v_n \qquad \sigma, e_m \Downarrow v_m}{\sigma, e_n - e_m \Downarrow v_n -_{\mathbb{N}} v_m}\ \text{ESub}$$

$$\frac{\sigma, e_n \Downarrow v_n \qquad \sigma, e_m \Downarrow v_m}{\sigma, e_n * e_m \Downarrow v_n *_{\mathbb{N}} v_m}\ \text{EMult}$$

Semantics

Inference Rules for ⇓ (commands)

EWHILET

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

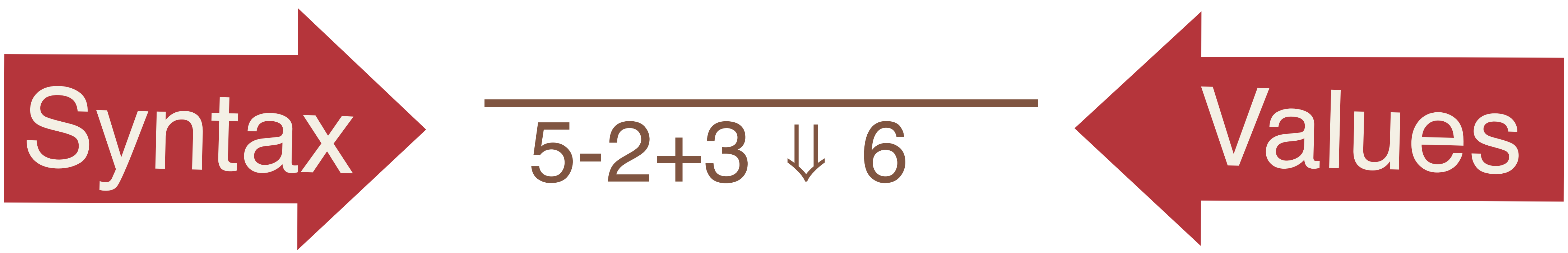
EWHILEF

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

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

Semantics (Week 11)

- Binary relation on pairs of syntax and values
- Read '$\Downarrow$' 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

Small-Step

- Binary relation on pairs of expressions
- Read ‘$e_1 \longrightarrow e_2$’ as ‘reduces to’
- Specifies single transition of abstract machine
- Exposes intermediate states

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!

Small-Step Semantics for Imp

Inference Rules for $\longrightarrow$

$$\frac{\sigma, a_1 \Downarrow v}{\sigma, x := a_1 \longrightarrow [x \mapsto v]\sigma, \text{skip}} \quad \underline{\text{CSAssn}}$$

$$\frac{\sigma_1, c_1 \longrightarrow \sigma_2, c_3}{\sigma_1, c_1; c_2 \longrightarrow \sigma_2, c_3; c_2} \quad \underline{\text{CSSeqStep}}$$

$$\frac{}{\sigma, \text{skip}; c_2 \longrightarrow \sigma, c_2} \quad \underline{\text{CSSeqSkip}}$$

Small-Step Semantics for Imp

Inference Rules for $\longrightarrow$

Reduction Rules

$$\frac{}{\sigma, \textbf{if } \text{true } \textbf{then } c_t \textbf{ else } c_f \textbf{ end} \longrightarrow \sigma, c_t} \text{ CSIfT}$$

$$\frac{}{\sigma, \textbf{if } \text{false } \textbf{then } c_t \textbf{ else } c_f \textbf{ end} \longrightarrow \sigma, c_f} \text{ CSIfF}$$

Congruence Rules

$$\frac{\sigma, b_1 \longrightarrow_B b_2}{\sigma, \textbf{if } b_1 \textbf{ then } c_t \textbf{ else } c_f \textbf{ end} \longrightarrow \sigma, \textbf{if } b_2 \textbf{ then } c_t \textbf{ else } c_f \textbf{ end}} \text{ CSIfStep}$$

Small-Step Semantics for Imp

Inference Rules for $\longrightarrow$

$$\frac{}{\sigma,\ \textbf{while}\ b\ \textbf{do}\ c \longrightarrow \sigma,\ \textbf{if}\ b\ \textbf{then}\ c;\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end}\ \textbf{else}\ \text{skip}\ \textbf{end}}\ \text{CSWHILE}$$

Denotational Semantics

Key Idea: 'Denotation' function translates source program to target mathematical object

- Define a **sem** … language T
- Deno … this domain:
- Denota … am means
- Abstrac … reasoning
  - Natura … program equivalence
- Finding domain can be tricky— Domain Theory

Important: $[\![\cdot]\!]$ is a total function— every program gets a meaning!

Denotational Semantics

Key Idea: 'Denotation' function translates source program to target mathematical object

$$
\begin{aligned}
[\![\cdot]\!]_{\mathbb{A}} &: \mathtt{A} \rightarrow \mathbb{N} \\
[\![\mathtt{n}]\!]_{\mathbb{A}} &\equiv [\![\mathtt{n}]\!]_{\mathbb{A}} +_{\mathbb{N}} [\![\mathtt{m}]\!]_{\mathbb{A}} \\
[\![\mathtt{x}]\!]_{\mathbb{A}} &\equiv \mathtt{??} \\
[\![\mathtt{n+m}]\!]_{\mathbb{A}} &\equiv [\![\mathtt{n}]\!]_{\mathbb{A}} +_{\mathbb{N}} [\![\mathtt{m}]\!]_{\mathbb{A}} \\
[\![\mathtt{n-m}]\!]_{\mathbb{A}} &\equiv [\![\mathtt{n}]\!]_{\mathbb{A}} -_{\mathbb{N}} [\![\mathtt{m}]\!]_{\mathbb{A}}
\end{aligned}
$$

Denotational Semantics

Key Idea: 'Denotation' function from program to meaning

$$[\![\cdot]\!]_C : C \to \mathcal{P}((\mathrm{Id} \to \mathbb{N}) \times (\mathrm{Id} \to \mathbb{N}))$$

$$[\![\mathtt{skip}]\!]_C \equiv \{(\sigma, \sigma)\}$$

$$[\![x{:=}a]\!]_C \equiv \{(\sigma, [x \mapsto v]\sigma) \mid (\sigma, v) \in [\![a]\!]_A \}$$

$$[\![c_1;c_2]\!]_C \equiv \{(\sigma_1, \sigma_3) \mid \exists \sigma_2.\ (\sigma_1, \sigma_2) \in [\![c_1]\!]_C \wedge (\sigma_2, \sigma_3) \in [\![c_2]\!]_C\}$$

$$[\![\mathbf{if}\ b\ \mathbf{then}\ c_t\ \mathbf{else}\ c_e]\!]_C \equiv$$
$$\{(\sigma_1, \sigma_2) \mid (\sigma_1, \mathtt{true}) \in [\![e_B]\!]_B \wedge (\sigma_1, \sigma_2) \in [\![c_t]\!]_C \}$$
$$\cup \{(\sigma_1, \sigma_2) \mid (\sigma_1, \mathtt{false}) \in [\![e_B]\!]_B \wedge (\sigma_1, \sigma_2) \in [\![c_e]\!]_C\}$$

Denotational Semantics

Key Idea: ‘Denotation’ function from program to meaning

$$[\![\cdot]\!]_C : C \to \mathcal{P}((\mathtt{Id} \to \mathbb{N}) \times (\mathtt{Id} \to \mathbb{N}))$$

$$[\![\mathtt{while\ b\ do\ c\ end}]\!]_C =$$

$$\{(\sigma, \sigma) \mid (\sigma, \mathtt{false}) \in [\![e_B]\!]_B\}$$
$$\cup \{(\sigma_1, \sigma_3) \mid (\sigma_1, \mathtt{true}) \in [\![e_B]\!]_B \wedge \exists \sigma_2.\ (\sigma_1, \sigma_2) \in [\![c]\!]_C$$
$$\wedge\ (\sigma_2, \sigma_3) \in [\![\mathtt{while\ b\ do\ c\ end}]\!]_C\}$$

- ★ The meaning of while is defined in terms of the meaning of while
- ★ This is not a *definition*, it is a *recursive equation*
- ★ Goal: find a **set** that satisfies this equation

Fixpoints

- A **fixpoint** is solution Fix_F to a recursive equation of the form:

$$Fix_F = F\,(Fix_F)$$

$$\text{where}\quad F : \mathcal{P}A \rightarrow \mathcal{P}A$$

- A fixpoint is also a solution to this sequence:

$$Fix_F \;=\; F^0(\emptyset) \cup F^1(\emptyset) \cup F^2(\emptyset) \cup F^3(\emptyset) \cup \ldots$$

Recap

- <u>Key Idea</u>: 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

Axiomatic Semantics (Week 12)

- Operational Semantics
  - Simple abstract machine shows *how* to evaluate expression
- Denotational Semantics
  - Map language construct to mathematical domains (e.g., sets) to describe what expressions mean

Metatheoretic Properties

Can Prove:

- Determinism of Evaluation
- Soundness of Program Transformations
- Program Equivalence

Axiomatic Semantics

Axiomatic Semantics

- Meaning given by proof rules
- Useful for reasoning about properties of *specific* programs

- <u>Step 1</u>: Define a language of claims
- <u>Step 2</u>: Define a set of rules (axioms) to build proofs of claims
- <u>Step 3</u>: Verify specific programs

Hoare Triple

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

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

Rule Review

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

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

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

$$\frac{\vdash \{P_W\}\ c\ \{Q_S\} \quad P \rightarrow P_W \quad Q_S \rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}\ \textsc{HLConseq}$$

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:

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

The invariant must:

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

Decorated Programs

Idea: include assertions in program

$\{ \text{True} \} \rightarrow \{ m = m \}$
X := m;
$\{ X = m \} \rightarrow \{ X = m \wedge p = p \}$
Z := p;
$\{ X = m \wedge Z = p \} \rightarrow \{ Z - X = p - m \}$
while X ≠ 0 **do**
$\{ Z - X = p - m \wedge X \neq 0 \} \rightarrow \{ (Z - 1) - (X - 1) = p - m \}$
Z := Z - 1;
$\{ Z - (X - 1) = p - m \}$
X := X - 1
$\{ Z - X = p - m \}$
end;
$\{ Z - X = p - m \wedge \neg (X \neq 0) \} \rightarrow \{ Z = p - m \}$

Precondition Inference


```
{{ 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 }}
```

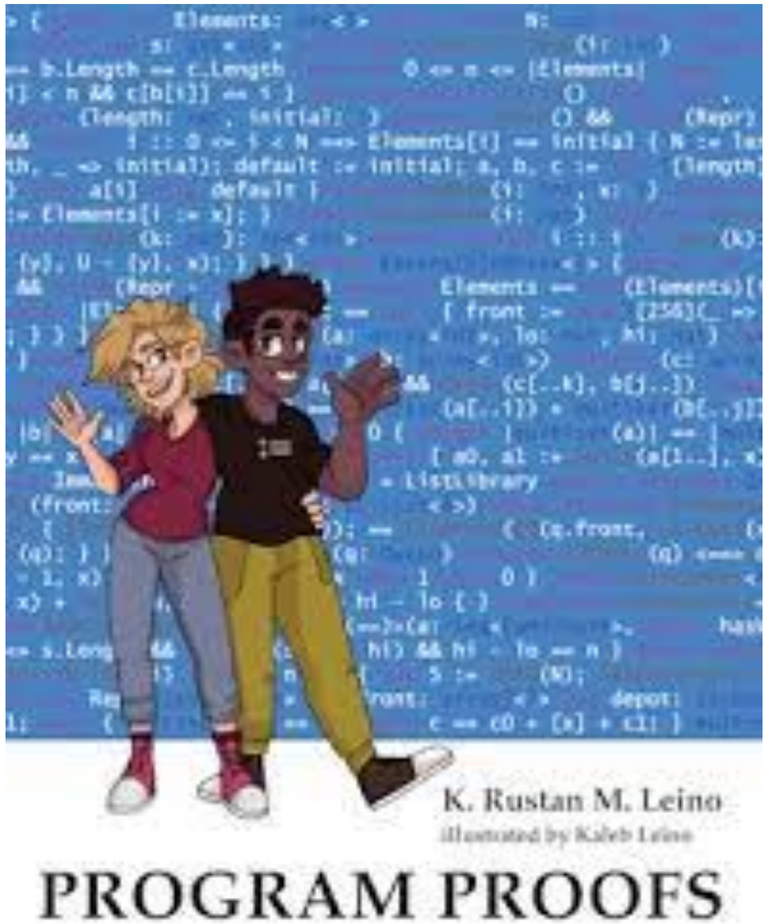
This style of proof construction is known as weakest precondition inference

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

Dafny (Weeks 13 and 14)

- Solver-aided language and verifier
- Language is statically-typed
- Imperative (with lots of functional language features)
- Compiles to C#, Java, Go, Python, …



Reference manual:

https://dafny.org/dafny/DafnyRef/DafnyRef.html

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;
}
```

Basic setup

- Specify correctness conditions as pre/post-conditions that can be checked (mostly) automatically using a WP inference procedure
- But, not all properties we wish to verify can be expressed in terms of actions on the transition relation defined by axiomatic rules

Need proof techniques that allow us to verify properties over:

1. Inductive datatypes (e.g., lists, trees, …)
2. Semantic objects (e.g., heaps)
3. Imperative data structures (e.g, arrays)

Additionally, Dafny verifies total correctness

- Hoare rules only assert partial correctness properties
- Need additional insight to reason about termination

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]

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;
      }
    }
}
```

Proofs by Contradiction

General shape:

$$\frac{!Q \;\texttt{->}\; (R \;/\backslash\; !R)}{Q}$$

```
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)
}
```