Type checking

Jianguo Lu

November 27, 2014

slides adapted from Sean Treichler and Alex Aiken's

Outline

1. Language translation
2. Type checking
3. optimization

Semantic analysis

Ensure that the program has a well-defined meaning.

- Verify properties of the program that aren't caught during the earlier phases:
  - Variables are declared before they're used.
  - Expressions have the right types.
  - Arrays can only be instantiated with NewArray.
  - Classes don't inherit from nonexistent base classes
  - ...
- Once we finish semantic analysis, we know that the user's input program is legal.

Types of Semantic Checks

- **Type checks:** operator applied to incompatible operands?
- **Flow of control checks**: break (outside while?)
- **Uniqueness checks**: labels in case statements
- ...

Challenges in Semantic Analysis

Reject the largest number of incorrect programs.
Accept the largest number of correct programs.

Type checking

the activity of ensuring that the operands of an operator are of compatible types
Problem: Verify that a type of a construct matches that expected by its context.

- Examples
  - mod requires integer operands (PASCAL)
  - (dereferencing): applied to a pointer
  - $a[i]$: indexing applied to an array
  - function applied to correct arguments.
- Compatible: either legal for the operator
- type error: application of an operator to an operand of an inappropriate type

type checking and type inference

- inference: infer the types not explicitly decalred
- checking: prove the type if correct

type checking approaches

static typing

- at compile time, analyze possible input for each operation
- prove that these inputs are ok

dynamic typing

- before the execution of the operation, check whether the input is acceptable
- raise run-time exception if not (not a desirable feature)

Type system

a formal system to generate proofs

- WFF

$$\vdash e : T$$

$$E \vdash e : T$$

  type expressions
- some axioms to give types of literals, ...
- inference rules to determine types of complex expressions

soundness and completeness

Two desirable properties of a type system (and any formal system)
But not automatically guaranteed!

Soundness

all derived judgements are true statements.

- type system models run time behaviour
- whenever $\vdash e : T$, e evaluates to a value of T.

Completeness

all true statements can be derived by the formal system

- any property that holds for all possible executions can be described in the type system

axioms and inference rules

example of axioms

- "2 is a integer" $\vdash 2 : int$
- "3 is a integer" $\vdash 3 : int$

example of rules

- if 2 and 3 are integers, 2+3 is an integer
-

$$\frac{\vdash 2 : int;\ \ \vdash 3 : int}{\vdash 2 + 3 : int} \tag{1}$$

universal qunatification

- for $\forall$ x and y, x and y are integers, x+y is an integer
-

$$\frac{\vdash x : int;\ \ \vdash y : int}{\vdash x + y : int} \tag{2}$$

rules and a proof

rules

$$\frac{\text{x is an integer literal}}{\vdash x : int} \quad \text{(T-int)}$$

$$\frac{\begin{array}{c}\vdash e1 : int \\ \vdash e2 : int\end{array}}{\vdash e1 + e2 : int} \quad \text{(T-add)}$$

a simple proof

$$\frac{\dfrac{\text{2 is an integer literal}}{\vdash 2 : int} \qquad \dfrac{\text{3 is an integer literal}}{\vdash 3 : int}}{\vdash 2 + 3 : int} \quad (3)$$

rules for boolean

some boolean type rules

$$\frac{\vdash e : bool}{\vdash !e : bool} \quad \text{(T-not)}$$

$$\frac{\begin{array}{c}\vdash e1 : int \\ \vdash e2 : int \\ op \in \{\leq, \geq\}\end{array}}{\vdash e1\ op\ e2 : bool} \quad \text{(T-compare)}$$

environment

type of a variable

$$\frac{\text{x is an object identifier}}{\vdash x : ??}$$

- the local structure of the program does not give enough information
- need to add global info to the environment
- all rules can have environment, even if they do not use environment info

environment examples

$$\frac{O(x) = T}{O \vdash x : T} \quad \text{(T-var)}$$

$$\frac{O \vdash e1 : int \quad O \vdash e2 : int}{O \vdash e1 + e2 : int}$$

subtyping

what is the type of x in an assignment statement?

$$x \leftarrow e; \tag{4}$$

a rule for assignment stmt

$$\frac{\begin{array}{c} O(x) = T \\ O \vdash e : T \end{array}}{O \vdash x \leftarrow e : T}$$

what is there is a subtyping?

subtypes

axioms for classes

$$\frac{}{\vdash C \leq C} \quad \text{(reflexitivity)}$$

$$\frac{}{\vdash C \leq parent(C)} \quad \text{(inheritance)}$$

$$\frac{\vdash C1 \leq C2 \quad \vdash C2 \leq C3}{\vdash C1 \leq C3} \quad \text{(transitivity)}$$

rule for subtype

subtype rule

$$\frac{O(x) = T1 \quad O \vdash e : T2 \quad T2 \leq T1}{O \vdash x \leftarrow e : T2} \quad ()$$

if statement/expression

a simple rule

$$\frac{O \vdash \quad e1 : bool \quad O \vdash \quad e2 : T \quad O \vdash \quad e3 : T}{O \vdash (e1)?e2 : e3 \ : T} \ ()$$

what if e2 and e3 have different types?

if with least upper bound

$$\frac{O \vdash \quad e1 : bool \quad O \vdash \quad e2 : T1 \quad O \vdash \quad e3 : T2}{O \vdash (e1)?e2 : e3 \ : T1 \sqcup T2} \ ()$$

$T1 \sqcup T2$: the closest common ancestor in the class hierarchy

definition of least upper bound

least upper bound

$$\frac{\begin{array}{c} C1 \leq C3 \\ C2 \leq C3 \\ \forall C4.C1 \leq C4 \wedge C2 \leq C4 \rightarrow C3 \leq C4 \end{array}}{T1 \sqcup T2 = C3} \quad ()$$

method call

$$\frac{\begin{array}{c} O \vdash e0 : T0 \\ O \vdash e1 : T1 \\ \cdots \\ O \vdash en : Tn \end{array}}{O \vdash e0.f(e1, \ldots, en) \quad :??} \qquad ()$$

- need to track the types of methods
- function M maps a method to its type (input types and return type)

$$M(C, f) = (T1, T2, \ldots, Tn, Tr) \tag{5}$$

$$f : T1 \times T2 \cdots \times Tn \rightarrow Tr \tag{6}$$

method call

$$\frac{\begin{array}{c} M(T0, f) = (T1', \ldots, Tn', Tr) \\ \forall i \in \{1, ;n\}. Ti \leq Ti' \\ O \vdash e0 : T0 \\ O \vdash e1 : T1 \\ \ldots \\ O \vdash en : Tn \end{array}}{O \vdash e0.f(e1, \ldots, en) \ : \ Tr} \quad ()$$

- need to track the types of methods
- function M maps a method to its type (input types and return type)

$$M(C, f) = (T1, T2, \ldots, Tn, Tr) \quad (7)$$

$$f : T1 \times T2 \cdots \times Tn \rightarrow Tr) \quad (8)$$

optimization

- Want to rewrite code so that it's:
  - faster, smaller, consumes less power, etc.
  - while retaining the "observable behavior"
  - usually: input/output behavior
  - often need analysis to determine that a given
  - optimization preserves behavior.
  - often need profile information to determine that a given optimization is actually an improvement.
- Often have two flavors of optimization:
  - high-level: e.g., at the AST-level (e.g., inlining)
  - low-level: e.g., right before instruction selection (e.g., register allocation)

peephole optimization

- Take a sequence of existing source code or assembly instructions
- Look at small windows (peepholes) of a few instructions at a time
- Match against patterns that can be replaced by 'better' sequences of instructions
- Repeat until 'no' peepholes' match patterns

algebraic optimizations

Constant folding (delta reductions)

- 3+4 ==> 7, x*1 ==> x
- if true then s else t ==> s

Strength reduction

x*2 ==> x+x, x div 8 ==> x » 3

Inlining, constant propagation, copy propagation, dead-code elimination, etc (beta reduction)

let val x = 3 in x + x end ==> 3 + 3

Common sub-expression elimination (beta expansion)

(length x) + (length x) ==> let val i = length x in i+i end

more examples

Loop invariant removal:

```
for (i=0; i<n; i+=s*10)

==>

int t = s*10;
for (i=0;i<n;i+=t)
```

Loop interchange:

```
for (i=0; i<n; i++)
for (j=0; j<n; j++)
s += A[j][i];


for (j=0; j<n; j++)
for (i=0; i<n; i++)
s += A[j][i];
```