Programming Languages and Compilers (CS 421)



Elsa L Gunter 2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

10/2/14



- Data types play a key role in:
 - Data abstraction in the design of programs
 - Type checking in the analysis of programs
 - Compile-time code generation in the translation and execution of programs
 - Data layout (how many words; which are data and which are pointers) dictated by type

10/2/14 2



Terminology

- Type: A type t defines a set of possible data values
 - E.g. short in C is $\{x \mid 2^{15} 1 \ge x \ge -2^{15}\}$
 - A value in this set is said to have type t
- Type system: rules of a language assigning types to expressions

10/2/14



Types as Specifications

- Types describe properties
- Different type systems describe different properties, eg
 - Data is read-write versus read-only
 - Operation has authority to access data
 - Data came from "right" source
 - Operation might or could not raise an exception
- Common type systems focus on types describing same data layout and access methods

10/2/14 4



Sound Type System

- If an expression is assigned type t, and it evaluates to a value v, then v is in the set of values defined by t
- SML, OCAML, Scheme and Ada have sound type systems
- Most implementations of C and C++ do not



Strongly Typed Language

- When no application of an operator to arguments can lead to a run-time type error, language is strongly typed
 - Eg: 1 + 2.3;;
- Depends on definition of "type error"

10/2/14 5

10/2/14

2/14 6



Strongly Typed Language

- C++ claimed to be "strongly typed", but
 - Union types allow creating a value at one type and using it at another
 - Type coercions may cause unexpected (undesirable) effects
 - No array bounds check (in fact, no runtime checks at all)
- SML, OCAML "strongly typed" but still must do dynamic array bounds checks, runtime type case analysis, and other checks

10/2/14



Static vs Dynamic Types

- Static type: type assigned to an expression at compile time
- Dynamic type: type assigned to a storage location at run time
- Statically typed language: static type assigned to every expression at compile time
- Dynamically typed language: type of an expression determined at run time

10/2/14 8



Type Checking

- When is op(arg1,...,argn) allowed?
- Type checking assures that operations are applied to the right number of arguments of the right types
 - Right type may mean same type as was specified, or may mean that there is a predefined implicit coercion that will be applied
- Used to resolve overloaded operations

10/2/14



Type Checking

- Type checking may be done statically at compile time or dynamically at run time
- Dynamically typed (aka untyped) languages (eg LISP, Prolog) do only dynamic type checking
- Statically typed languages can do most type checking statically

10/2/14 10



Dynamic Type Checking

- Performed at run-time before each operation is applied
- Types of variables and operations left unspecified until run-time
 - Same variable may be used at different types



Dynamic Type Checking

- Data object must contain type information
- Errors aren't detected until violating application is executed (maybe years after the code was written)

10/2/14 11

10/2/14

12



Static Type Checking

- Performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time

10/2/14



Static Type Checking

- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can't check types that depend on dynamically computed values
 - Eq: array bounds

10/2/14 14



Static Type Checking

- Typically places restrictions on languages
 - Garbage collection
 - References instead of pointers
 - All variables initialized when created
 - Variable only used at one type
 - Union types allow for work-arounds, but effectively introduce dynamic type checks

10/2/14 15



13

17

Type Declarations

- Type declarations: explicit assignment of types to variables (signatures to functions) in the code of a program
 - Must be checked in a strongly typed language
 - Often not necessary for strong typing or even static typing (depends on the type system)

10/2/14 16



Type Inference

- Type inference: A program analysis to assign a type to an expression from the program context of the expression
 - Fully static type inference first introduced by Robin Miller in ML
 - Haskle, OCAML, SML all use type inference
 - Records are a problem for type inference

10/2/14



Format of Type Judgments

A type judgement has the form

 $\Gamma \mid -\exp : \tau$

- Γ is a typing environment
 - Supplies the types of variables and functions
 - Γ is a set of the form $\{x:\sigma,\ldots\}$
 - For any x at most one σ such that $(x : \sigma \in \Gamma)$
- exp is a program expression
- τ is a type to be assigned to exp
- |- pronounced "turnstyle", or "entails" (or "satisfies" or, informally, "shows")



Axioms - Constants

 $\Gamma \mid -n : \text{int}$ (assuming *n* is an integer constant)

 Γ |- true : bool Γ |- false : bool

- These rules are true with any typing environment
- Γ , n are meta-variables

10/2/14

19

21



Axioms – Variables (Monomorphic Rule)

Notation: Let $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$ Note: if such σ exits, its unique

Variable axiom:

$$\overline{\Gamma \mid -x:\sigma} \quad \text{if } \Gamma(x)=\sigma$$

10/2/14 20



Simple Rules - Arithmetic

Primitive operators ($\oplus \in \{+, -, *, ...\}$):

$$\frac{\Gamma \mid -e_1:\tau_1 \qquad \Gamma \mid -e_2:\tau_2 \quad (\oplus):\tau_1 \rightarrow \tau_2 \rightarrow \tau_3}{\Gamma \mid -e_1 \oplus e_2:\tau_3}$$

Relations (\sim \in { < , > , =, <=, >= }):

$$\frac{\Gamma \mid -e_1 : \tau \qquad \Gamma \mid -e_2 : \tau}{\Gamma \mid -e_1 \sim e_2 : \mathsf{bool}}$$

For the moment, think τ is int

10/2/14



Example: $\{x:int\} | -x + 2 = 3 : bool$

What do we need to show first?

$$\{x:int\} \mid -x + 2 = 3 : bool$$

10/2/14 22



Example: $\{x:int\} | -x + 2 = 3 : bool$

What do we need for the left side?

$$\frac{\{x: int\} \mid -x+2: int \qquad \{x: int\} \mid -3: int \\ \{x: int\} \mid -x+2=3: bool}{\text{Rel}}$$

10/2/14

23

Example:
$$\{x:int\} \mid -x + 2 = 3 : bool$$

How to finish?

$$\frac{\{x: int\} \mid - x: int \mid \{x: int\} \mid - 2: int}{\{x: int\} \mid - x + 2: int} \xrightarrow{AO} \{x: int\} \mid - 3: int \\ \hline \{x: int\} \mid - x + 2 = 3: bool}$$



Example: $\{x:int\} | -x + 2 = 3 :bool$

Complete Proof (type derivation)

$$\frac{\frac{\text{Var}}{\{x:\text{int}\}\mid - x:\text{int}} \frac{\text{Const}}{\{x:\text{int}\}\mid - 2:\text{int}}}{\frac{\{x:\text{int}\}\mid - x + 2:\text{int}}{\{x:\text{int}\}\mid - x + 2 = 3:\text{bool}}} \frac{\text{Const}}{\{x:\text{int}\}\mid - 3:\text{int}}$$

10/2/14



Simple Rules - Booleans

Connectives

$$\frac{\Gamma \mid -e_1 \text{ : bool} \qquad \Gamma \mid -e_2 \text{ : bool}}{\Gamma \mid -e_1 \text{ \&\& } e_2 \text{ : bool}}$$

$$\frac{\Gamma \mid -e_1 : \mathsf{bool} \quad \Gamma \mid -e_2 : \mathsf{bool}}{\Gamma \mid -e_1 \mid \mid e_2 : \mathsf{bool}}$$

10/2/14 26



Type Variables in Rules

If_then_else rule:

$$\frac{\Gamma \mid -e_1 : \mathsf{bool} \quad \Gamma \mid -e_2 : \tau \quad \Gamma \mid -e_3 : \tau}{\Gamma \mid - (\mathsf{if} \ e_1 \ \mathsf{then} \ e_2 \ \mathsf{else} \ e_3) : \tau}$$

- τ is a type variable (meta-variable)
- Can take any type at all
- All instances in a rule application must get same type
- Then branch, else branch and if_then_else must all have same type

10/2/14 27



25

29

Function Application

Application rule:

$$\frac{\Gamma \mid -e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \mid -e_2 : \tau_1}{\Gamma \mid -(e_1 e_2) : \tau_2}$$

• If you have a function expression e₁ of type τ₁ → τ₂ applied to an argument e₂ of type τ₁, the resulting expression e₁e₂ has type τ₂

10/2/14 28



Fun Rule

- Rules describe types, but also how the environment Γ may change
- Can only do what rule allows!
- fun rule:

$$\frac{\{x : \tau_1\} + \Gamma \mid -e : \tau_2}{\Gamma \mid -\text{fun } x -> e : \tau_1 \to \tau_2}$$

10/2/14

Fun Examples

$$\frac{\{y : int \} + \Gamma \mid -y + 3 : int}{\Gamma \mid -fun \ y -> y + 3 : int \rightarrow int}$$

$$\frac{\{f: \mathsf{int} \to \mathsf{bool}\} + \Gamma \mid -f \ 2 :: [\mathsf{true}] : \mathsf{bool} \ \mathsf{list}}{\Gamma \mid -(\mathsf{fun} \ f \ -> f \ 2 :: [\mathsf{true}])} \\ : (\mathsf{int} \to \mathsf{bool}) \to \mathsf{bool} \ \mathsf{list}$$



(Monomorphic) Let and Let Rec

let rule:

$$\frac{\Gamma \mid -e_1 : \tau_1 \quad \{x : \tau_1\} + \Gamma \mid -e_2 : \tau_2}{\Gamma \mid -(\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

let rec rule:

$$\frac{\{x: \tau_1\} + \Gamma \mid -e_1:\tau_1 \mid \{x: \tau_1\} + \Gamma \mid -e_2:\tau_2}{\Gamma \mid -(\text{let rec } x = e_1 \text{ in } e_2):\tau_2}$$

10/2/14 31



Which rule do we apply?

?
|- (let rec one = 1 :: one in let
$$x = 2$$
 in fun $y \rightarrow (x :: y :: one)$) : int \rightarrow int list

10/2/14 32



Example

10/2/14 33



Proof of 1

• Which rule?

{one : int list} |- (1 :: one) : int list

10/2/14 34



Proof of 1

Application

(3) (4)
{one : int list} |- {one : int list} |((::) 1): int list→ int list one : int list
{one : int list} |- (1 :: one) : int list

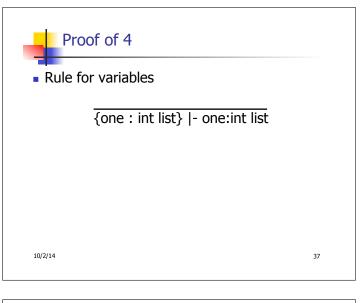
10/2/14 35

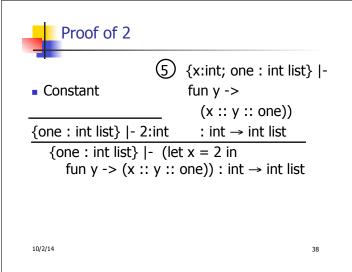


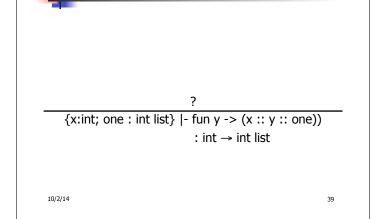
Proof of 3

Constants Rule Constants Rule

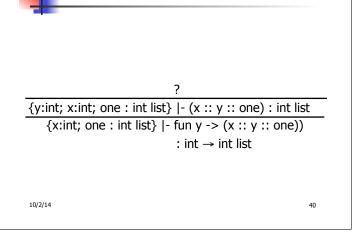
{one : int list} |- {one : int list} |- (::) : int \rightarrow int list \rightarrow int list \rightarrow int list \rightarrow int list \rightarrow int list



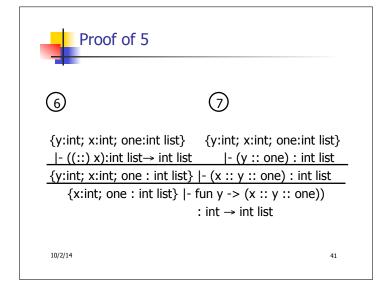


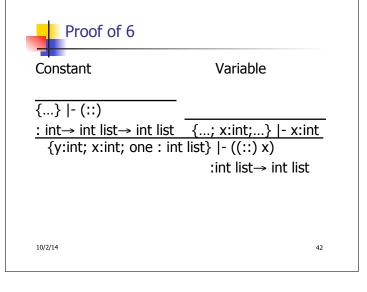


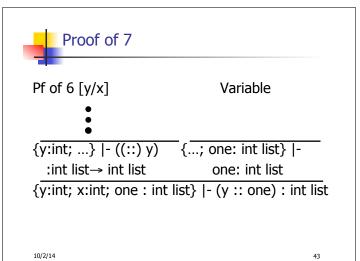
Proof of 5



Proof of 5









- Type Systems are logics; logics are type systems
- Types are propositions; propositions are types
- Terms are proofs; proofs are terms
- Function space arrow corresponds to implication; application corresponds to modus ponens

10/2/14 44



Curry - Howard Isomorphism

Modus Ponens

$$\frac{A \Rightarrow B \quad A}{B}$$

Application

$$\frac{\Gamma \mid -e_1 : \alpha \rightarrow \beta \quad \Gamma \mid -e_2 : \alpha}{\Gamma \mid -(e_1 e_2) : \beta}$$

10/2/14 45



Mia Copa

- The above system can't handle polymorphism as in OCAML
- No type variables in type language (only metavariable in the logic)
- Would need:
 - Object level type variables and some kind of type quantification
 - let and let rec rules to introduce polymorphism
 - Explicit rule to eliminate (instantiate) polymorphism