Semantic Analysis

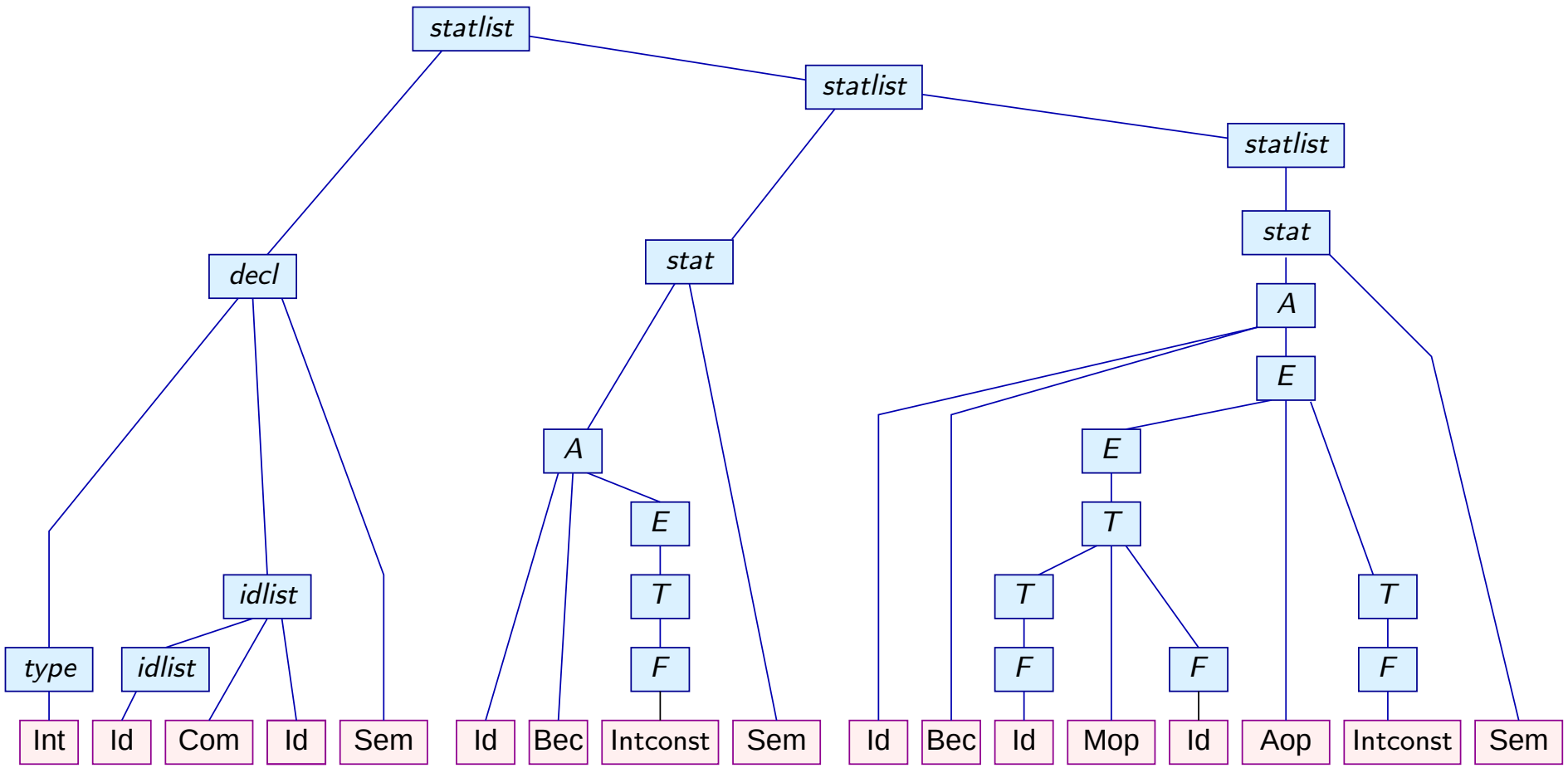
Sebastian Hack
Saarland University

W2015

Saarland University, Computer Science

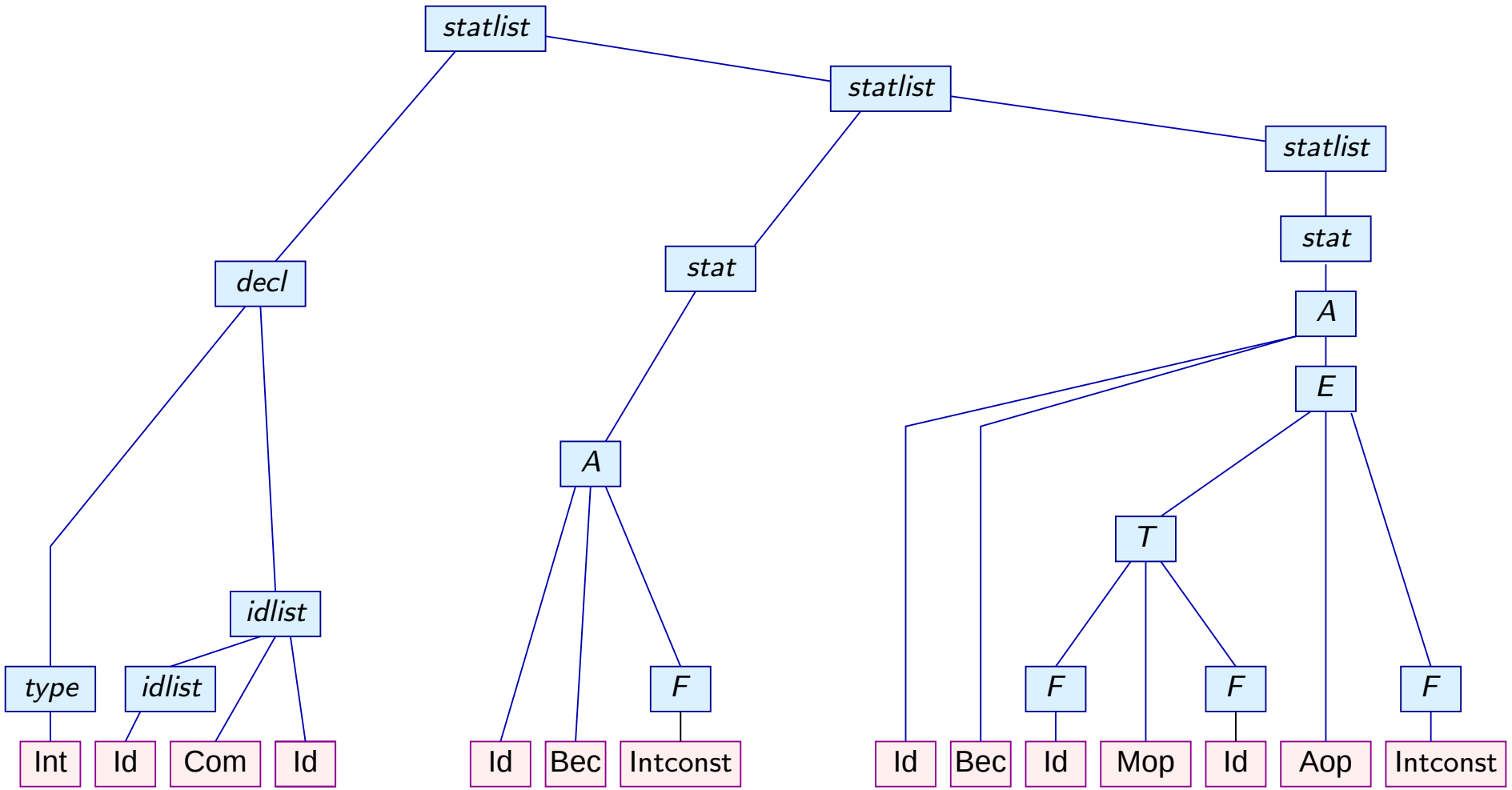
Abstract Syntax Trees (AST)

- Delete Terminals that were only there to make the grammar unambiguous
- Remove Nonterminals in chain productions
  (typical if precedence and associativity were expressed in grammar)



Abstract Syntax Trees (AST)

- Delete Terminals that were only there to make the grammar unambiguous
- Remove Nonterminals in chain productions
  (typical if precedence and associativity were expressed in grammar)



Semantic Analysis

Many programming languages have a static semantics. Interpretation of the syntax without knowing the inputs of the program.

Purpose:

- Refute (more) meaningless programs
- Infer information about program helpful for implementation

Program analyses with respect to the static semantics are often called semantic analyses. Commonly, we have two of them:

- Name Analysis
- Type Checking

Do we need semantic analysis?

Consider the following Python program:

```
def f(x):
    if x = 0:
        return x
    else:
        return y
```

What happens on `f(0)` and `f(1)`?

Do we need semantic analysis?

Python does not have a static semantics, hence it has to "rule out" many invalid programs at runtime:

- When an identifier is declared, enter it into a definition table
- When it is used, look up, if there is a value for that identifier
- Incurs significant runtime overhead
- Allows more flexible (really really really?) code:
  i.e. can add fields to objects at runtime

Similar things happen because of "duck typing":

- Resolve operator overloading at runtime: `a + b`
  Do we add two ints? Floats? Append to a string? Or is a an object of a class that redefines +?
- Similarly for methods: `a.b(c)`
  Does the class of `a` really provide a method `b` or a superclass?

To avoid runtime overhead, we use static semantics to make these questions decidable statically.

Limits of static semantics

- C leaves semantics of many programs (under certain inputs) undefined. Result: runtime errors, e.g. division by zero, dereference of non-allocated memory
- Static semantics cannot capture them: in general undecidable
- Make language more restrictive to avoid some of them. E.g. Java:
  - Cannot take address of a field or array cell
  - Garbage collection policy avoids dangling pointers
- "Totalize" behavior, e.g.
  - Throw exception on `null` dereference
  - Array out-of-bounds check
  - Incurs runtime overhead!

Name Analysis

- Every entity (variable, function, data type, etc.) has a defining occurrence (definition) that gives it a name and describes its properties with respect to static semantics (e.g. type)
- Every defining occurrence has a scope in which it is valid
- Scope is the static pendant to lifetime of referred object (at runtime)
- An applied occurrence refers to a definition

Purpose of name analysis:

- Ensure that every applied occurrence refers to a unique defining occurrence
- Relate applied occurrence to defining occurrence in AST

Issues: Validity, Visibility, Qualification, Namespaces, Overloading

Validity

The scope (of validity) of a definition of an identifier x is the part of the program (nodes in the AST) in which an applied occurrence of x refers to it.

Inside a scope, each identifier can only defined once.

Example

```
{
    int y;
    // ...
    int x = 1;
    // ...
    y = x + 1;
}
```

The definition of `x` is valid inside the entire block. But only visible after its definition.

Visibility

Inside its scope, a definition of an identifier x can be invisible (hidden, overwritten). Then, it is still forbidden to redefine x in the same scope (but in a nested one!) but not possible to refer to that definition:

```
for (int i = 0; i < n; i++)
    for (int i = 0; i < m; i++)
        A[i] = ...; // app occ of i refers
                    // to innermost definition
```

Possible in C. Java forbids overwriting definitions of local variables but allows

```
class X {
    int x;
    void foo(int x) {
        x = 1;
    }
}
```

Qualification

Some entities are composed of other entities (e.g. modules, classes, structs). Their definition opens a scope in which their constituents are defined:

```
struct vec3_t {
    float x, y, z;
};
```

The scope of `x`, `y`, and `z` is limited by `{ }`.

Use qualification to refer to constituents:

```
vec3_t v;
// ...
v.x = 1.0f;
```

In the static semantics, the `.` makes the identifier on the right-hand side refer to its definition inside the scope of the entity described by the left-hand side.

Multiple Name Spaces

Many programming languages allow multiple name spaces. More than one definition of the same identifier can be valid and visible at the same program point. Must be clear from the context which entity it is referred to.

Example (C)

labels, structs/unions, and functions/variables are in disjoint name spaces.

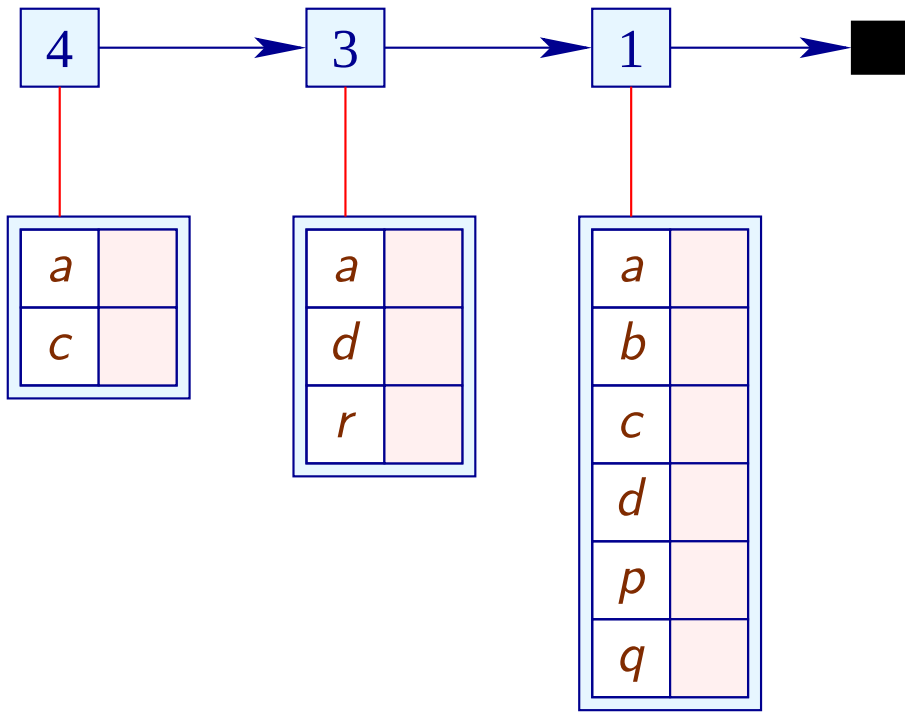
```
struct x {
    int x;
};

int foo(int x) {
    struct x y; // x refers to struct
    y.x = x;    // refers to parameter
    goto x;     // refers to label
x:
    return y.x; // refers to parameter
}
```

Symbol (Definition) Tables

To detect the valid defining occurrence for an applied occurence and to handle visibility one typically uses a stack of environments (Env = Id $\rightarrow$ Type).

```
1
int a; int b;
void p(); void q();
void p()
  2
  {
    int a; int c;
  }
int c; int d;
void q()
  3
  {
    int a; int d;
    void r()
      4
      {
        int a; int c;
              *
      }
  }
```



Pink boxes are pointers to the AST node of the identifier's declaration.

Symbol Table API

```
public class Scope {

  private final Map<String, Declaration> decls;
  private final Scope parent;

    private Scope(Scope parent) {
        this.parent = parent;
    }

    private Scope() {
        this(null);
    }

    public boolean isRoot()   { return parent = null; }
    public Scope enterBlock() { return new Scope(this); }
    public Scope leaveBlock() { return parent; }
```

Symbol Table API

```
  public static class NotFound extends Exception { }
  public static class AlreadyDefined extends Exception {

  public Declaration lookup(String name) throws NotFound {
    if (decls.containsKey(name))
      return decls.get(name);
    else if (!isRoot())
      return parent.lookup(name);
    else
      throw new NotFound();
  }

  public void add(Declaration decl) throws AlreadyDefined {
    String name = decl.getToken().getText();
    if (decls.containsKey(name))
      throw new AlreadyDefined();
    decls.put(name, decl);
  }
}
```

Type Checking C

Type checking C is done by a bottom-up traversal of the AST.

Resolve overloading based on types of operands and implicit type cast rules:

```
public Binary extends Expression {
    protected abstract Type applyOperatorTypeRules(Type left, T

    public Type getType() {
        if (this.type = null) this.type = computeType();
        return this.type;
    }

    protected Type computeType() {
        Type lt = left.getType();
        Type rt = right.getType();
        if (lt.isError() ∨ rt.isError())
            return Type.ERROR;
        return applyOperatorTypeRules(lt, rt);
    }
}
```

Type Inference

Often types of variables are uniquely identified by the way the variables are used. Use this information to avoid declarations and infer types automatically.

This is common in functional language and modern functional/imperative languages such as Scala, Rust, OCaml, ...

Consider following example:

```
let rec fac = fun → if x ≤ 0 then 1
                    else x * fac (x - 1)
```

Because one branch uses the int constant 1, the term `x * fac (x - 1)` must be an int as well. Hence `x` and `fac (x - 1)` must be ints as well. And finally `fac` is a function from `int → int`.

A Toy Functional Language: Syntax

Syntax:

$$
\begin{array}{rcl}
\mathit{Expr} \ni e & ::= & c \mid x \mid \oplus \overline{e} \\
& \mid & \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \\
& \mid & (e_1, \ldots, e_k) \mid [] \mid e_1 :: e_2 \\
& \mid & (e_1\ e_2) \mid (\texttt{fun } x \to e) \\
& \mid & \texttt{let } x_1 = e_1 \texttt{ in } e_0 \\
& \mid & \texttt{let rec } x_1 = e_1 \texttt{ and } \ldots \texttt{ and } x_n = e_n \texttt{ in } e_0
\end{array}
$$

Types:

$$
\mathit{Type} \ni t ::= \texttt{int} \mid \texttt{bool} \mid (t_1 * \cdots * t_n) \mid t\ \texttt{list} \mid t_1 \to t_n
$$

A Toy Functional Language: Static Semantics

Define a relation

$$\Gamma \vdash e : t \subseteq (Id \to Type) \times Expr \times Type$$

to indicate that under the environment Γ, expression e has type t.

Define typing rules inductively over the syntax.

Axioms:

$\Gamma \vdash c : t_c$	$(Const)$	t_c type of constant c
$\Gamma \vdash [] : t$ list	(Nil)	$\forall t$
$\Gamma \vdash x : (\Gamma\, x)$	(Var)	look up type of x in environment

A Toy Functional Language: Static Semantics

Op: $$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}}$$

Comp: $$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \textbf{bool}}$$

If: $$\frac{\Gamma \vdash e_0 : \textbf{bool} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash (\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) : t}$$

Tupel: $$\frac{\Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash (e_1, \ldots, e_m) : (t_1 * \ldots * t_m)}$$

Cons: $$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \text{ list}}{\Gamma \vdash (e_1 :: e_2) : t \text{ list}}$$

App: $$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}$$

Fun: $$\frac{\Gamma \oplus \{x \mapsto t_1\} \vdash e : t_2}{\Gamma \vdash \textbf{fun } x \to e : t_1 \to t_2}$$

Let: $$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \oplus \{x_1 \mapsto t_1\} \vdash e_0 : t}{\Gamma \vdash (\textbf{let } x_1 = e_1 \textbf{ in } e_0) : t}$$

Letrec: $$\frac{\Gamma' \vdash e_1 : t_1 \quad \ldots \quad \Gamma' \vdash e_m : t_m \quad \Gamma' \vdash e_0 : t}{\Gamma \vdash (\textbf{let rec } x_1 = e_1 \textbf{ and} \ldots \textbf{and } x_m = e_m \textbf{ in } e_0) : t}$$
$$\text{with} \quad \Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$$

An Example Derivation

$$\dfrac{\dfrac{\Gamma \vdash x : \textbf{int} \quad \Gamma \vdash 0 : \textbf{int}}{\Gamma \vdash x \leq 0 \ : \ \textbf{bool}} \cdots \quad \cdots \quad \dfrac{\Gamma \vdash x : \textbf{int} \quad \dfrac{\Gamma \vdash \mathsf{fac} : \textbf{int} \to \textbf{int} \quad \dfrac{\Gamma \vdash x : \textbf{int} \quad \Gamma \vdash 1 : \textbf{int}}{\Gamma \vdash x - 1 : \textbf{int}}}{\Gamma \vdash \mathsf{fac}\ (x-1) \ : \ \textbf{int}}}{\Gamma \vdash x \cdot \mathsf{fac}\ (x-1) \ : \ \textbf{int}} \quad \dfrac{\Gamma \vdash 1 \ : \ \textbf{int}}{}}{\Gamma \vdash \textbf{if}\ x \leq 0\ \textbf{then}\ 1\ \textbf{else}\ x \cdot \mathsf{fac}\ (x-1) \ : \ \textbf{int}}$$

Remarks

- An important property of a type system is preservation:
  The type of an expression does not change under evaluation.

  $$\text{If } \Gamma \vdash e_1 : \tau \text{ and } e_1 \Rightarrow^* e_2 \text{ then } \Gamma \vdash e_2 : \tau$$

  Our type system exhibits preservation.

- preservation + progress = type safety
  "Well-typed programs don't go wrong"

- Given an environment, the rules can be used to check if the environment leads to a correct typing

- Note that an expression can have multiple types. For every Γ and every t, we can derive

  $$\Gamma \vdash \texttt{fun x} \rightarrow \texttt{x} : t \rightarrow t$$

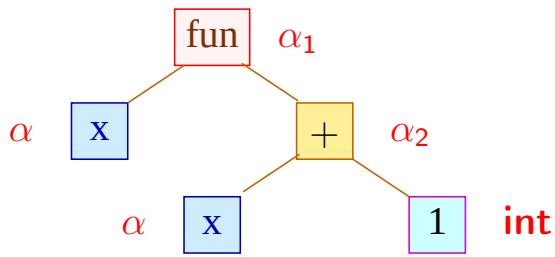
- How do we characterize all valid types for an expression?

An Equation System

- Build an equation system to describe all valid types for an expression
- Introduce type variables for the types of expressions with unknown type
- Technicality: To have one equation system for an expression, name variable apart to make their name unique
- Build equations based on typing rules

Consider `fun x ->x + 1`. Based on the typing rules, we come up with the following equations:

Fun	:	α_1	$=$	$\alpha \to \alpha_2$
Op	:	α_2	$=$	**int**
		α	$=$	**int**
		int	$=$	**int**



and hence

$\alpha = \textbf{int}$ $\alpha_1 = \textbf{int} \to \textbf{int}$ $\alpha_2 = \textbf{int}$

An Equation System

Let $\alpha[e]$ be the type variable for expression e.

Const:	$e \equiv b$	$\alpha[e] = t_b$
Nil:	$e \equiv [\,]$	$\alpha[e] = \alpha$ list $\quad (\alpha$ fresh$)$
Op:	$e \equiv e_1 + e_2$	$\alpha[e] = \textbf{int}$ $\alpha[e_1] = \textbf{int}$ $\alpha[e_2] = \textbf{int}$
Comp:	$e \equiv e_1 = e_2$	$\alpha[e_1] = \alpha[e_2]$ $\alpha[e] = \textbf{bool}$
Tupel:	$e \equiv (e_1, \ldots, e_m)$	$\alpha[e] = (\alpha[e_1] * \ldots * \alpha[e_m])$
Cons:	$e \equiv e_1 :: e_2$	$\alpha[e_2] = \alpha[e_1]$ list $\alpha[e] = \alpha[e_1]$ list
If:	$e \equiv \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$	$\alpha[e_0] = \textbf{bool}$ $\alpha[e] = \alpha[e_1]$ $\alpha[e] = \alpha[e_2]$
Fun:	$e \equiv \textbf{fun } x \rightarrow e_1$	$\alpha[e] = \alpha[x] \rightarrow \alpha[e_1]$
App:	$e \equiv e_1\ e_2$	$\alpha[e_1] = \alpha[e_2] \rightarrow \alpha[e]$
Letrec:	$e \equiv \textbf{let rec } x_1 = e_1 \textbf{ and} \ldots \textbf{ and } x_m = e_m \textbf{ in } e_0$	$\alpha[x_1] = \alpha[e_1] \quad \ldots$ $\alpha[x_m] = \alpha[e_m]$ $\alpha[e] = \alpha[e_0]$

A solution σ is a unifier, a substitution (= map from type variables to types) such that $\sigma s_i \equiv \sigma t_i$ for every equation $s_i = t_i$ in the equation system.

Correctness

To make use of the equation system, we have to show the following theorem:

Theorem

Let e be an expression, V be the set of variables in e and E be the equation system for e. Then,

1. *If σ is a solution of E, then there is a derivation of $\Gamma \vdash e : t$ with*

$$\Gamma = \{x \mapsto \sigma(\alpha[x]) \mid x \in V\} \quad \textit{and} \quad t = \sigma(\alpha[e])$$

2. *Consider a derivation of the judgement $\Gamma \vdash e : t$ that contains judgements $\Gamma \vdash e' : t_{e'}$ for all subterms e' of e. Then, the substitution*

$$\sigma(\alpha[e']) = \begin{cases} t_{e'} & e' \text{ subterm of } e \\ \Gamma\, x & e' \equiv x \in V \end{cases}$$

is a solution of E.

Substitutions: Examples

1. Consider the equation

$$Y = X \to X$$

   The set of solutions is given by the substitution

$$\{X \mapsto t, Y \mapsto (t \to t)\}$$

   for every type t.

2. The equation

$$X \to \mathsf{int} = \mathsf{bool} \to Z$$

   has exactly one solution:

$$\{X \mapsto \mathsf{bool}, Z \mapsto \mathsf{int}\}$$

3. The equation

$$\mathsf{bool} = X \to Y$$

   has no solution.

Substitutions: Definitions

- σ is idempotent if $\sigma \circ \sigma = \sigma$
  Means that there is no variable Y with $\sigma(Y) = X$ with $\sigma(X) \neq X$.

- An idempotent σ is most general if for every other idempotent unifier τ, there is an appropriate substitution τ' with $\tau = \tau' \circ \sigma$. E.g.

$$\{Y \mapsto (X \to X), X \mapsto X\}$$

  is the most general unifier of the first example on the last slide.

- A set of term equations $s_i = t_i$, $1 \leq i \leq m$ has either no solution or a most general idempotent unifier.

Computing the most general unifier

Given the equation system $s_i = t_i$, $1 \leq i \leq m$, start with
unifyList $[(s_1, t_1), \ldots, (s_m, t_m)]$ $\emptyset$

$$
\begin{array}{l}
\textbf{let rec}\ \text{unify}\ (s,t)\ \theta\ =\ \textbf{if}\ \theta\, s \equiv \theta\, t\ \textbf{then}\ \theta \\
\qquad\qquad \textbf{else}\ \textbf{match}\ (\theta\, s, \theta\, t) \\
\qquad\qquad\quad \textbf{with}\ \ (X,t)\ \rightarrow\ \textbf{if}\ \text{occurs}\,(X,t)\ \textbf{then}\ \text{Fail} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \{X \mapsto t\} \circ \theta \\
\qquad\qquad\qquad\ \mid\ \ (t,X)\ \rightarrow\ \textbf{if}\ \text{occurs}\,(X,t)\ \textbf{then}\ \text{Fail} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \{X \mapsto t\} \circ \theta \\
\qquad\qquad\qquad\ \mid\ \ (f(s_1,\ldots,s_k), f(t_1,\ldots,t_k))\ \rightarrow \\
\qquad\qquad\qquad\qquad\quad \text{unifyList}\ [(s_1,t_1),\ldots,(s_k,t_k)]\ \theta \\
\qquad\qquad\qquad\ \mid\ \ (a,a)\ \rightarrow\ \theta \\
\qquad\qquad\qquad\ \mid\ \ \_\ \rightarrow\ \text{Fail} \\
\textbf{and}\ \ \text{unifyList list}\ \theta\ =\ \textbf{match}\ \text{list} \\
\qquad\qquad \textbf{with}\ \ [\,]\ \rightarrow\ \theta \\
\qquad\qquad\ \mid\ \ ((s,t) :: \text{rest})\ \rightarrow\ \textbf{let}\ \theta = \text{unify}\ (s,t)\ \theta \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \textbf{if}\ \theta = \text{Fail}\ \textbf{then}\ \text{Fail} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{else}\ \text{unifyList rest}\ \theta
\end{array}
$$

Syntax-Directed Version (Algorithm W)

Solving the equation system as a whole does not give precise error messages (we don't know where an error was). Instead of collecting equations, solve equations in a syntax-directed way:

$$
\begin{array}{lllll}
\textbf{let rec}\ \mathcal{W}\ e\ (\Gamma,\theta) & = & \textbf{match}\ e & & \\
\quad \textbf{with} & c & \to & (t_c,\theta) & \\
\quad\ \mid & [\,] & \to & \textbf{let}\alpha = \mathsf{new}() & \\
 & & & \textbf{in}\ (\alpha\ \mathsf{list},\theta) & \\
\quad\ \mid & x & \to & (\Gamma(x),\theta) & \\
\quad\ \mid & (e_1 :: e_2) & \to & \quad \textbf{let} \quad (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) & \\
 & & & \textbf{in let} \quad (t_2,\theta) = \mathcal{W}\ e_2\ (\Gamma,\theta) & \\
 & & & \textbf{in let} \quad \theta = \mathsf{unify}\ (t_1\ \mathsf{list}, t_2)\ \theta & \\
 & & & \textbf{in}\ \ (t_2,\theta) & \\
\quad\ \mid & (e_1\,'+'\,e_2) & \to & \quad \textbf{let} \quad (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) & \\
 & & & \textbf{in let} \quad (t_2,\theta) = \mathcal{W}\ e_2\ (\Gamma,\theta) & \\
 & & & \textbf{in let} \quad \theta = \mathsf{unify}\ (\textbf{int}, t_1)\ \theta & \\
 & & & \textbf{in let} \quad \theta = \mathsf{unify}\ (\textbf{int}, t_2)\ \theta & \\
 & & & \textbf{in}\ \ (\textbf{int},\theta) & \\
\quad\ \mid & (e_1\ e_2) & \to & \quad \textbf{let} \quad (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) & \\
 & & & \textbf{in let} \quad (t_2,\theta) = \mathcal{W}\ e_2\ (\Gamma,\theta) & \\
 & & & \textbf{in let} \quad \alpha = \mathsf{new}\,() & \\
 & & & \textbf{in let} \quad \theta = \mathsf{unify}\ (t_1, t_2 \to \alpha)\ \theta & \\
 & & & \textbf{in}\ \ (\alpha,\theta) &
\end{array}
$$

Syntax-Directed Version (Algorithm W)

$$
\begin{array}{lll}
\mid & (\text{fun } x \rightarrow e) & \\
& \rightarrow & \textbf{let} \quad \alpha = \text{new}() \\
& & \textbf{in let} \quad (t, \theta) = \mathcal{W}\ e\ (\Gamma \oplus \{x \mapsto \alpha\}, \theta) \\
& & \textbf{in } (\alpha \rightarrow t, \theta) \\
\mid & (\textbf{let } x_1 = e_1 \textbf{ in } e_0) & \\
& \rightarrow & \textbf{let} \quad (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
& & \textbf{in let} \quad \Gamma = \Gamma \oplus \{x_1 \mapsto t_1\} \\
& & \textbf{in let} \quad (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
& & \textbf{in } \ (t_0, \theta) \\
\mid & (\text{let rec } x_1 = e_1 \text{ and} \ldots \text{ and } x_m = e_m \text{ in } e_0) & \\
& \rightarrow & \textbf{let} \quad \alpha_1 = \text{new}() \\
& & \quad \ldots \\
& & \textbf{in let} \quad \alpha_m = \text{new}() \\
& & \textbf{in let} \quad \Gamma = \Gamma \oplus \{x_1 \mapsto \alpha_1, \ldots, x_m \mapsto \alpha_m\} \\
& & \textbf{in let} \quad (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
& & \textbf{in let} \quad \theta = \text{unify } (\alpha_1, t_1)\ \theta \\
& & \quad \ldots \\
& & \textbf{in let} \quad (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) \\
& & \textbf{in let} \quad \theta = \text{unify } (\alpha_m, t_m)\ \theta \\
& & \textbf{in let} \quad (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
& & \textbf{in } \ (t_0, \theta)
\end{array}
$$

(Let-) Polymorphism

Consider the expression

```
let single = fun y → [y]
in  single (single 1)
```

We derive $\gamma \to \gamma\,\texttt{list}$ as the type for `single`.
Because of `(single 1)` we unify γ with `int`.
The outer expression then wants to unify `int` with `int list` which fails.

Reason:
Type variables have to be instantiated once for the expression.

Solution:
Introduce type schemas and allow type variables to be instantiated differently in subexpressions.

$$\forall \alpha_1, \ldots, \alpha_n.t$$

generalizes type variables $\alpha_1, \ldots, \alpha_n$ in type expression t.

(Let-) Polymorphism

Introduce new typing rules for let and let rec that generalize type variables that do not occur in environment:

$$\text{Inst:}\quad \frac{\Gamma(x) = \forall\, \alpha_1, \ldots, \alpha_k.t}{\Gamma \;\vdash\; x \;:\; t[t_1/\alpha_1, \ldots, t_k/\alpha_k]} \qquad \text{(for all } t_1, \ldots, t_k\text{)}$$

$$\text{Let:}\quad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \oplus \{x \mapsto \text{close}\, t_1\, \Gamma\} \vdash e_0 : t_0}{\Gamma \;\vdash\; (\textbf{let}\; x_1 = e_1\; \textbf{in}\; e_0) \;:\; t_0}$$

The function close $t\ \Gamma$ generalizes all type variables in t that do not occur in Γ. We modify Algorithm W accordingly:

$$\begin{array}{l}
\textbf{fun}\ \text{inst}\ (\forall\, \alpha_1, \ldots, \alpha_k.\, t)\ = \\
\qquad\qquad \textbf{let}\ \ \beta_1 = \text{new}() \\
\qquad\qquad \ldots \\
\qquad\quad \textbf{in let}\ \ \beta_k = \text{new}() \\
\qquad\quad \textbf{in}\ \ t[\beta_1/\alpha_1, \ldots, \beta_k/\alpha_k]
\end{array}$$

$$\begin{array}{lll}
\textbf{let rec}\ \ \mathcal{W}\ e\ (\Gamma, \theta) & = \ldots & \\
\mid \quad x & \rightarrow & (\text{inst}\ (\theta(\Gamma(x))), \theta) \\
\mid \quad (\text{let}\ x_1 = e_1\ \text{in}\ e_0) & & \\
\qquad\qquad & \rightarrow \qquad \textbf{let} & (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
\qquad\qquad & \quad\ \textbf{in let} & s_1 = \text{close}\ (\theta\ t_1)\ (\theta \circ \Gamma) \\
\qquad\qquad & \quad\ \textbf{in let} & \Gamma = \Gamma \oplus \{x_1 \mapsto s_1\} \\
\qquad\qquad & \quad\ \textbf{in let} & (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
\qquad\qquad & \quad\ \textbf{in} & (t_0, \theta)
\end{array}$$