Programming Languages and Compilers (CS 421)



Grigore Rosu 2110 SC, UIUC

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

Slides by Elsa Gunter, based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

2/25/2014



Objective

- The objective of today's class is to discuss a formal type system for a simple functional language
- We will only discuss the monomorphic case today; polymorphism is covered next lecture
- We first briefly review material about types and various types of type checkers, to set the ground and motivation for a type system

2/25/2014



Why Data Types?



- 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

2/25/2014

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

2/25/2014



Types as Specifications



- Types describe properties
- Different type systems describe different properties, eq
 - 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

2/25/2014



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"

2/25/2014

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

2/25/2014



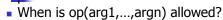
Static vs Dynamic Types



- 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

2/25/2014

Type Checking



- 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

2/25/2014

10



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



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

2/25/2014

2/25/2014



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)



Static Type Checking

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

2/25/2014

13

15



2/25/2014

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
 - Eg: array bounds

2/25/2014



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

2/25/2014



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)

2/25/2014

17



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

2/25/2014

18

14

Format of Type Judgments



• A *type judgement* has the form

$$\Gamma$$
 |- exp : τ

- Γ is a typing environment
 - Supplies the types of variables and functions
 - Γ is a list of the form $[x:\sigma,\ldots]$
- exp is a program expression
- τ is a type to be assigned to exp
- |- pronounced "turnstyle", or "entails" (or "satisfies")

2/25/2014 19



Types Systems as Proof Systems



Type systems are usually defined as proof systems, using proof derivation rules

Conclusion

read as: if I have proofs for Hypothesis 1, ..., Hypothesis n, then this rule allows me to construct a proof derivation for Conclusion

If n = 0 then rule is called "axiom"

22



Axioms - Constants



|-n: int (assuming n is an integer constant)

I- true: bool

I- false : bool

- These rules are true with any typing environment
- n is a meta-variable

2/25/2014

Axioms – Variables (Monomorphic Rule)

Notation: Let $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$ and there is no $x : \tau$ to the left of $x : \sigma$ in Γ

Variable axiom:

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

2/25/2014

21



Simple Rules - Arithmetic



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

$$\frac{\Gamma \mid -e_1:\tau \quad \Gamma \mid -e_2:\tau \quad (\oplus):\tau \to \tau \to \tau}{\Gamma \mid -e_1 \oplus e_2:\tau}$$

Relations (
$$\sim \in \{ <, >, =, <=, >= \}$$
):
$$\frac{\Gamma \mid -e_1 : \tau \qquad \Gamma \mid -e_2 : \tau}{\Gamma \mid -e_1 \sim e_2 : \text{bool}}$$

For the moment, think τ is int

2/25/2014



Simple Rules - Booleans

Connectives

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

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

2/25/2014



Type Variables in Rules



• If then else rule:

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

- 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

2/25/2014



Function Application



Application rule:

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

• If you have a function expression e_1 of type $\tau_1 \rightarrow \tau_2$ applied to an argument of type τ_1 , the resulting expression has type τ_2

2/25/2014



Fun Rule



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

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

2/25/2014

25

27

Fun Examples

$$\frac{[y: int] + \Gamma |- y + 3: int}{\Gamma |- fun y -> y + 3: int \rightarrow int}$$

$$\begin{array}{c} [f: \mathsf{int} \to \mathsf{bool}] + \Gamma \mid \mathsf{-f2} :: [\mathsf{true}] : \mathsf{bool} \ \mathsf{list} \\ \\ \Gamma \mid \mathsf{-} (\mathsf{fun} \ \mathsf{f} \, \mathsf{->f2} :: [\mathsf{true}]) \\ \\ : (\mathsf{int} \to \mathsf{bool}) \to \mathsf{bool} \ \mathsf{list} \end{array}$$

2/25/2014



(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:

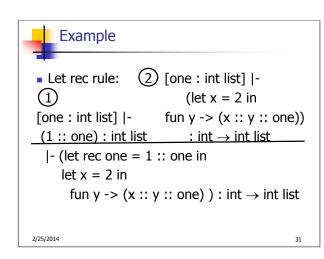
$$\frac{[x: \tau_1] + \Gamma \vdash e_1:\tau_1 \quad [x: \tau_1] + \Gamma \vdash e_2:\tau_2}{\Gamma \vdash (\text{let rec } x = e_1 \text{ in } e_2):\tau_2}$$

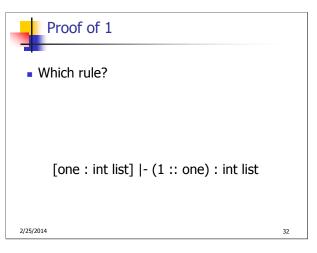
2/25/2014

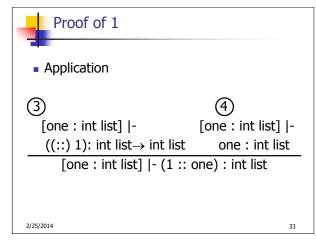
Example

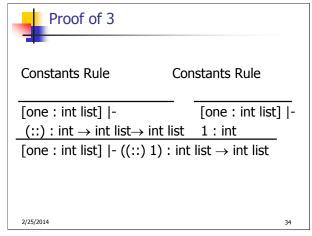
Which rule do we apply?

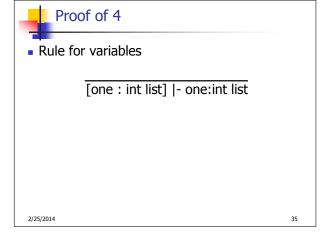
|- (let rec one = 1 :: one in
let
$$x = 2$$
 in
fun $y \rightarrow (x :: y :: one)$) : int \rightarrow int list

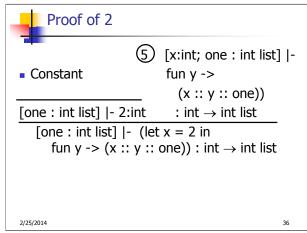


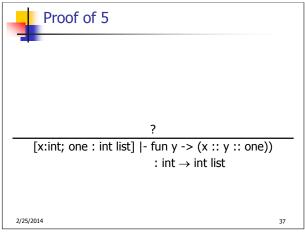


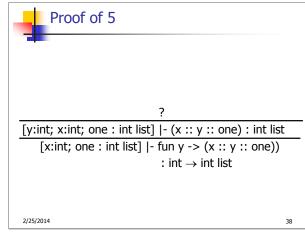


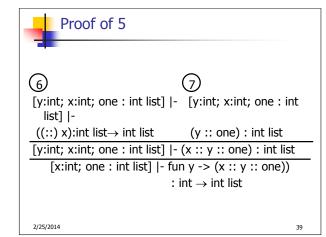


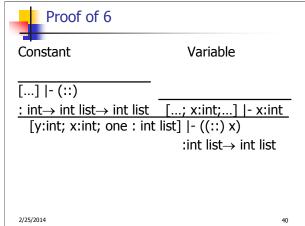


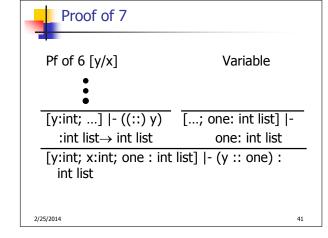


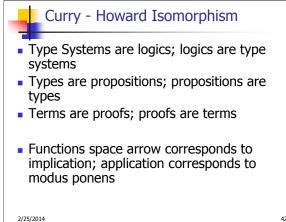














Curry - Howard Isomorphism

Modus Ponens

$$\begin{array}{c} \mathsf{A} \Rightarrow \mathsf{B} & \mathsf{A} \\ \hline \mathsf{B} & \end{array}$$

Application

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

2/25/2014 43



Mia Copa

- The above system can't handle polymorphism as in OCAML
- No type variables in type language (only meta-variable 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

2/25/2014