COMP 181

Lecture 11

Static checking

October 10, 2006

Last time

- Syntax-directed translation
  - *Attribute grammars*
    - Formal system for performing systematic computations during parsing
    - Complex evaluation issues
    - Not widely used
  - *Ad-hoc translation*
    - Add code to productions
    - Evaluation must follow parsing scheme
    - Used by most parser generators

Adding actions

- ***L-attributed*** definition
  - Use values from parent and siblings
  - For production $\mathbf{A} \rightarrow \mathbf{X_1\ X_2 \ldots X_n}$
  - Each attribute of X_i depends on
    - Attributes of $X_1\ X_2 \ldots X_{i-1}$, and
    - Inherited attributes of A
- Suited to LL parsing
  - Evaluate in a single top-down pass (left to right)
  - Pass values down through recursive descent
  - Table driven: store intermediate values on stack

Adding actions

- ***S-attributed*** definition
  - All attributes are synthesized
  - For production $\mathbf{A} \rightarrow \mathbf{X_1\ X_2 \ldots X_n}$
  - Value of **A** is computed as a function of the attributes already computed for $\mathbf{X_1\ X_2 \ldots X_n}$
- Suited to LR parsing
  - Can be computed in a single bottom-up pass
  - Associate pieces of code with each production
  - At each reduction, the code is executed

Example

Name for the value associaed with this production

Arbitrary code between {: and :}

```
expr_part ::= expr SEMI ;

expr ::= expr:e1 PLUS expr:e2
         {: RESULT = new Integer(e1.intValue() +
                                 e2.intValue()); :}
       | expr:e1 MINUS expr:e2
         {: RESULT = new Integer(e1.intValue() -
                                 e2.intValue()); :}
         ...
       | LPAREN expr:e RPAREN
         {: RESULT = e; :}
       | NUMBER:n
         {: RESULT = n; :}
       ;
```

RESULT refers to the attribute of the LHS non-terminal

Another example

- Build an abstract syntax tree

```
expr_part ::= expr SEMI ;

expr ::= expr:e1 PLUS expr:e2
         {: RESULT = new AddNode(e1, e2); :}
       | expr:e1 MINUS expr:e2
         {: RESULT = new SubNode(e1, e2); :}
         ...
       | LPAREN expr:e RPAREN
         {: RESULT = e; :}
       | NUMBER:n
         {: RESULT = new NumberNode(n); :}
       ;
```

Implementation

- How does this work?
  - Where are the attributes stored?
  - What do e1, e2, n, RESULT refer to?
- **Key**: store attributes on stack

  At a reduction of $A \rightarrow \beta$
  - Pop 3 × |β| symbols – 1 symbol, 1 state, **1 value**
  - Map values to names:
    - expr:**e1** PLUS expr:**e2**
    - **e2** = top of stack, then PLUS, then **e1**
  - Invoke action code on values – store in RESULT
  - Push RESULT back on stack with new symbol, state

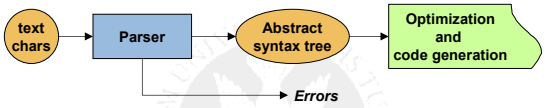
Prelude

- What is this spacecraft?

  Mars Climate Orbiter
- What happened to it?
  - After 268 day journey, it attempted to enter into Mars orbit
  - Passed too close, crashed
- Why?

  Engineering team at Lockheed using ***English units***, NASA using ***metric units***

Where are we…



- Parsing complete
  - Syntax is correct
  - Built an internal representation

    (usually an abstract syntax tree)
  - Now what?

Beyond syntax

- What's wrong with this code?

  (Note: it parses perfectly)

```
foo(int a, char * s){ … }

int bar() {
  int f[3];
  int i, j, k;
  char *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 5;
  j = i + k;
  printf("%s,%s.\n",p,q);
  goto label23;
}
```


Errors

- Undeclared identifier
- Multiply declared identifier
- Index out of bounds
- Wrong number or types of args to call
- Incompatible types for operation
- Break statement outside switch/loop
- Goto with no label

Kinds of checks

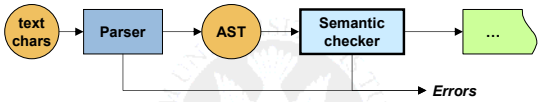
- Uniqueness checks
  - Certain names must be unique
  - Many languages require variable declarations
- Flow-of-control checks
  - Match control-flow operators with structures
  - Example: break applies to innermost loop/switch
- Type checks
  - Check compatibility of operators and operands

Program checking

Why do we care?

- Obvious:
  - Report mistakes to programmer
  - Avoid bugs: *f[6] will cause a run-time failure*
  - Help programmer verify intent
- How do these checks help compiler?
  - Allocate right amount of space for variables
  - Select right machine operations
  - Proper implementation of control structures

Program checking



- ***Semantic*** checking
  - Beyond syntax: hard to express directly in grammar
  - Requires extra computation, extra data structures
  - **Goals**:
    - Better error checking – "deeper"
    - Give back-end everything it needs to generate code

Program checking

When are checks performed?

- ***Static*** checking
  - At compile-time
  - Detect and report errors by analyzing the program
- ***Dynamic*** checking
  - At run-time
  - Detect and handle errors as they occur
- What are the pros and cons?
  - Efficiency? Completeness? Developer vs user experience? Language flexibility?

Uniqueness checks

- Based on symbol tables
  - Add entries to symbol table
  - Check references against table
  - Example: declarations
    - Add each declaration to table
    - Make sure entries are unique
    - Check that variable uses refer to entries in table
- Closely tied to notion of ***scope***

  We'll address these topics when we discuss procedures

Type checking

- Today:
  - Focus on types
  - Static type checking
- As programmers…
  - We have an intuitive notion of types
  - What is a type?

Types

- From *Types and Programming Languages*

  "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."
- **Idea**
  - Divide possible values into groups – types
  - Disallow certain behaviors based on membership

Types and compilers

- Compiler must understand the type system of the language
  - Enforce limitations
  - Implement checks and conversions
  - Provide concrete representation – bits
- Responsibilities vary by language
  - C/C++ – all type checking occurs in compiler
  - Java – mixed model
  - Perl, Ruby – no compiler, all dynamic checking

Type systems

From language specifications:

"The result of a unary & operator is a pointer to the object referred to by the operand. If the type of the operand is "T", the type of the result is "pointer to T".

"If both operands of the arithmetic operators addition, subtraction and multiplication are integers, then the result is an integer"

Properties of types

What do these excerpts imply?

- Types have structure
  "Pointer to T" and "Array of Pointer to T"
- Expressions have types
  Types are derived from operands by rules
- Goal: determine types for all parts of a program

Type expressions

(Not to be confused with types of expressions)

- Build a description of a type from:
  - Basic types – also called "primitive types"
    Vary between languages: int, char, float, double
  - Type constructors
    Functions over types that build more complex types
  - Type variables
    Unspecified parts of a type – polymorphism
  - Type names
    An "alias" for a type expression – `typedef` *in C*

Type constructors

- Arrays
  - If T is a type, then *array*(T) is a type denoting an array with elements of type T
  - May have a size component: *array*(I,T)
- Products and records
  - If T_1 and T_2 are types, then $T_1 \times T_2$ is a type denoting pairs of two types
  - May have labels for records/structs
    ("name", char *) × ("age", int)

Type constructors

- Pointers
  - If T is a type, the *pointer*(T) denotes a pointer to T
- Functions or function ***signatures***
  - If D and R are types then D → R is a type denoting a function from domain type D to range type R
  - For multiple inputs, domain is a product
  - Notice: primitive operations have signatures
    Mod % operator: int × int → int

Type checking

- Define language ***type system***
  - Set of rules for assigning type expressions to the parts of a program
- Implemented by ***type checker***
  - Derives types using rules
  - Static (compile-time) or dynamic (run-time)
- Type checking may fail
  - Handling errors depends on specific constructs

Example

- Static type checker for C
- Assume:

  We can get declared types of identifiers, functions
- Rules:

Expression	*Type rule*
E_1 [E_2]	**if type(E_2) is int and type(E_1) is *array*(T) result type is T else error**
* E	**if type(E) is *pointer*(T) result type is T else error**

Example

- What about function calls?
  - Consider single argument case

Expression	*Type rule*
E_1 (E_2)	**if type(E_1) is D → R and type(E_2) is D result type is R else error**

  - Extends to multiple arguments (products)
- What is fundamental operation?

  "If two type expressions are equivalent then…"

Type equivalence

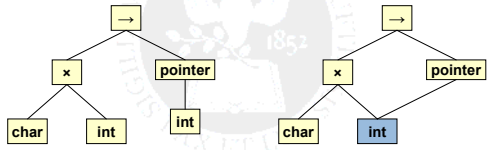
- Implementation: ***structural equivalence***
  - Same basic types
  - Same set of constructors applied
- Recursive test:

```
function equiv(s, t)
    if s and t are the same basic type
        return true
    if s = pointer(s1) and t = pointer(t1)
        return equiv(s1,t1)
    if s = s1×s2 and t = t1×t2
        return equiv(s1,t1) && equiv(s2,t2)
…etc…
```

Efficiency is critical

Representation

- Represent types as graphs
  - Node for each type
  - Often created as a DAG for efficiency



Function: (char × int) → int *

Structural equivalence

- Efficient implementation
  - Recursively descend tree, compare basic types
  - Recursively descend DAG until common node

    (Same type always represented by same node)
- Many subtle variations in practice
  - Special rules for parameter passing
    - C: array T[] is compatible with T*
    - Pascal, Fortran: leaving off size of array
  - Type qualifiers: `const`, `static`, etc.

Notions of equivalence

- Different way of handling type names
- Structural equivalence
  - Ignores type names
  - `typedef int * numptr` → `numptr ≡ int *`
  - Not always desirable
- ***Name equivalence***
  - Types are equivalent if they have the same name
  - Solves an important problem: recursive types

Recursive types

- Cycle in the type graph:

```
struct cell {
    int info;
    struct cell * next;
}
```

- C uses structural equivalence for everything except structs
  - The name "struct cell" is used instead of checking the actual fields in the struct
  - Can we have two compatible struct definitions?

Java types

- Type equivalence for Java

```
class Foo {          class Bar {
  int x;               int w;
  float y;             float z;
}                    }
```

- Can we pass Bar objects to a method taking a type Foo?
  - No
  - Java uses name equivalence for classes
  - What can we do in C?

Type checking

- Consider this case:
  What is the type of **x+i** if **x** is `float` and **i** is `int`
- Is this an error?
- Compiler fixes the problem
  - Convert into compatible types
  - Automatic conversions are called ***coercions***
  - Rules can be complex
    - in C, large set of rules for called ***integral promotions***
    - Goal is to preserve information

Type coercions

- Rules
  - Find a common type
  - Add explicit conversion into the AST

Expression	*Type rule*
$E_1 + E_2$	if type(E_1) is int and type(E_2) is int result type is int if type(E_1) is int and type(E_2) is float result type is float if type(E_1) is float and type(E_2) is int result type is float …etc…

Implementing type checkers

Expression	*Type rule*
$E \rightarrow E_1 [E_2]$	if type(E_2) is int and type(E_1) is *array*(T) type(E) = T else error
$E \rightarrow * E$	if type(E) is *pointer*(T) type(E) is T else error

- Does this form look familiar?
  - Type checking fits into syntax-directed translation
  - One reason why declarations precede stmts

Expressions

- Why compute types of all expressions?
  - Example: x + y + z *what is type of (x+y)?*
- Already mentioned
  - Check for correctness
  - Add coercions
- Code generation
  - Most machines cannot add 3 values
  - Must break up the expression
  - Generate storage for intermediate results

Interesting cases

- What about printf?
  - printf(const char * format, …)
  - Implemented with varargs
  - Format specifies which arguments should follow
  - Who checks?
- Array bounds
  - Array sizes rarely provided in declaration
  - Cannot check statically

Overloading

- "+" operator
  - Same syntax, multiple implementations
  - C: `float` versus `int`
  - C++: arbitrary user implementation
- How to decide which one?
  - Use types of the operands
  - Find operator with the right type signature
- How does this interact with coercions?

Object oriented types

```
class  foo  {    ….    }
class  bar  extends  foo  {   …   }
```

- What is relationship between `foo` and `bar`?
  - `bar` is a ***subtype*** of `foo`
  - Any code that accepts a `foo` object can also accept a `bar` object
- Modify type compatibility rules
  - Formal parameter can accept any subtype
  - Same holds for assignment

Polymorphism

- Ordinary procedures
  - Accept fixed type signature
  - Example: `search(char c, string s)`
- Generic procedures
  - Work on arguments of different types
- User-defined generics
  - Define interface with type variables
  - Example: `search(El c, El [] list)`

Polymorphism

How is polymorphism implemented?

- Type erasure
  - Convert all type variables into Object
  - Java approach
  - Pros and cons?

Pro: backward compatible with JVM
Con: cannot support primitives

- Instantiation
  - Generate a new implementation for each combination of type arguments
  - C++ approach
  - Pros and cons?

Pro: works with any types
Con: possible code bloat

Type checking polymorphism

- Problem:

  How do we check generics statically?

```
int search(El x, El [] list) {
  for (int i = 0; i < list.length; i++)
    if (x.compareTo(list[i])) return i;
  return -1;
}
...
search(5, A);
```

  - What do we need to know?
    - Does El have a method "compareTo"?
    - Is A an array of type int?

Type checking polymorphism

- Checking generic code:
  - C++ templates
    - Checked at instantiation
    - Plug in the type variables
    - Compile and check resulting code
  - Java generics
    - Constrain type variables
    - Declare "type El must implement interface X"
    - Can check generic code independent of use

Type checking polymorphism

- Checking uses of generics
  - Is `search(5, A)` correct? *(Let's say int A[20];)*
  - Find a mapping from 5 and A to El and El []
  - Mapping: El is int
  - This process is called ***unification***
- Unification
  - Given two type expressions, one with type variables
  - Find a substitution of type variables that turns it into the other type expression
  - Used in many other areas: theorem proving, Prolog

Template metaprogramming

- C++ templates are very powerful

```
template<int N>
class Factorial {
public:
    enum { value = N * Factorial<N-1>::value };
};

class Factorial<1> {
public:
    enum { value = 1 };
};

x = Factorial<8>::value;
```


More…

```
template<bool C> class IfThen { };

class IfThen<true> {
public:
    static inline void do() { statement1; }    // true case
};

class IfThen<false> {
public:
    static inline void do() { statement2; }    // false case
};

// Replacement for 'if/else' statement:
IfThen<condition>::do();
```


Type inference

- Languages without declarations
  - ML, Haskell
  - Still statically typed
  - Types determined by use
- Requires ***type inference*** algorithm
  - Determine constraints from program
  - Results in type expressions with type variables
  - Compute a consistent assignment to variables

Back to Mars Orbiter

- Language support for units
  - Idea: make units a part of the type
  - Use polymorphism for operators
  - Type-check the computations
- Example:

```
double<kg> weight;
double<s> time;
double<m> distance;
double<m s^-2> gravity = 9.8 * m/(s * s);
double<kg m s^-2> force = weight * gravity;
```

- Issues:
  - Type checker must understand algebra
  - Generics are tricky *(what is the type of sqrt?)*

Next time…

- The procedure abstraction
  - Symbol tables and scopes
  - Run-time environments
  - Storage allocation, stacks
- *Coming soon*: new programming assignment

Generics in Java

- New in Java 1.5
  - Type variables, like C++ templates
  - Not as powerful (on purpose)

```
boolean search(T el, List<T> list)
```

- Question:
  - How is this different from:

```
boolean search(Object el, List list)
```

(Where list is a linked list of Object references)