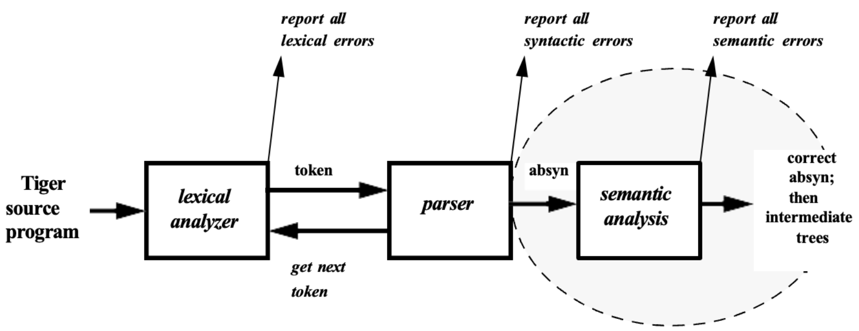
编译原理

5. 语义分析

rainoftime.github.io
浙江大学
计算机科学与技术学院

Content



report all
lexical errors
report all
syntactic errors
report all
semantic errors
Tiger
source
program
lexical
analyzer
token
get next
token
parser
absyn
semantic
analysis
correct
absyn;
then
intermediate
trees

The Limitations of CFG

- Consider the following grammar and programs
  - Does the corresponding parser accept the programs?

```
S ::= Decl  Stmt
Decl ::= Type  id | Decl; Decl
Type ::= string | int
Stmt ::= Stmt; Stmt |
         id = Exp | ...
Exp ::= Exp * Exp | id | num | char* | ...
```

string x; int z; x= "hello world"; z=x+1;	int x; int z; z=x+1;

- Many things can not be decided by syntax analysis
  - Does the dimension of a reference match the declaration?
  - Is an array access out of bound?
  - Where should a variable be stored (heap, stack,…)
  - …

(广义的)Semantic Analysis

- The reason of the limitations is that answering those questions depends on values instead of syntax

- We need to analyze the semantics of programs
  - Usually, this is done by traversing/analyzing various program representations
  - Examples of representations: AST, control flow graph (CFG), program dependence graph (PDG), value flow graph (VFG), SSA (single static assignment).

- Sample semantic analysis: type checking, code generation, dead code elimination, register allocation, etc

(狭义的)Semantic Analysis

- **编译原理相关课程默认的语义分析**
- **Determine some static properties of program via AST, e.g.,**
  - Scope and visibility of names
    - every variable is declared before use
  - Types of variables, functions, and expression
    - every expression has a proper type
    - function calls conform to definitions
  - …

```
ex.c:4:5: warning: assignment makes integer from pointer without a cast
ex.c:3:11: error: ‘i’ undeclared (first use in this function)
```

- **Translate the AST into some intermediate code**
  - Section 7 of Tiger book

Outline

1. Symbol Table
2. Symbols in the Tiger Compiler
3. Type Checking

1. Symbol Table

- **Symbol Table**
- **Efficient Implementation**

Symbol Table

- **Binding**: give a symbol a meaning, denotes by $\mapsto$
  - E.g.,

Name/Symbol	**Meaning/Attribute**
type identifier	type (e.g., int , string)
variable identifier	type, value, access info, …
function identifier	args.&reult type, …

- **Environment**: a set of bindings
  - E.g., an environment {g $\mapsto$ string, a $\mapsto$ int}
- Symbol table: the implementation of environment

Semantic analysis: traverse the abstract syntax tree (AST) in certain order while maintaining a symbol table

Motivating Example of Symbol Table

- Suppose at the very beginning the environment is σ_0

```
1. function f(a:int, b:int, c:int) =
2.     (print_int(a+c) ;
3.      let var j := a+b
4.          var a := "hello"
5.        in print(a); print_int(j)
6.      end;
7.      print_int(b)
8.     )
```

Line 1: $\sigma_1 = \sigma_0 + \{a \mapsto \text{int},\ b \mapsto \text{int},\ c \mapsto \text{int}\}$

Line 3: $\sigma_2 = \sigma_1 + \{j \mapsto \text{int}\}$

- In line2, the identifiers **a, c** can be looked up in σ_1
- In lines 3, the environment is updated

Motivating Example of Symbol Table

- Each local variable has a ***scope*** in which it is **visible**.

1.	`function f(a:int, b:int, c:int) =`	$\sigma_1=\sigma_0+\{a \mapsto int, b \mapsto int, c \mapsto int\}$
2.	`    (print_int(a+c) ;`	
3.	`     let var j := a+b`	$\sigma_2=\sigma_1+\{j \mapsto int\}$
4.	`         var a := "hello"`	$\sigma_3=\sigma_2+\{a \mapsto string\}$
5.	`       in print(a); print_int(j)`	
6.	`     end;`	
7.	`     print_int(b)`	
8.	`    )`	

- Let's consider line 4: $\sigma_3 = \sigma_2 + \{a \mapsto string\}$
  - σ2 contains a ↦ int. So, what is the binding of a in σ3?
  - 在σ3中，{a ↦ string}**覆盖**了σ_2 中a的binding!

约定: Bindings in the right-hand table override those in the left

Motivating Example of Symbol Table

- Each local variable has a ***scope*** in which it is **visible**.

1.	function f(a:int, b:int, c:int) =	$\sigma_1=\sigma_0+\{a \mapsto int, b \mapsto int, c \mapsto int\}$
2.	(print_int(a+c) ;	
3.	let var j := a+b	$\sigma_2=\sigma_1+\{j \mapsto int\}$
4.	var a := “hello”	$\sigma_3=\sigma_2+\{a \mapsto string\}$
5.	in print(a); print_int(j)	
6.	end;	σ_2, σ_3 都被“丢弃”
7.	print_int(b)	
8.	)	

- As the semantic analysis reaches **the end of each scope**, the identifier bindings local to that scope are discarded.
  - In line 6, $\boldsymbol{\sigma_3}$ is discarded and $\boldsymbol{\sigma_1}$ is back

小结: Symbol Table需实现的接口

- **insert**: 将名称绑定到相关信息(如类型), 如果名称已在符号表中定义，则新的绑定优先于旧的绑定

- **lookup**: 在符号表中查找名称，以找到名称绑定的信息

- **beginScope**: 进入一个新的作用域

- **endScope**: 退出作用域，将符号表恢复到进入之前的状态

Multiple Symbol Tables

- In some languages there can be several active environments at once: Each module, or class, or record in the program has a symbol table σ of its own.

```
package M;
class E {
    static int a = 5;
}
class N {
    static int b = 10;
    static int a = E.a + b;
 }
class D {
    static int d = E.a + N.a;
 }
```

Java

$\sigma 1 = \{ a \mapsto \textit{int} \}$

$\sigma 2 = \{ E \mapsto \sigma 1 \}$

$\sigma 3 = \{ b \mapsto \textit{int} , a \mapsto \textit{int} \}$

$\sigma 4 = \{ N \mapsto \sigma 3 \}$

$\sigma 5 = \{ d \mapsto \textit{int} \}$

$\sigma 6 = \{ D \mapsto \sigma 5 \}$

$\sigma 7 = \sigma 2 + \sigma 4 + \sigma 6$

In Java, forward reference is allowed. so *E*, *N*, and *D* are all compiled in the environment σ7. The result of the analysis is $\{M \mapsto \sigma 7\}$.

1. Symbol Table

- Symbol Table
- Efficient Implementation

Implementing Symbol Tables

$\sigma_2 = \sigma_1 + \{j \mapsto \text{int}\}$

- **Imperative Style**

  有了新的就看不到老的，但是退出scope时还能回得去

  – Modify σ1 until it becomes σ2

  – While σ2 exists, we cannot look things up in σ1

  – When we are done with σ2, we can undo the modification to get σ1 back again

- **Functional Style**

  每次发生变化的时候老的都保留着

  – Keep σ1 in pristine condition while creating σ2 and σ3

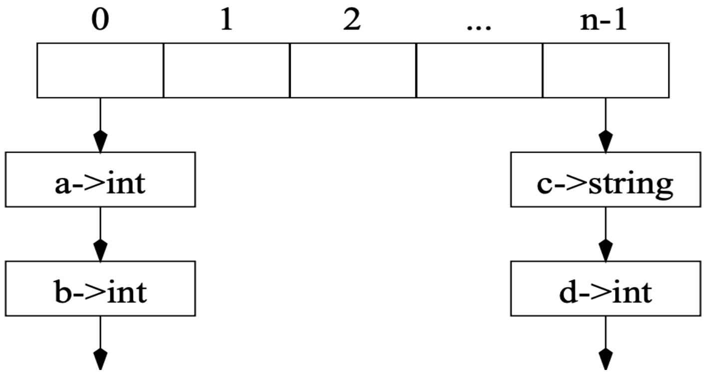
  – Easy to restore σ1

Either style can be used.

Problem Statement: Imperative Symbol Tables

- **Problem 1: How to support efficient lookup?**
  - **Solution:** Use hash table!
  - E.g., $\sigma' = \sigma + \{a \mapsto int\}$: insert <a, int> in the hash table

- **Problem 2: How to support deletion and revert to previous environment?**
  - **Solution**: Hash Table with external chaining
  - E.g., hash(a) -> <a, int> -> <a, string>



Efficient Imperative Symbol Tables

- **Hash Tables** - **an array of buckets**
- **Bucket** - **list of entries** (each entry maps identifier to binding)

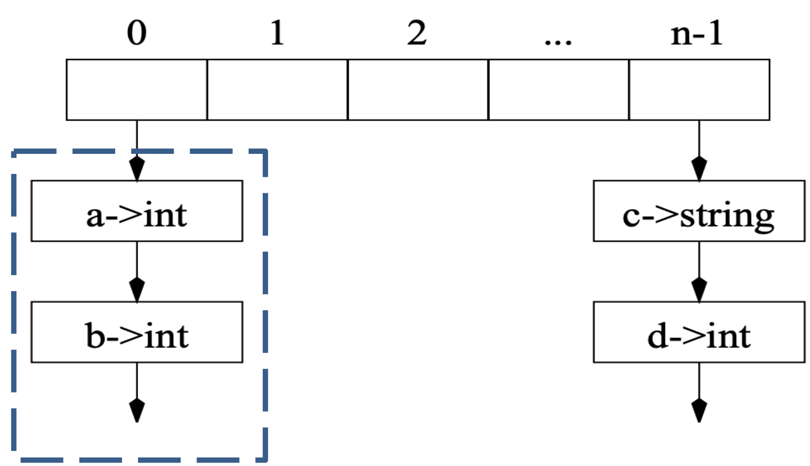
```
struct bucket { string key; void *binding;
struct bucket *next; };
#define SIZE 109

struct bucket *table[SIZE];

unsigned int hash(char *s0)
{ unsigned int h=0; char *s;
  for(s=s0; *s; s++)
    h=h*65599 + *s;
  return h;
}

struct bucket *Bucket (string key, void
*binding, struct bucket *next) {
  struct bucket *b=checked_malloc(sizeof(*b));
  b->key = key; b->binding = binding; b->next
= next;
  return b; }
```

$h = (\alpha^{n-1}c_1+\alpha^{n-2}c_2 +\ldots+ \alpha\, c_{n-1}+ c_n)$



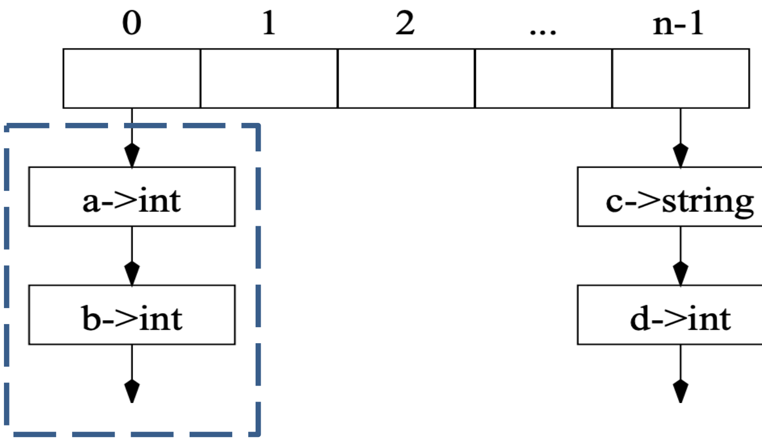
bucket

Efficient Imperative Symbol Tables

- Lookup entry for identifier *key* in symbol table:
  1. Apply *hash function* to `key` for getting array element `index`
  2. Traverse bucket in `table[index]` to find the binding.

     (table[*x*]: all entries whose keys hash to *x*)

```
void *lookup(string key) {
  int index=hash(key)%SIZE
  struct bucket *b;
  for (b = table[index]; b; b=b->next)
    if (0==strcmp(b->key,key))
      return b->binding;
  return NULL;
}
```

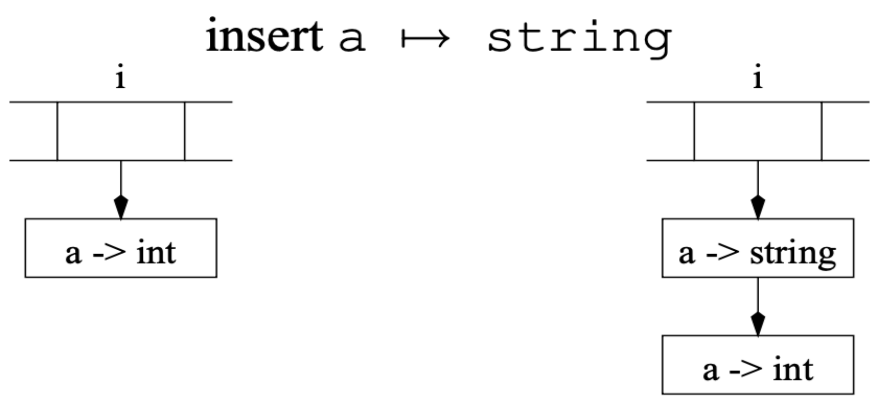


E.g., 假设想要查找标识符a的类型

1. 假定hash(c) = 0
2. 从table[0]的第一个元素开始逐一比较

Efficient Imperative Symbol Tables

- Insert new element at front of bucket
  - Consider $\sigma + \{a \mapsto \tau 2\}$ when σ contains $a \mapsto \tau 1$ already.
  - The insert function leaves $a \mapsto \tau 1$ in the bucket and puts $a \mapsto \tau 2$ earlier in the list: hash(a) -> <a, τ2 > -> <a, τ1>




```
void insert(string key, void *binding) {
  int index=hash(key)%SIZE;
  table[index]=Bucket(key, binding, table[index]);
}
```

Efficient Imperative Symbol Tables

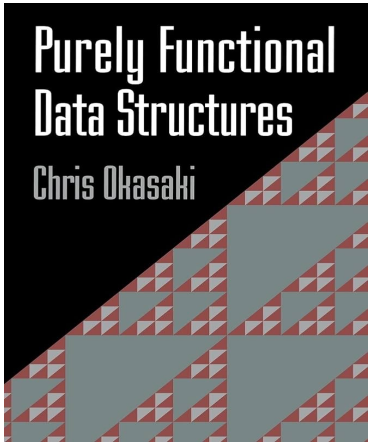
- Restore: pop items off items at front of bucket.
  - Consider $\sigma + \{a \mapsto \tau 2\}$ when σ contains $a \mapsto \tau 1$ already.
  - When pop(a) is done at the end of a's scope, σ is restored. (insertion and pop work in a stack-like fashion)
    - E.g., hash(a) -> <a, τ2 > -> <a, τ1> 变成 hash(a) -> <a, τ1>

```
void pop(string key) {
  int index=hash(key)%SIZE
  table[index]=table[index].next;
}
```

注: 为了处理不同scope, 还需其他辅助信息，比如指导“需要pop几次”(见第2部分 “Symbols in the Tiger Compiler”)

Efficient Functional Symbol Tables

- Basic idea the approach:
  - When implementing `σ2 = σ1 + {x ↦ t}`
  - Creating a **new** table `σ2`, instead of modifying `σ1`
  - When deleting, restore to the old table (方便快速"回退")
- 函数式编程中的"不可变数据结构"思想



https://en.wikipedia.org/wiki/Purely_functional_data_structure

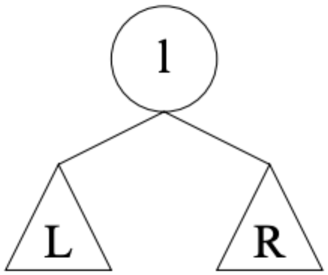
https://xavierleroy.org/CdF/2022-2023/1.pdf

Efficient Functional Symbol Tables

- Basic idea the approach:
  - When implementing **σ2 = σ1 + {x ↦ t}**
  - Creating a **new** table **σ2**, instead of modifying **σ1**
  - When deleting, restore to the old table

- 实现方式: binary search trees (BSTs).
  - Each node contains mapping from identifier (key) to binding.
  - Use string comparison for "less than" ordering.
  - For all nodes n ∈ L, key(n) < key(l)

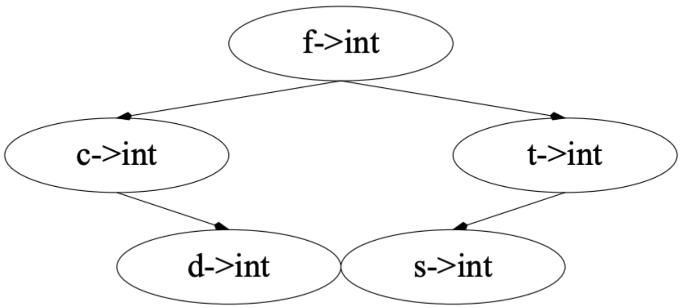
    For all nodes n ∈ R, key(n) > = key(l)



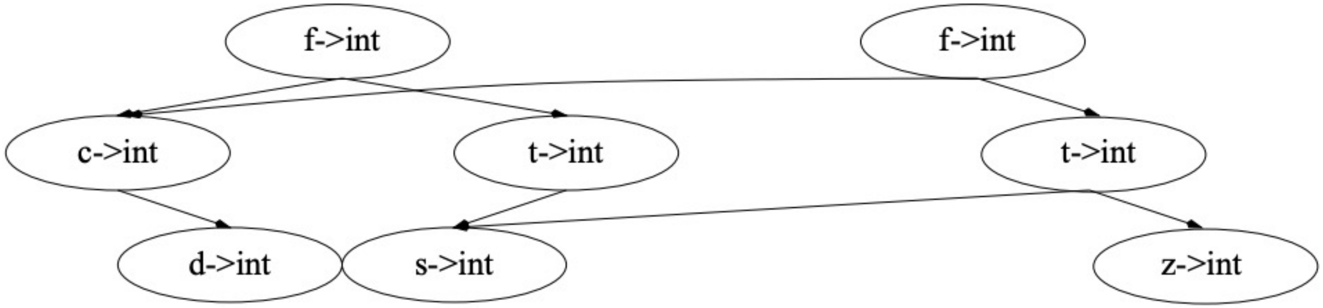
注: 用上述efficient impressive的方式也能实现，但是不够高效(见附录/虎书)

Efficient Functional Symbol Tables

- 实现方式: binary search trees (BSTs)
- **Lookup**： Olog(n) for a balanced tree of n nodes.



- **Insert:** Copy the nodes from the root to the parent of the inserted node
  - insert z → int, create node z, copy all ancestors of z



目的/效果: 避免完整拷贝所有旧版本

Summary: Implementation of Symbol Tables

- **Imperative Style : (side effects)**
  - When beginning-of-scope entered, entries added to table using side-effects. (old table destroyed)
  - When end-of-scope reached, auxiliary info used to remove previous additions. (old table reconstructed)
- **Functional Style : (no side effects)**
  - When beginning-of-scope entered, new environment created by adding to old one, but old table remains intact
  - When end-of-scope reached, retrieve old table.

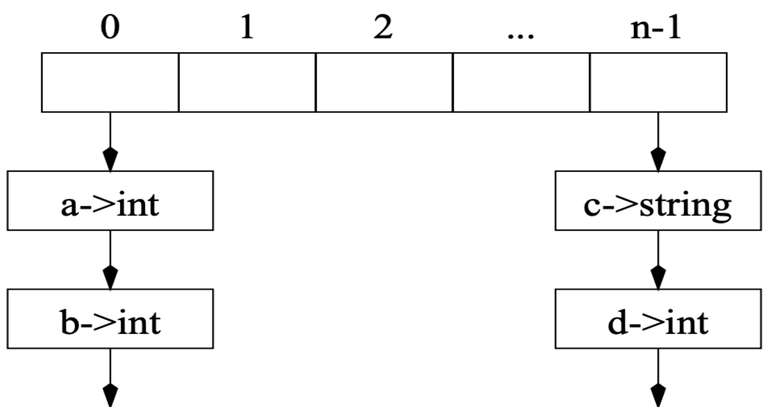
它们对lookup, insert, beginScope, endScope等接口的支持，复杂度上各有优势

2. Symbols in the Tiger Compiler

Issues With Table Implementations

- **Problem**: When key value = string, wee need expensive string comparisons when doing lookup operation.

```
void *lookup(string key) {
  int index=hash(key)%SIZE
  struct bucket *b;
  for (b = table[index]; b; b=b->next)
    if (0==strcmp(b->key,key))
      return b->binding;
  return NULL;
}
```



- **Solution**: use *symbol* data structure instead
  - Each symbol object associated with integer value
  - All occurrences of same string map onto same symbol (2 different strings map onto different symbols)
  - key value = symbol ⇒ do cheap integer comparisons during lookup

Symbols in The Tiger Compiler

The interface of symbols and symbol tables

- void *: We want different notions of binding for different purposes in the compiler – type bindings for types, value bindings for variables and functions

```
typedef struct S_symbol_ *S_symbol;
S_symbol S_symbol (string);
string S_name(S_symbol);

typedef struct TAB_table_ *S_table;
S_table S_empty( void);
void S_enter( S_table t,S_symbol sym, void *value);
void *S_look( S_table t, S_symbol sym);
void S_beginScope( S_table t);
void S_endScope( S_table t);
```

The Implementation of Symbols

```
static S_symbol mksymbol (string name , S_symbol next) {
  S_symbol s = checked_malloc(sizeof(*s));
  s->name = name; s->next = next;
  return s;
}

S_symbol S_symbol (string name) {
        int index = hash(name)%SIZE;
        S_symbol syms = hashtable[index], sym;
        for ( sym = syms; sym; sym = sym->next)
          if (0 == strcmp(sym->name, name)) return sym;
        sym = mksymbol(name,syms);
        hashtable[index] = sym;
   return sym;
}

string S_name (S_symbol sym) {
  return sym->name;
}
```

The Implementation of Symbol Tables

- The Tiger compiler in C uses destructive-update environments.
- An imperative table is implemented using a hash table.

```
// make a new S_Table
S_table S_empty(void) {
  return TAB_empty();
}
// insert a binding
void S_enter(S_table t, S_symbol sym, void
*value){
  TAB_enter(t,sym,value);
}
// look up a symbol
void *S_look(S_table t, S_symbol sym) {
  return TAB_look(t,sym);
}
```

The Implementation of Symbol Tables

For destructive-update environments:

- **S_beginScope**: Remembers the current state of the table
- **S_endScope**: Restores the table to where it was at the most recent beginScope that has not already been ended.

```
static struct S_symbol_ marksym = { “<mark>”, 0 };

void S_beginScope ( S_table t) {
  S_enter(t, &marksym, NULL);
}

void S_endScope( S_table t) {
  S_symbol s;
  do
    s= TAB_pop(t);
  while (s != &marksym);
}
```

The Implementation of Symbol Tables

Auxiliary stack:

- Showing in what order the symbols were “pushed” into the symbol table.
- As each symbol is popped, the head binding in its bucket is removed.
- **beginScope**: pushes a special marker onto the stack
- **endScope**: pops symbols off the stack until finds the topmost marker.

The Implementation of Symbol Tables

- The auxiliary stack can be integrated into the Binder by having a global variable top showing the most recent Symbol bound in the table.
- Pushing: copy top into the prevtop field of the Binder.

```
struct TAB_table_ {
  binder table[TABSIZE];
  void *top;
};


static binder Binder(void *key, void *value, binder next, void
*prevtop) {
  binder b = checked_malloc(sizeof(*b));
  b->key = key; b->value=value; b->next=next;
  b->prevtop = prevtop;
  return b;
}
```

3. Type Checking

- 类型及其作用
- Tiger类型系统
- Tiger类型检查

编程语言中的类型

- **变量的类型**
  - 限定了变量在程序执行期间的取值范围
- **类型化的语言(typed language)**
  - 变量都被给定类型的语言，如C/C++、Java、Go
  - 表达式、语句等程序构造的类型都可以静态确定
- **未类型化的语言(untyped language)**
  - 不限制变量值范围的语言, 如 LISP、JavaScript

no static types,
而非没有类型

类型在编程语言中的作用

- **开发效率：更高层次的编程抽象, e.g.,**
  - 多态、代数数据类型、依赖类型…
  - hoogle利用类型信息搜索API

编得快

- **运行性能：类型指导的编译优化, e.g.,**
  - 静态类型绑定避免运行时检查
  - 类型信息优化内存布局

跑得快

- **安全可靠：内存安全乃至功能正确, e.g.,**
  - Rust线性类型保障内存安全
  - LiquidHaskell精化类型保障功能正确

信得过

例: Refinement Type类型签名 add_two (x : int | x> 0) → x + 2

类型系统的形式化(不要求掌握)

- 类型系统是一种逻辑系统

有关自然数的逻辑系统

- 自然数表达式

  a+b, 3

- 良形公式

  a+b=3, (d=3) ∧ (c<10)

- 推理规则

$$\frac{a < b, \quad b < c}{a < c}$$

程序语言的类型系统

- 类型表达式

  int, int → int

- 定型断言(typing assertion)

  $x: \text{int} \mid\!- \ x+3 : \text{int}$

- 定型规则(typing rules)

$$\frac{\Gamma \mid\!- M : \text{int}, \ \Gamma \mid\!- N : \text{int}}{\Gamma \mid\!- M + N : \text{int}}$$

用推理规则来确定有哪些类型、
表达式的类型是什么

2. Type Checking

- 类型及其作用
- **Tiger类型系统**
- Tiger 类型检查

Key Problems in Type Checking

- **What are valid type expressions ?**
  - 总共有哪些类型，每个类型表达式对应到什么?
  - e.g., int, string, unit, nil, array of int, record ...
- **How to define two types are equivalent ?**
  - 比如，两个record类型是否相同?
- **What are the typing rules ?**
  - 要检查什么，比如形参和实参是否一致?

Tiger语言总共有哪些类型

- **The primitive type**: int, string
- **The constructed type**: using records and arrays from other types (primitive, record, or array)

typec → *type* *type-id* = *ty*

ty → *type-id*

→ '{' *tyfields* '}'

→ *array of* *type-id*

tyfields → ε

→ *id*: *type-id* {, *id*:*type-id*}

类型的文法定义

```
let type a = {x: int; y: int}
    type b = {x: int; y: int}
    var i : a := ...
    var j : b := ...
in i := j
end
```

类型声明案例

Tiger语言总共有哪些类型

- What are the binding of type identifiers and expresions:?

typec → *type* *type-id* = *ty*

ty → *type-id*

→ '{' *tyfields* '}'

→ *array of* *type-id*

tyfields → ε

→ *id: type-id {, id:type-id}*

```
typedef struct Ty_ty_ *Ty_ty;
struct Ty_ty_ {
  enum {Ty_record, Ty_nil, Ty_int, Ty_string,
        Ty_array, Ty_name, Ty_void} kind;
  union {
    Ty_fieldList record;
    Ty_ty array;
    struct {S_symbol sym; Ty_ty ty;} name;
  } u;
};
```

- When processing mutually recursive types: Ty_Name(sym, NULL)
    - a place-holder for the type-name sym

Tiger语言中的类型等价

- 注意: Every Tiger-language “record type expression” creates a new (and different) record type !

```
let type a = {x: int; y: int}
    type b = {x: int; y: int}
    var i : a := ...
    var j : b := ...
in i := j
end
```

It is illegal in Tiger.

```
let type a = {x: int; y: int}
    type b = a
    var i : a := ...
    var j : b := ...
in i := j
end
```

It is legal in Tiger.

类型等价Type Equivalence

- **Name equivalence (NE)** T1 and T2 are equivalent iff T1 and T2 are identical type names defined by the exact same **type declaration**.
- **Structure equivalence (SE)** T1 and T2 are equivalent iff T1 and T2 are composed of the same constructors applied in the same order.
  - **Tiger uses name equivalence.** E.g., `point` and `ptr` are equivalent under SE but not equivalent under NE

```
type point = {x : int, y : int}
type ptr = {x : int, y : int}
function f(a : point) = a
```

注: Type equivalence影响类型检查，如是否需要在Typing environment增加新的类型? 函数的形参和实参类型识别匹配?

Tiger语言的命名空间

- Tiger has two separate name spaces:
  - **Types**
  - **Functions** and **variables**

```
let type a = int
    var a := 1
in ...
end
```

Both a's can be used

```
let function a (b: int) = ...
    var a := 1
in ...
end
```

var a hides function a

3. Type Checking

- 类型及其作用
- Tiger类型系统
- Tiger 类型检查

Environments for Type Checking

```
let type a = int
    var a: a:= 5
    var b: a:= a
in b+a
end
```

- Tiger语义分析需要维护2个环境:
  1. **Type environment**

     Maps type symbol to type that it stands for Symbol -> Ty_ty
  2. **Value environment**
  - Maps variable symbol to its type symbol -> Ty_ty
  - Maps function symbol to parameter and result types

    symbol -> struct {Ty_tyList formals, Ty_ty result;}

Environments for Type Checking

```
typedef struct E_enventry_ *E_enventry;
struct E_enventry_ {
  enum {E_varEntry, E_funEntry} kind;
  union {
    struct {Ty_ty ty;} var;
    struct {Ty_tyList formals; Ty_ty result;} fun;
  } u;
};

E_enventry E_VarEntry(Ty_ty ty);
E_enventry E_FunEntry(Ty_tyList formals, Ty_ty result);

S_table E_base_tenv(void); /* Ty_ty environment */
S_table E_base_venv(void); /* E_enventry environment */
```

- Predefined type and value environments
  - Type environment "int" ↦ Ty_int, "string" ↦ Ty_string.
  - Value environment contains predefined functions of Tiger

Type-Checking for Tiger

- The Semant module (semant.h, semant.c) performs semantic analysis including type-checking – of abstract syntax

- Semantic analysis: four recursive functions over AST
  - 既做语义检查又做中间代码(IR Code)生成

```
Struct expty transVar (S_table venv, S_table tenv, A_var v);
Struct expty transExp (S_table venv, S_table tenv, A_exp a);
Void transDec (S_table venv, S_table tenv, A_dec d);
Ty_ty transTY (S_table tenv, A_ty a);
```

For now, not concerned with translation into IR code

Type-Checking for Tiger

1. **Type-checking expressions**
2. **Type-checking declarations**
   - Variable declarations
   - Type declarations
   - Function declarations
   - Recursive type declarations
   - Recursive function declarations

1. Type-Checking Expressions

```
Struct expty transVar (S_table venv, S_table tenv, A_var v);
Struct expty transExp (S_table venv, S_table tenv, A_exp a);
Void transDec (S_table venv, S_table tenv, A_dec d);
Ty_ty transTY (S_table tenv, A_ty a);
```

- **transExp :** Type-checking expressions: query && update the environments
- **Input** venv: value env., tenv: type env., a: expression

  Output: a translated expression and its Tiger-language type

```
struct expty {Tr_exp exp; Ty_ty ty;};
```

Example: Type-Checking e1 + e2

- Tiger's nonoverloaded type-checking for '+' expression
  - Suppose we we want to type-check e1 + e2
  - Both e1, e2 must be ints; Type of the expression is int

```
struct expty transExp(S_table venv, S_table tenv, A_exp a) {switch(a->kind)
{
   ...
   case A_opExp: {
     A_oper oper = a->u.op.oper;
     struct expty left =transExp(venv,tenv,a->u.op.left);
     struct expty right=transExp(venv,tenv,a->u.op.right);
     if (oper==A_plusOp) {
       if (left.ty->kind!=Ty_int)
         EM_error(a->u.op.left->pos, "integer required");
       if (right.ty->kind!=Ty_int)
         EM_error(a->u.op.right->pos,"integer required");
       return expTy(NULL,Ty_Int());
     }...
   }
 }
 assert(0); /* should have returned from some clause of the switch */
}
```

Rules for Type-Checking Expressions

- **Function call**: the types of formal parameters must be equivalent to the types of actual arguments.
- **If-expression** : **`if`** exp_1 **`then`** exp_2 **`else`** exp_3

  The type of exp_1 must be integer, the types of exp_2 and exp_3 should be equivalent.
- …

For more info, read **Appendix in Tiger Book.**

Type-Checking for Tiger

1. Type-checking expressions

2. **Type-checking declarations**
    - Variable declarations
    - Type declarations
    - Function declarations
    - Recursive type declarations
    - Recursive function declarations

2. Type-Checking Declarations

- **Declarations modify environments!**
- In Tiger, declarations appear only in a let expression.

```
struct expty transExp (S_table venv, S_table tenv, A_exp a)
{
  switch(a->kind) {
    ...
    case A_letExp: {
      struct expty exp;
      A_decList d;
      S_beginScope(venv); S_beginScope(tenv);
      for (d = a->u.let.decs; d; d=d->tail)
        transDec(venv,tenv,d->head);
      exp = transExp(venv,tenv,a->u.let.body);
      S_endScope(tenv); S_endScope(venv);
      return exp;
    }
  }...
}
```

- 变量声明、类型声明、函数声明
- let中声明的可以用在in里面

Variable Declarations

- Processing a variable declaration without a type constraint, e.g.,

var x := exp

```
void transDec(S_table venv, S_table tenv, A_dec d) {
  switch(d->kind) {
    case A_varDec: {
      struct expty e = transExp(venv,tenv,d->u.var.init);
      S_enter(venv, d->u.var.var, E_VarEntry(e.ty));
    }
  ...
  }
  ...
```

exp的类型ty(通过transExp获得)就是x的类型

Variable Declarations

- Processing a variable declaration with a type constraint:

  var x : type-id := exp

  - It will be necessary to check that the constraint and the initializing expression are compatible.
  - Also, initializing expressions of type Ty_Nil must be constrained by a Ty_Record type.

Type Declarations

- Nonrecursive type declarations

type type-id = ty

  - transTy: translates A_ty to Ty_ty recursively

直接利用抽象语法树(AST)节点上的类型信息(A_ty)

```
void transDec(S_table venv, S_table tenv, A_dec d) {
  ...
  case A_typeDec: {
    S_enter(tenv, d->u.type->head->name, transTy(d->u.type->head->ty));
  }
  ...
}
```

NOTICE: This program fragment only handles the type-declaration list of length 1!

Function Declarations

- Nonrecursive function declarations

function id (tyfields) : type-id = exp

```
void transDec(S_table venv, S_table tenv, A_dec d)
{
  switch(d->kind) {
    ...
    case A_functionDec: {
      A_fundec f = d->u.function->head;
      Ty_ty resultTy = S_look(tenv, f->result);
      Ty_tyList formalTys =
makeFormalTyList(tenv,f->params);
      S_enter(venv, f->name,
E_FunEntry(formalTys,resultTy));
      S_beginScope(venv);
      {A_fieldList l; Ty_tyList t;
       for(l=f->params, t=formalTys; l; l=l->tail,
t=t->tail)
         S_enter(venv,l->head->name,E_VarEntry(t-
>head));
      }
      transExp(venv, tenv, d->u.function->body);
      S_endScope(venv);
      break;
    }
    ...
```

```
function f(a:int):int = body
```

1. Look up ‘int’ in type env, ‘int’ $\rightarrowtail$Ty_int
2. venv’= venv + f $\rightarrowtail${ formals = Ty_int, result = Ty_int}
3. venv’’ = venv’ + a $\rightarrowtail$Ty_int
4. Type-check the body in {tenv, venv’’} using transExp
5. Return {tenv, venv’} for use in processing expressions which refer to f

NOTICE: The above code handles very restricted functions

Function Declarations

- The above code is very stripped-down implementation:
  - it handles only the case of a single function;
  - it handles only a function with a result
  - it doesn't handle program errors
  - it doesn't check that the type of the body expression matches the declared result type
  - …
- ***function f(a: ta, b: tb) : rt = body***
- ***makeFormalTyList***: traverses the list of formal parameters and returns a list of their types

Recursive Type Declarations

- How to convert the following declaration into the internal type representations ?

  type list = {first: int, rest: list}

- **Problem**: Processing type list = {first: int, rest: list} requires to lookup list from the type environment
  - *undefined type!*

- How to handle this problem?

Recursive Type Declarations

```
struct {
      S_symbol sym;
      Ty_ty ty;}
```

type list = {first: int, rest: list}

- **Solution: use two passes**
  - **1.** Put all the "headers" in the environment first (though they do not have bodies)
    - What is a header in this example? **type list =**
    - Use the special "name" type for the header
  - 2. Call transTy on the "body" of the type declaration
    - That is, the record type expression **{first: int, rest: list}**
    - The type that transTy returns can then be assigned into the ty field within the Ty_Name struct.

Recursive Type Declarations

- Every cycle in a set of mutually recursive type declarations **MUST** pass through a record or array declaration！
  - Tiger语言的重要特性之一

Tiger
```
type a = b
type b = a
```

It is illegal in Tiger.

Tiger
```
type a = b
type b = {i: a}
```

Tiger
```
type a = b
type b = d
type c = a
type d = a
```

It is illegal in Tiger.

contains an illegal cycle:
a -> b -> d -> a

Otherwise, the type checker will not terminate!

Recursive Function Declarations

- **Mutually recursive functions** handled similarly

  f call g, g call f

- **Problem**: P when we process the right hand side of function declarations, we may encounter symbols that are not defined in the env yet
- **Solution**
  - First pass: gathers information about the *header* of each function but leaves the bodies of the functions untouched.
  - Second pass: processes the bodies of all functions with the augmented environment.

Thank you all for your attention

Efficient Functional Symbol Tables

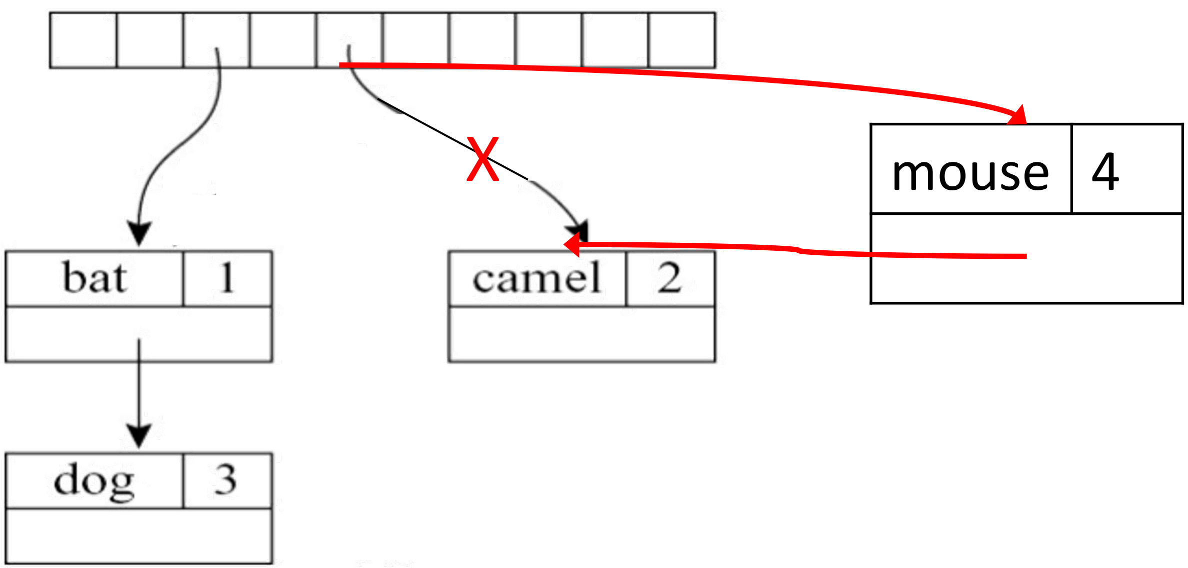
Functional Style

- We wish to compute $\sigma' = \sigma + \{a \mapsto \tau\}$ in such a way that we still have σ available to look up identifiers.
- We create a new table by computing the "sum" of an existing table and a new binding.

- Can we achieve this with hash tables?

Efficient Functional Symbol Tables

- $m2 = m1 + \{mouse \mapsto 4\}$
- $m1 = \{bat \mapsto 1, camel \mapsto 2, dog \mapsto 3\}$, suppose index(camel) = index(mouse) = 5
- Hash(mouse) ==> <mouse, 4>: no longer have m1



Efficient Functional Symbol Tables

- m2 = m1 + {mouse ↦ 4}
- m1 = {bat ↦ 1,camel ↦ 2,dog ↦ 3}, suppose index(camel) = index(mouse) = 5
- Hash(mouse) ==> <mouse, 4>: no longer have m1
- Copy the array, but share all the old buckets: not efficient
  - The array in a hash table should be quite large