Challenges for General Attribute Systems

- An evaluation strategy can only exist if for any abstract syntax tree, the dependencies between attributes are acyclic.
- Checking that no cyclic attribute dependencies can arise is DEXPTIME-complete [Jazayeri, Odgen, Rounds, 1975].

Idea: Compute a set of dependency graphs for each symbol $s \in T \cup N$.

- Initialize $G(s) = \emptyset$ for each $s \in T$ and set $S(s) = \{G_s\}$ for each $s \in T$ where $G_s$ is the dependency graph of $s$.
- For each rule $s := s_1 \ldots s_n$ of the non-terminal $s \in N$ with RHS $s_1 \ldots s_n$, extend $G(s)$ with graphs obtained by embedding the dependency graphs $G(s_1), \ldots, G(s_n)$ into the child attributes of the dependency graph of that rule.

Computing Dependencies

Example: Given the grammar $S := a | b$ with these dependencies:

\[ G(S) = \emptyset, \ G(a) = \{ k[0] \rightarrow j[0] \}, \text{ and } G(b) = \{ i[0] \rightarrow h[0] \}. \]
Computing Dependencies

Example: Given the grammar $S \ ::= \ a \mid b$ with these dependencies:

Start with $G(S) = \emptyset$, $G(a) = \{k[0] \rightarrow j[0]\}$, and $G(b) = \{i[0] \rightarrow h[0]\}$.

For rule $S \ ::= \ a$, embed $G(a)$ into the child attributes of rule $S \ ::= \ a$,
yielding

$$G'(S) = \{h[1] \rightarrow h[0], b[1] \rightarrow k[1], j[1] \rightarrow i[1], j[1] \rightarrow j[0], k[1] \rightarrow j[l]\}$$

Computing Dependencies (cont’d)

Result so far:

$$G'(S) = \{h[1] \rightarrow h[0], b[1] \rightarrow k[1], j[1] \rightarrow i[1], j[1] \rightarrow j[0], k[1] \rightarrow j[l]\}$$

Given rule $S \ ::= \ b$, embed $G(b)$ into the child attributes of rule $S \ ::= \ a$,
yielding

$$G''(S) = G'(S) \cup \{h[1] \rightarrow h[0], b[1] \rightarrow k[1], j[1] \rightarrow i[1], j[1] \rightarrow j[0], i[l] \rightarrow h[l]\}$$

None of the graphs in $G''$ contain a cycle $\sim$ every derivable abstract syntax tree can be evaluated.
Dependencies for Recursive Rules

Problem: our approach fails for grammar $S ::= T | a | b, T ::= S$ with

$$G(T) = \{ h[0] = j[1], j[0] = h[1], i[1] = k[0], k[1] = i[0] \}$$

Consider inserting $G(T)$ into the initial $G(S)$:

- projection ensures finiteness of graphs

$T ::= S$

Dependencies for Recursive Rules

Problems for our approach fail for grammar $S ::= T | a | b, T ::= S$ with

$$G(T) = \{ h[0] = j[1], j[0] = h[1], i[1] = k[0], k[1] = i[0] \}$$

Consider inserting $G(T)$ into the initial $G(S)$:

- projection ensures finiteness of graphs
- maximum number of graphs for $S \in T \cup N$ and $n$ attributes is
  - there are $2 \cdot (\binom{n}{2}) = n(n-1)$ possible directed edges the dependency graph of $S$
  - since $G(S)$ is a set, it contains at most $2^{n(n-1)}$ graphs
Strongly Acyclic Attribute Dependencies

**Problem:** with larger grammars, this algorithm is too expensive

**Goal:** find a *sufficient* condition for an attribute system to be acyclic.

**Idea:** Compute a *single* dependency graph for each symbol $s \in N$.

- Initialise $G(s)$ with the local dependency graph of $s \in N \cup T$.
- For each rule $s := s_1 \cdots s_n$ of $s$,
- embed the graph $G(s_i)$ at the $i$-th position by
  - projecting the edges of $G(s_i)$ onto $a(0) \ldots z(0)$
  - add these edges to $G(s)$ as edges over $a[i] \ldots z[f]$
- if the new $G(s)$ contains a cycle, report "may have cycle"
- re-evaluate each rule until none of the graphs change anymore

From Dependencies to Evaluation Strategies

**Possible strategies:**
From Dependencies to Evaluation Strategies

Possible strategies:

1. let the user define the evaluation order
2. compute a strategy based on the dependencies:
   - compute a linear order from the partial order defined by $G(v_i)$
     - if the set of dependence graphs is used, compute a different
       linearization depending on the children
     - evaluate the attributes in the sequence indicated by the linear order
   - Example: regular expression attribute grammar:
     - in each $G(v_i)$, we can add the following edges:
       - $e[i] \rightarrow a[i]$
       - $e[i] \rightarrow f[i]$
       - $f[i] \rightarrow a[i]$
     - any linearization now allows the following strategy: traverse AST trice, each visit computing one of $e, f, g$
3. consider a fixed strategy and only allow an attribute system that can be evaluated using this strategy

Question: What are good linearizations?

Linear Order from Dependency Partial Order

Possible automatic strategies:

1. demand-driven evaluation
   - start with the evaluation of any required attribute
   - if the equation for this attribute relies on as-yet unevaluated attributes, compute these recursively
   - visits the nodes of the syntax tree on demand
   - (following a dependency on the parent requires a pointer to the parent)
Linear Order from Dependency Partial Order

Possible automatic strategies:

- **demand-driven evaluation**
  - start with the evaluation of any required attribute
  - if the equation for this attribute relies on as-of-yet unevaluated attributes, compute these recursively
  - \(\sim\) visits the nodes of the syntax tree on demand
  - (following a dependency on the parent requires a pointer to the parent)

- **evaluation in passes**
  - minimise the number of visits to each node
  - organise the evaluation of the tree in passes
  - for each pass, pre-compute a strategy to visit the nodes together
    with a local strategy for evaluation within each node type

---

Example for Demand-Driven Evaluation

Compute `next` at the leaves of \(a(a|b)\) in the expression \((a|b)*a(a|b)\):

\[
\begin{align*}
\begin{array}{l}
\text{\}: \quad \text{next}[1] & := \text{next}[0] \\
\quad \text{next}[2] & := \text{next}[0]
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
\text{\}: \quad \text{next}[1] & := \text{first}[2] \cup (\text{empty}[2] ? \text{next}[0]; \emptyset) \\
\quad \text{next}[2] & := \text{next}[0]
\end{array}
\end{align*}
\]

---

Example for Demand-Driven Evaluation

Compute `next` at the leaves of \(a(a|b)\) in the expression \((a|b)*a(a|b)\):

\[
\begin{align*}
\begin{array}{l}
\text{\}: \quad \text{next}[1] & := \text{next}[0] \\
\quad \text{next}[2] & := \text{next}[0]
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
\text{\}: \quad \text{next}[1] & := \text{first}[2] \cup (\text{empty}[2] ? \text{next}[0]; \emptyset) \\
\quad \text{next}[2] & := \text{next}[0]
\end{array}
\end{align*}
\]
Example for Demand-Driven Evaluation

Compute \texttt{next} at the leaves of \((a|b|)\) in the expressionn \(((a|b|)\ast a(a|b|)):\n
\[
\begin{align*}
\texttt{next}[1] & := \texttt{next}[0] \\
\texttt{next}[2] & := \texttt{next}[0]
\end{align*}
\]
\[
\begin{align*}
\texttt{next}[1] & := \texttt{first}[2] \cup (\texttt{empty}[2] \ ? \ \texttt{next}[0]) \\
\texttt{next}[2] & := \texttt{next}[0]
\end{align*}
\]

Demand-Driven Evaluation

Observations

- only required attributes are evaluated
- the evaluation sequence depends – in general – on the actual syntax tree
- the algorithm must track which attributes it has already evaluated
- the algorithm may visit nodes more often than necessary
- each node must contain a pointer to its parent
- the algorithm is not local

Observations

- only required attributes are evaluated
- the evaluation sequence depends – in general – on the actual syntax tree
- the algorithm must track which attributes it has already evaluated
- the algorithm may visit nodes more often than necessary
- each node must contain a pointer to its parent
- the algorithm is not local

approach only beneficial in principle:
- evaluation strategy is dynamic: difficult to debug
- computation of all attributes is often cheaper
- usually all attributes in all nodes are required

\[\leadsto\] perform evaluation in \textit{passes}
Evaluation in Passes

**Idea:** Traverse the syntax tree several times; each time, evaluate those equations \( a[i] = f(b[i_0], \ldots, z[i_z]) \) whose attributes \( b[i_0], \ldots, z[i_z] \) are already evaluated.

For a **strongly acyclic attribute system**:
- the local dependencies in \( G(s) \) at \( s \) define a sequence in which children can be visited so that at least one attribute can be evaluated after the visit of \( s \).
- in each pass through the tree at least one more attribute is evaluated.
- it requires at most \( n \) passes for evaluating \( n \) attributes.
- since a traversal strategy exists for evaluating one attribute, it might be possible to find a strategy to evaluate more attributes \( \leadsto \) the optimisation problem?!
- the ability to group attributes depends on the design of the equation system.

... in the example:
- **empty** and **first** can be computed together.
- **next** must be computed in a separate pass.

Implementing Local Evaluation

Consider example: numbering the leaves of a syntax tree.
Implementing Local Evaluation
Consider example: numbering the leafs of a syntax tree

Implementing Numbering of Leafs
Idea:
- use helper attributes \texttt{pre} and \texttt{post}
- in \texttt{pre} we pass the value of the last leaf down (inherited attribute)
- in \texttt{post} we pass the value of the last leaf up (synthetic attribute)

\[
\begin{align*}
\text{root:} & \quad \texttt{pre}[0] := 0 \\
& \quad \texttt{pre}[1] := \texttt{pre}[0] \\
& \quad \texttt{post}[0] := \texttt{post}[1] \\
\text{node:} & \quad \texttt{pre}[1] := \texttt{pre}[0] \\
& \quad \texttt{pre}[2] := \texttt{post}[1] \\
& \quad \texttt{post}[0] := \texttt{post}[2] \\
\text{leaf:} & \quad \texttt{post}[0] := \texttt{pre}[0] + 1
\end{align*}
\]

The Local Attribute Dependencies
- the attribute system is apparently strongly acyclic
- each node computes
  - the inherited attributes before descending into a child node (corresponding to a pre-order traversal)
  - the synthetic attributes after returning from a child node (corresponding to post-order traversal)

The Local Attribute Dependencies
- the attribute system is apparently strongly acyclic
- each node computes
  - the inherited attributes before descending into a child node (corresponding to a pre-order traversal)
  - the synthetic attributes after returning from a child node (corresponding to post-order traversal)
- if all attributes can be computed in a single depth-first traversal that proceeds from left- to right (with pre- and post-order evaluation)
- then we call this attribute system \texttt{l-attributed.}
**L-attributed**

**Definition**
An attribute system is L-attributed, if for all productions \( s := s_1 \ldots s_n \) every inherited attribute of \( s_j \) where \( 1 \leq j \leq n \) only depends on
- the attributes of \( s_1, s_2, \ldots, s_{j-1} \) and
- the inherited attributes of \( s \).

**Origin:**
- the attributes of an L-attributed grammar can be evaluated during parsing
- important if no syntax tree is required or if error messages should be emitted while parsing
- example: pocket calculator

**L-attributed**

**Definition**
An attribute system is L-attributed, if for all productions \( s := s_1 \ldots s_n \) every inherited attribute of \( s_j \) where \( 1 \leq j \leq n \) only depends on
- the attributes of \( s_1, s_2, \ldots, s_{j-1} \) and
- the inherited attributes of \( s \).

**Practical Applications**
- symbol tables, type checking/inference, and simple code generation can all be specified using L-attributed grammars

L-attributed grammars have a fixed evaluation strategy: a single depth-first traversal
- in general: partition all attributes into \( A = A_1 \cup \ldots \cup A_n \) such that for all attributes in \( A_i \), the attribute system is L-attributed
- perform a depth-first traversal for each attribute set \( A_i \)
Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using $L$-attributed grammars
- most applications annotate syntax trees with additional information
- the nodes in a syntax tree often have different types that depends on the non-terminal that the node represents

Implementation of Attribute Systems

In object-oriented languages, use a visitor pattern:

- class with a method for every non-terminal in the grammar
  ```java
  public abstract class Regex {
      public abstract void accept(Visitor v);
  }
  ```
- by overwriting one of the following methods, we implement an attribute-specific evaluation
  ```java
  public interface Visitor {
      public void visit(Dot re) { re.children(this); }
      public void visit(Ear re) { re.children(this); }
      public void visit(OrEx re) {
        public void visit(Token tok) {} // R \rightarrow R | R
      }
  }
  ```
- we pre-define a depth-first traversal of the syntax tree
  ```java
  public class OrEx extends Regex {
      public void accept(Visitor v) { v.visit(this); }
      public void children(Visitor v) {
        l.accept(v); r.accept(v);
      }
  }
  ```
Chapter 2: Symbol Tables

Symbol Tables

Consider the following Java code:

```java
void foo() {
    int A;
    void bar() {
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    bar();
    write(A);
}
```

- within the body of `bar` the definition of `A` is shadowed by the local definition
- each declaration of a variable `v` requires the compiler to set aside some memory for `v`; in order to perform an access to `v`, we need to know to which declaration the access is bound
- we consider only static binding, where the definition of a name `v` is in scope at all program points within the block
- however, the binding is not visible within local declarations of `v` are in scope

Scope of Identifiers

```java
void foo() {
    int A;
    void bar() {
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    bar();
    write(A);
}
```

Scope of Identifiers

```java
void foo() {
    int A;
    void bar() {
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    bar();
    write(A);
}
```
Scope of Identifiers

```java
void foo() {
    int A;
    void bar() {
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    bar();
    write(A);
}
```

administration of identifiers can be quite complicated...

Visibility Rules in Object-Oriented Languages

```java
1 public class Foo {
2    int x = 17;
3    protected int y = 5;
4    private int z = 42;
5    public int b() { return x; }
6 }
7 class Bar extends Foo {
8    protected double y = 0.5;
9    public int b(int a)
10    { return a+x; }
11 }
```

Observations:

- public
- protected
- no modifier
- private

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>no modifier</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
</tbody>
</table>

Dynamic Resolution of Functions

```java
public class Foo {
    protected int foo() { return 1; }
}
```

```java
class Bar extends Foo {
    protected int foo() { return 2; }
    public int test(boolean b) {
        Foo x = (b) ? new Foo() : new Bar();
        return x.foo();
    }
```

Observations:

- private member z is only visible in methods of class Foo
- protected member y is visible in the same package and in sub-class Bar, but here it is *shadowed* by double y
- Bar does not compile if it is not in the same package as Foo
- methods with the same name are different if their arguments differ — *static overloading*
Dynamic Resolution of Functions

```java
public class Foo {
    protected int foo() { return 1; }
}
class Bar extends Foo {
    protected int foo() { return 2; }
    public int test(boolean b) {
        Foo x = (b) ? new Foo() : new Bar();
        return x.foo();
    }
}
```

Observations:
- the type of `x` is `Foo` or `Bar`, depending on the value of `b`
- `x.foo()` either calls `foo` in line 2 or in line 5

Resolving Identifiers

Observation: each identifier in the AST must be translated into a memory access

Problem: for each identifier, find out what memory needs to be accessed by providing rapid access to its declaration

Idea:
1. **rapid access**: replace every identifier by a unique "name", namely an integer
   - integers as keys: comparisons of integers is faster
   - replacing various identifiers with number saves memory
2. link each usage of a variable to the declaration of that variable
   - track data structures to distinguish declared variables and visible variables
   - for languages without explicit declarations, create declarations when a variable is first encountered
(1) Replace each Occurrence with a Number

Rather than handling strings, we replace each string with a unique number.

Idea for Algorithm:

Input: a sequence of strings
Output: 1 sequence of numbers
2 table that allows to retrieve the string that corresponds to a number

Apply this algorithm on each identifier in the scanner.

Example for Applying this Algorithm

Input:

Output:

Example for Applying this Algorithm

Input:

Output:

Example for Applying this Algorithm

Input:

Output:

and
Implementing the Algorithm: Specification

Idea:
- implement a partial map $S : \text{String} \rightarrow \text{int}$
- use a counter variable $\text{count} = 0$ to track the number of different identifiers found so far

We thus define a function $\text{int get\_index(String w)}$:

```java
int get\_index(String w) { 
    if ($S(w) \equiv \text{undefined}$) {
        $S = S + \{w \rightarrow \text{count}\};$
        return $\text{count}++$
    } else return $S(w)$;
}
```

Data Structures for Partial Maps

possible data structures:
- list of pairs $(w, i) \in \text{String} \times \text{int}$:
  - $O(1)$
  - $O(n)$ $\sim$ too expensive $\times$
- balanced trees:
  - $O(\log(n))$
  - $O(\log(n))$ $\sim$ too expensive $\times$
- hash tables:
  - $O(1)$
  - $O(1)$ on average $\checkmark$

Data Structures for Partial Maps

possible data structures:
- list of pairs $(w, i) \in \text{String} \times \text{int}$:
  - $O(1)$
  - $O(n)$ $\sim$ too expensive $\times$
- balanced trees:
  - $O(\log(n))$
  - $O(\log(n))$ $\sim$ too expensive $\times$
- hash tables:
  - $O(1)$
  - $O(1)$ on average $\checkmark$
Data Structures for Partial Maps

possible data structures:
- list of pairs \((w, i) \in \text{String} \times \text{int}\):
  - \(O(1)\)
  - \(O(n)\)
  \(\sim\) too expensive \(\times\)
- balanced trees:
  - \(O(\log(n))\)
  - \(O(\log(n))\)
  \(\sim\) too expensive \(\times\)
- hash tables:
  - \(O(1)\)
  - \(O(1)\)
  on average \(\checkmark\)

Caveat: we will see that the handling of scoping requires additional operations that are hard to implement with hash tables.

An Implementation using Hash Tables

- allocated an array \(M\) of sufficient size \(m\)
- choose a hash function \(H : \text{String} \rightarrow [0, m - 1]\) with the following properties:
  - \(H(w)\) is cheap to compute
  - \(H\) distributes the occurring words equally over \([0, m - 1]\)
- Possible choices \((\tilde{x} = (x_0, \ldots, x_{r-1})):\)
  \(H_0(\tilde{x}) = (x_0 + x_{r-1}) \mod m\)
  \(H_1(\tilde{x}) = (\sum_{i=0}^{r-1} x_i \cdot p^i) \mod m\)
  \(H_2(\tilde{x}) = (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \ldots))) \mod m\)
  für eine Primzahl \(p\) (z. B. 31)
- We store the pair \((w, i)\) in a linked list located at \(M[H(w)]\)

Computing a Hash Table for the Example

With \(m = 7\) and \(H_0\) we obtain:

\[
\begin{array}{c|c|c}
0 & \text{If} & 8 \text{ the 10} \\
1 & & \\
2 & \text{pickled} & 6 \text{ peck 4 pickled 2} \\
3 & \text{of} & 5 \text{ where 9 peppers 7} \\
4 & & \\
5 & \text{pick} & 1 \text{ pete 0 a 3} \\
6 & & \\
\end{array}
\]

In order to find the index for the word \(w\), we need to compare \(w\) with all words \(x\) for which \(H(w) = H(x)\).

Data Structures for Partial Maps

possible data structures:
- list of pairs \((w, i) \in \text{String} \times \text{int}\):
  - \(O(1)\)
  - \(O(n)\)
  \(\sim\) too expensive \(\times\)
- balanced trees:
  - \(O(\log(n))\)
  - \(O(\log(n))\)
  \(\sim\) too expensive \(\times\)
- hash tables:
  - \(O(1)\)
  - \(O(1)\)
  on average \(\checkmark\)

Caveat: we will see that the handling of scoping requires additional operations that are hard to implement with hash tables.
Resolving Identifiers: (2) Symbol Tables

Check for the correct usage of variables:
- Traverse the syntax tree in a suitable sequence, such that
  - each definition is visited before its use
  - the currently visible definition is the last one visited
- for each identifier, we manage a stack of scopes
- if we visit a declaration of an identifier, we push it onto the stack
- upon leaving the scope, we remove it from the stack
- if we visit a usage of an identifier, we pick the top-most declaration from its stack
- if the stack of the identifier is empty, we have found an error

Example: A Table of Stacks

```
{ int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c; // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

Example: A Table of Stacks

```
{ int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c; // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

Resolving: Rewriting the Syntax Tree

- d declaration node
- b basic block
- a assignment

```
{ int a, b; // V, W
  b = 5;
  if (b>3) {
    int a, c; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c; // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```
Resolving: Rewriting the Syntax Tree

d declaration node
b basic block
a assignment

```
{ int a, b; // V, W
  b = 5;
  if (b>3) {
    int c, d; // X, Y
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c; // Z
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```