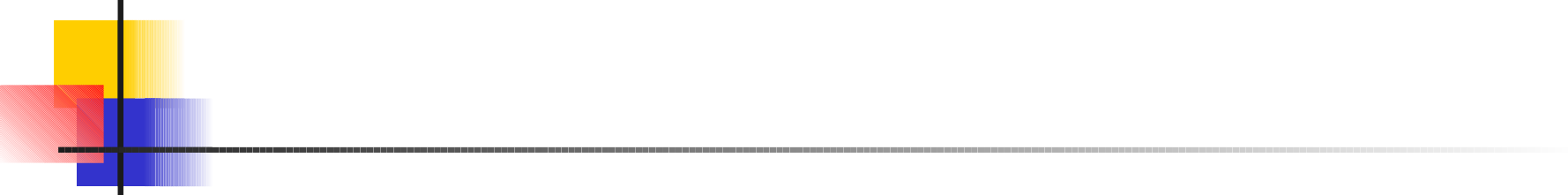


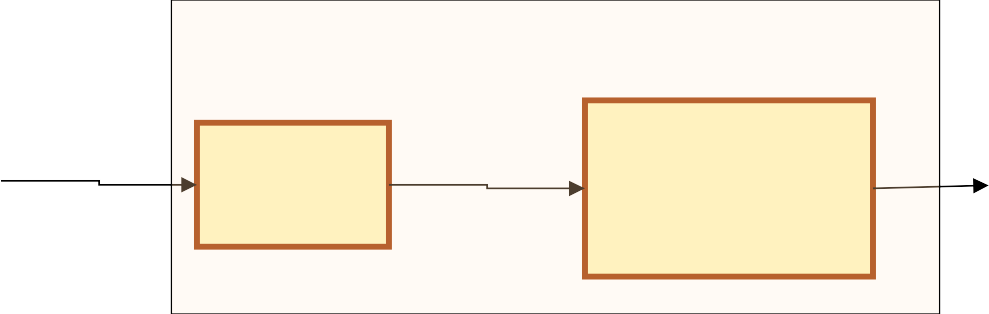
Compiler Design

Type Checking



Static Checking

Token Stream



Abstract Syntax Tree

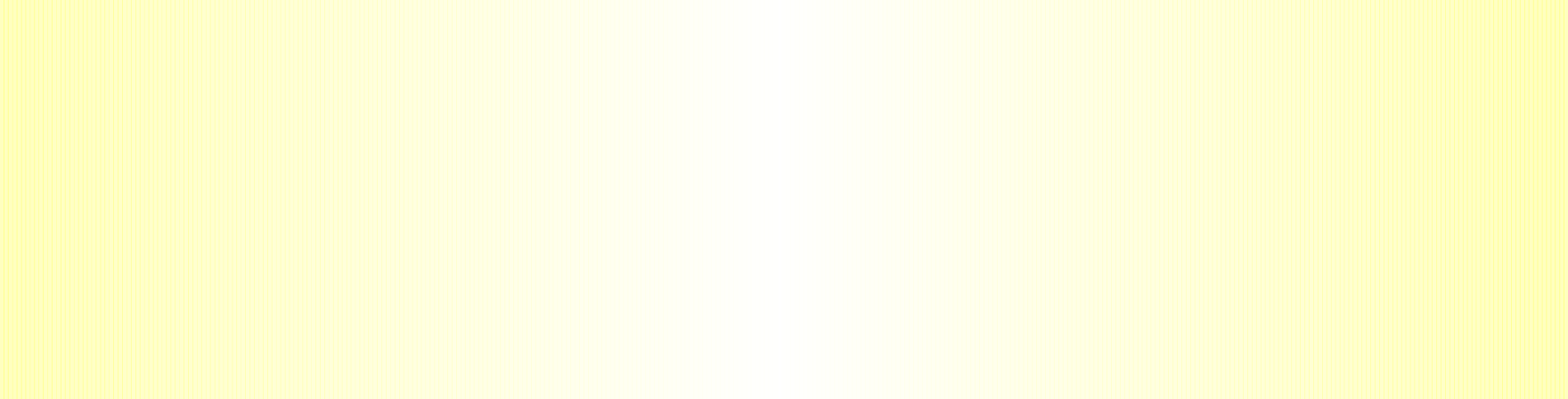
Parser

Intermediate Code Generator

Decorated Abstract Syntax Tree

Static Checker

Intermediate Code





Static (Semantic) Checks



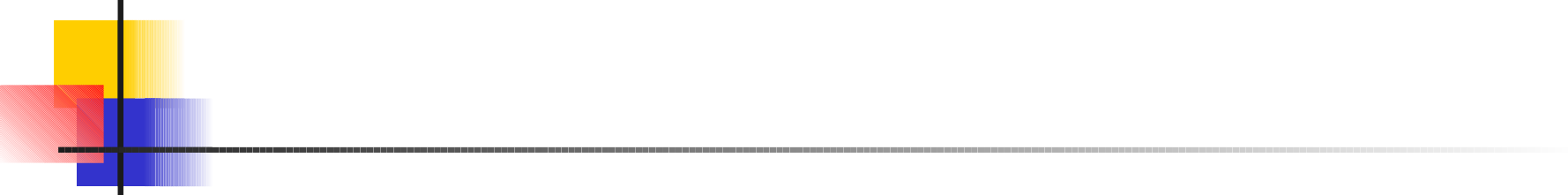






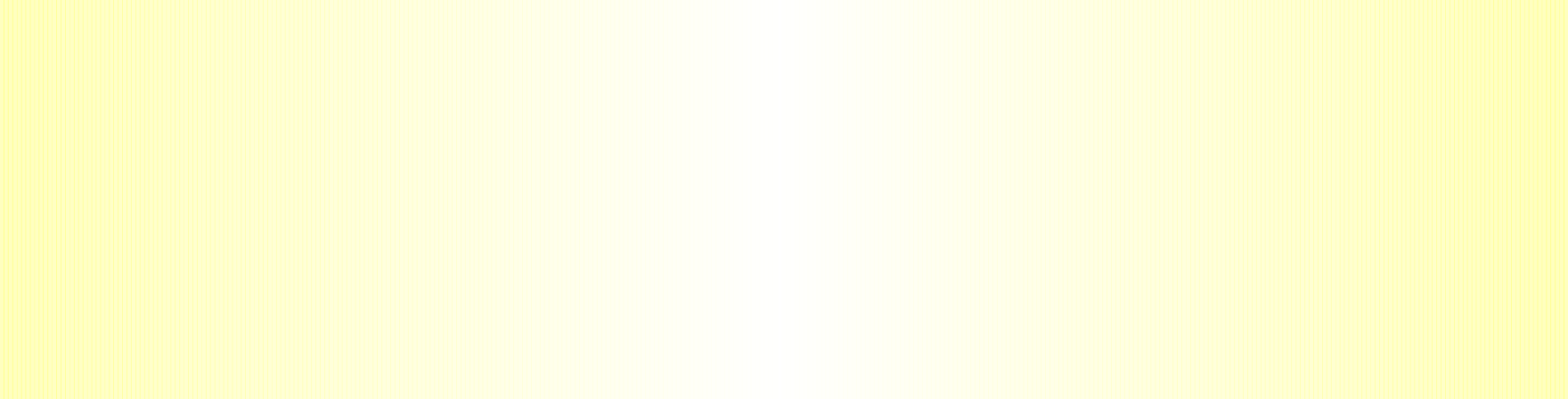
Type checks: operator applied to incompatible operands? Flow of control checks: break (outside while?) Uniqueness checks: labels in case statements

Name related checks: same name?



Type Checking

### 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

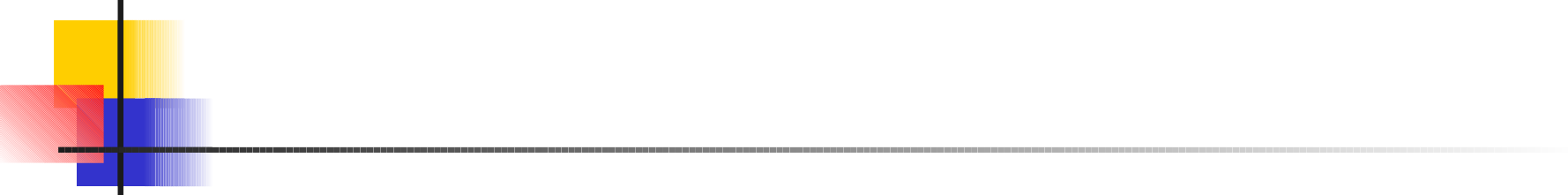
f(a1, a2, …, an) – function applied to correct arguments.

* Information gathered by a type checker:
  + Needed during code generation.

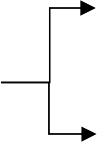


Type Systems

* A collection of rules for assigning type expressions to the various parts of a program.
* Based on: Syntactic constructs, notion of a type.
* Example: If both operators of “+”, “-”, “\*” are of type integer then so is the result.
* Type Checker: An implementation of a type system.
  + Syntax Directed.
* Sound Type System: eliminates the need for checking type errors during run time.



Type Expressions

* + Implicit Assumptions:
    - Each program has a type
    - Types have a structure

Expressions Statements

### Basic

Types

Type

Constructors

Boolean Real Enumerations Void

Variables

Character integer Sub-ranges Error Names

Arrays Records Sets Pointers Functions

(strings)

# Representation of Type Expressions

**→**

**x pointer**

**→**

**x pointer**

**cell**

**x**

**= record x**

**x**

**char char**

**integer**

**char**

**integer**

**info**

**int**

**next**

**ptr**

Tree DAG

struct cell {

int info;

struct cell \* next;

(char x char)→ pointer (integer) };

# Type Expressions Grammar



### Type → int | float | char | …

| void

| error

| name

| variable

| array( size, Type)

| record( (name, Type)\*)

| pointer( Type)

| tuple((Type)\*)

Basic Types

Structured Types

### | fcn(Type, Type) (Type

→ Type)

# A Simple Typed Language



### Program → Declaration; Statement

Declaration

→ Declaration; Declaration

| id: Type

Statement

Expression

→ Statement; Statement

| id := Expression

| if Expression then Statement

| while Expression do Statement

→ literal | num | id

| Expression mod Expression

| E[E] | E ↑ | E (E)



Type Checking Expressions

E → int\_const

E → float\_const

{ E.type = int }

{ E.type = float }

E → id

{ E.type = sym\_lookup(id.entry, type) }

E → E1 + E2 {E.type = if E1.type {int, float} |

E2.type  {int, float}) then error

else if E1.type == E2.type == int then int

else float }



Type Checking Expressions

E → E1 [E2] {E.type = if E1.type = array(S, T) ∧

E2.type = int then T else error}

E → \*E1 {E.type = if E1.type = pointer(T) then T

else error}

E → &E1 {E.type = pointer(E1.type)}

E → E1(E2) {E.type = if (E1.type = fcn(S, T) ∧

E2.type = S, then T else error}

E → (E1, E2) {E.type = tuple(E1.type, E2.type)}



Type Checking Statements

## S → id := E

{S.type := if id.type = E.type then void else error}

S → if E then S1 {S.type := if E.type = boolean

then S1.type else error}

S → while E do S1 {S.type := if E.type = boolean

then S1.type}

S → S1; S2 {S.type := if S1.type = void ∧

S2.type = void then void else error}



Equivalence of Type Expressions

### Problem: When in E1.type = E2.type?

* We need a precise definition for type equivalence
* Interaction between type equivalence and type representation

Example: type vector = array [1..10] of real type weight = array [1..10] of real var x, y: vector; z: weight

### Name Equivalence: When they have the same name.

* x, y have the same type; z has a different type.

Structural Equivalence: When they have the same structure.

* x, y, z have the same type.



Structural Equivalence

* Definition: by Induction
  + Same basic type (basis)
  + Same constructor applied to SE Type (induction step)
  + Same DAG Representation

### In Practice: modifications are needed

* + Do not include array bounds – when they are passed as parameters
  + Other applied representations (More compact)

### Can be applied to: Tree/ DAG

* + Does not check for cycles
  + Later improve it.



Algorithm Testing Structural Equivalence

**function** sequiv(s, t): **boolean**

{ if (s ∧ t are of the same basic type) **return** true; if (s = array(s1, s2) ∧ t = array(t1, t2))

**return** sequiv(s1, t1) ∧ sequiv(s2, t2);

if (s = tuple(s1, s2) ∧ t = tuple(t1, t2))

**return** sequiv(s1, t1) ∧ sequiv(s2, t2); if (s = fcn(s1, s2) ∧ t = fcn(t1, t2))

**return** sequiv(s1, t1) ∧ sequiv(s2, t2);

if (s = pointer(s1) ∧ t = pointer(t1))

**return** sequiv(s1, t1);

}



Recursive Types

Where: Linked Lists, Trees, etc.

How: records containing pointers to similar records Example: type link = ↑ cell;

cell = record info: int; next = link end

Representation:

**cell = record x**

**x x**

**info int next ptr**

**cell = record x**

**x x**

**info int next ptr**

DAG with Names

**cell**

Substituting names out (cycles)

### C Policy: avoid cycles in type graphs by:



Recursive Types in C

* + - Using structural equivalence for all types
    - Except for records → name equivalence
  + Example:
    - **struct cell {int info; struct cell \* next;}**

### Name use: name cell becomes part of the type of the record.

* + Use the acyclic representation
  + Names declared before use – except for pointers to records.
  + Cycles – potential due to pointers in records
  + Testing for structural equivalence stops when a record constructor is reached ~ same named record type?



Overloading Functions & Operators

### Overloaded Symbol: one that has different meanings depending on its context

* Example: Addition operator +
* Resolving (operator identification): overloading is resolved when a unique meaning is determined.
* Context: it is not always possible to resolve overloading by looking only the arguments of a function
  + Set of possible types
  + Context (inherited attribute) necessary



Overloading Example

function “\*” (i, j: integer) **return** complex; function “\*” (x, y: complex) **return** complex;

* Has the following types: fcn(tuple(integer, integer), integer) fcn(tuple(integer, integer), complex) fcn(tuple(complex, complex), complex)

int i, j; k = i \* j;



Narrowing Types

E’ → E

{E’.types = E. types

E.unique = if E’.types = {t} then t else error}

## E → id

{E.types = lookup(id.entry)}

E → E1(E2) {E.types = {s' |  s  E2.types and

t = E.unique

s→s’

 E1.types}

S = {s | s  E2.types and S→t E1.types}

E2.unique = if S ={s} then S else error E1.unique = if S = {s} then S→t else error



Polymorphic Functions

### Defn: a piece of code (functions, operators) that can be executed with arguments of different types.

* + Examples: Built in Operator indexing arrays, pointer manipulation
  + Why use them: facilitate manipulation of data structures regardless of types.
  + Example ML:

fun length(lptr) = if null (lptr) then 0

else length(+l(lptr)) + 1

P → D ; E



A Language for Polymorphic Functions

D → D ; D | id : Q

Q →  α. Q | T

T → fcn (T, T) | tuple (T, T)

| unary (T) | (T)

| basic

| α

E → E (E) | E, E | id



Type Variables

* Why: variables representing type expressions allow us to talk about unknown types.
  + Use Greek alphabets α, β, γ …

### Application: check consistent usage of identifiers in a language that does not require identifiers to be declared before usage.

* + A type variable represents the type of an undeclared identifier.

### Type Inference Problem: Determine the type of a language constant from the way it is used.

* + We have to deal with expressions containing variables.

# Examples of Type Inference



### Type link ↑ cell;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Procedure mlist | (lptr: | link; | procedure | p); |
| { while lptr <> | null |  |  |  |
| { p(lptr); | lptr := | lptr↑ | .next}} |  |

Hence: p: link → void

Function deref (p)

{ return p ↑; }

P: β, β = pointer(α)

Hence deref:  α. pointer(α) → α



Program in Polymorphic Language

deref:  α. pointer(α) → α

q: pointer (pointer (integer))

apply: α0

deref (deref( (q))

Notation:

→ fcn x tuple

deref0: pointer (α0 **)** → α0 apply: αi derefi: pointer

q: pointer (pointer (integer))

(αi **)** → αi

Subsripts i and o distinguish between the inner and outer occurrences of deref, respectively.



Type Checking Polymorphic Functions

* + Distinct occurrences of a p.f. in the same expression need not have arguments of the same type.
    - deref ( deref (q))
    - Replace α with fresh variable and remove  (αi, αo)
  + The notion of type equivalence changes in the presence of variables.
    - Use unification: check if s and t can be made structurally equivalent by replacing type vars by the type expression.
  + We need a mechanism for recording the effect of unifying two expressions.
    - A type variable may occur in several type expressions.



Substitutions and Unification

* Substitution S: a mapping from type variables to type expressions.

Function aplly (t: type Expr, S: Substitution): type Expr

{ if (t is a basic type) return t;

if (t is a variable) return S(t); -- check if t  S

if (t is t1 →

t2) return (apply (t1) →

apply (t2)); }

* Instance: S(t) is an instance of t written S(t) < t.
  + Examples: pointer (integer) < pointer (α) , int
* Unify: t1 ≈ t2 if  S. S (t1) = S (t2)
* Most General Unifier S: A substitution S:
  + S (t1) = S (t2)
  + S’. S’ (t1) = S’ (t2)  t. S’(t) < S(t).

→ real ≠

α→ α



Polymorphic Type checking Translation Scheme

E → E1 (E2) { p := mkleaf(newtypevar);

unify (E1.type, mknode(‘→’, E2.type, p); E.type = p}

E → E1, E2 {E.type := mknode(‘x’, E1.type, E2.type); }

E → id { E.type := fresh (id.type) }

fresh (t): replaces bound variables in t by fresh variables.

Returns pointer to a node representing result type.

fresh( α.pointer(α) → α) = pointer(α1) → α1.

unify (m, n): unifies expressions represented by m and n.

* Side-effect: keep track of substitution
* Fail-to-unify: abort type checking.

Given: derefo (derefi (q))



PType Checking Example

q = pointer (pointer (int))

Bottom Up: fresh (α. Pointer(α) → α)

→ **: 3**

**pointer : 2**

derefo

derefi q

→ **: 3**

**pointer : 2**

αo **: 1**

→ **: 3**

→ **: 6**

**pointer : 5**

αi **: 4**

**m**→ **: 6**

**pointer : 9**

**pointer : 8**

**integer : 7**

**n**→

α **: 1**

**: 6**

**pointer : 2**

αo **: 1**

**pointer : 5**

αi **: 4**

**pointer : 5**

**pointer : 8**

**integer : 7**

**β : 8**