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

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

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

Terminology

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

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

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”

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

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

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

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

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

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

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

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)

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

Format of Type Judgments

- A *type judgement* has the form

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

Types Systems as Proof Systems

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

$$\frac{\text{Hypothesis 1} \quad \dots \quad \text{Hypothesis n}}{\text{Conclusion}}$$

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"

Axioms - Constants

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

$$\frac{}{\text{|- true : bool}} \qquad \frac{}{\text{|- false : bool}}$$

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

Axioms – Variables (Monomorphic Rule)

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

Variable axiom:

$$\frac{}{\Gamma \text{ |- } x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$

Simple Rules - Arithmetic

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

$$\frac{\Gamma \text{ |- } e_1 : \tau \qquad \Gamma \text{ |- } e_2 : \tau \qquad (\oplus) : \tau \rightarrow \tau \rightarrow \tau}{\Gamma \text{ |- } e_1 \oplus e_2 : \tau}$$

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

$$\frac{\Gamma \text{ |- } e_1 : \tau \qquad \Gamma \text{ |- } e_2 : \tau}{\Gamma \text{ |- } e_1 \sim e_2 : \text{bool}}$$

For the moment, think τ is int

Simple Rules - Booleans

Connectives

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

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

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

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 e_1 of type $\tau_1 \rightarrow \tau_2$ applied to an argument of type τ_1, the resulting expression has type τ_2

Fun Rule

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

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

Fun Examples

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

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

(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:

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

Example

- Which rule do we apply?

$$\frac{?}{\mid\!- (\text{let rec one = 1 :: one in let x = 2 in fun y -> (x :: y :: one)}) : \text{int} \rightarrow \text{int list}}$$

Example

- Let rec rule: (2) [one : int list] |- (let x = 2 in fun y -> (x :: y :: one)) : int → int list

(1) [one : int list] |- (1 :: one) : int list

―――――――――――――――――――――――――――――

|- (let rec one = 1 :: one in
let x = 2 in
fun y -> (x :: y :: one)) : int → int list

Proof of 1

- Which rule?

[one : int list] |- (1 :: one) : int list

Proof of 1

- Application

(3) [one : int list] |- ((::) 1): int list→ int list

(4) [one : int list] |- one : int list

―――――――――――――――――――――――――――――

[one : int list] |- (1 :: one) : int list

Proof of 3

Constants Rule

―――――――――――――

[one : int list] |- (::) : int → int list→ int list

Constants Rule

―――――――――――――

[one : int list] |- 1 : int

―――――――――――――――――――――――――――――

[one : int list] |- ((::) 1) : int list → int list

Proof of 4

- Rule for variables

―――――――――――――

[one : int list] |- one:int list

Proof of 2

- Constant

―――――――――――――

[one : int list] |- 2:int

(5) [x:int; one : int list] |- fun y -> (x :: y :: one)) : int → int list

―――――――――――――――――――――――――――――

[one : int list] |- (let x = 2 in
fun y -> (x :: y :: one)) : int → int list

Proof of 5

?

[x:int; one : int list] |- fun y -> (x :: y :: one))
: int → int list

Proof of 5

?

[y:int; x:int; one : int list] |- (x :: y :: one) : int list

[x:int; one : int list] |- fun y -> (x :: y :: one))
: int → int list

Proof of 5

(6) [y:int; x:int; one : int list] |- ((::) x):int list→ int list

(7) [y:int; x:int; one : int list] |- (y :: one) : int list

[y:int; x:int; one : int list] |- (x :: y :: one) : int list

[x:int; one : int list] |- fun y -> (x :: y :: one))
: int → int list

Proof of 6

Constant: [...] |- (::) : int→ int list→ int list

Variable: [...; x:int;...] |- x:int

[y:int; x:int; one : int list] |- ((::) x)
:int list→ int list

Proof of 7

Pf of 6 [y/x]
⋮
[y:int; ...] |- ((::) y) :int list→ int list

Variable:
[...; one: int list] |- one: int list

[y:int; x:int; one : int list] |- (y :: one) : int list

Curry - Howard Isomorphism

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

Curry - Howard Isomorphism

- Modus Ponens

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

- Application

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

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