

Type System

- collection of types in programming language
- purpose
 - run-time safety, expressiveness (e.g., overloading), better code (e.g., more information to select most efficient instructions)
- type-checking
 - assigning or inferring of types, and checking for type-related errors

- Symbol table has names – what do these mean?
- names can be identified during syntactical analysis, but to determine meaning – semantic analysis
 - what kind of value is in x? number (how many bytes, representation), procedure (parameter and result types), (how long should x be preserved...)
 - syntax can force correct declaration – but have we actually “seen” x before its use?
- Semantic analysis goes beyond context-free syntax

- base types
 - (INT, STRING; int, char, bool)
- rules for constructing new types
 - (arrays, structures)
- determining $ty1 == ty2$
 - structural or name equivalence
 - “generative” (datatype ... in ML)
- rules for inferring type for each exp.
 - for each operator mapping between operand type and result (rules for conversion)
 - vars – have declared type, constants – implied
 - func – look at decl & compare actual and formal params
 - often bottom up – can perform with parsing
 - in general – need to establish relationships between type properties that may be influenced by other nodes (children, parents, siblings, self...) and ‘iterate’.

- in Tiger – different name spaces for types and {vars U funcs}
- create environments:
 - type: symbol -> Types.ty creates Env
 - base_tenv (INT, STRING)
 - enventry holds attributes for vars/funcs
 - var/func: symbol -> enventry
 - venv: predefined functions (ord, chr, size...)
- need to interpret meaning of vars, exps, decls, and type declarations.
- inference rules correspond to recursion in tree-walking code for type checker
 - can have a single procedure with specialized code for each node variant

- How does it work:
 - build symbol table with attributes
 - resolve references to names and find errors
 - same name in a contour, need type name have a var name...
 - walk tree again to assign types to expression nodes
 - check for errors

- need to establish binding <name, attributes> and build environments
- for Tiger, 2 environments, types and values (variables + functions)
- use these attributes to perform semantic checks:
 - type checking, operand overload, break from nested loops, check if all identifiers are defined...

Tiger Environments

```
signature TC_ENV =
sig
  type ty = Types.ty
  (* bindings (table entries) for variable environments *)
  datatype enventry
    = VAREntry of {ty: ty}
    | FUNEntry of {formals: ty list, result: ty}
  type tenv = ty Symbol.table
    (*type environment, giving types associated with
    type names*)
  type venv = enventry Symbol.table
    (*variable environments, giving types for declared
    variables and functions*)

  type env = tenv * venv
  val base_env : env
    (*contains predefined types (tenv) and functions (venv)*)
end (* signature TC_ENV *)
```

- Typechecker must translate all source level type specs into Types.ty representation
- must assign type to all vars and exps
- unique used for NAME – for recursive types
- need to establish rules and operations for Tiger type system! (project 5)

Types

```
(* types.sml *)
structure Types =
struct
  type unique = unit ref
  (* values used to provide unique type identities for
  * declared record and array types *)

  datatype ty
    = NIL (* null record, matches any record type *)
    | UNIT (* represents no value *)
    | INT (* primitive integer type *)
    | STRING (* primitive string type *)
    | RECORD of (Symbol.symbol * ty) list * unique
    | ARRAY of ty * unique
    | NAME of Symbol.symbol * ty option ref
      (* for forward references *)
    | ERROR (* for error recovery *)
end (* structure Types *)
```

- *unique* is used to provide unique identities for declared record and array types. These identities will be tested to determine type equality.
- *NIL* and *UNIT* are used internally, and are not directly expressible in the source code
- All type declarations cause type ids to be bound to *NAME* types, initially of form *NAME*(id, ref *NONE*), with the ref assigned the appropriate (*SOME* ty) value in a second pass.

Type-checking Exp

```
type tenv = Types.ty Symbol.table
type env = enventry Symbol.table

(* transexp : env * tenv -> exp -> ty *)
fun transexp (env,tenv) =
  let fun g(OpExp{left,oper=A.plusOp,right,pos}) =
      (checkInt(g left, pos);
       checkInt(g right, pos);
        Types.INT)
      | g(LetExp{decs, body, pos}) =
          let val (env',tenv') =
              transdecs (env,tenv) decs
              in transexp (env',tenv') body
              end
          end
      | ....
  in g
  end
```

Type-Checking Declarations

```
(* transdec : env * tenv -> dec -> env * tenv *)
fun transdec (env,tenv) =
  let fun g(VarDec{var,typ=NONE,init}) =
      let val ty = transexp (env,tenv) init
          val b = VAREntry{access=(),ty=ty}
          in (enter(env,var,b), tenv)
          end
      | g(FunctionDec[{name,params,body,pos,result=_}]) =
          let val b = FUNEntry{...}
              val env' = enter(env,name,b)
              val env'' = enterparams(params,env')
              in transexp (env'',tenv) body;
              (env', tenv)
              end
          end
      | g ...
  in g
  end
```

```
(* transdecs : env * tenv -> dec list -> env * tenv *)
fun transdecs (env, tenv) [] = (env,tenv)
  | transdecs (env, tenv) (a::r) =
    let val (env', tenv') = transdec (env, tenv) a
        in transdecs (env', tenv') r
    end
```

Tiger rules

- type equality – name, but also type $a = c!$
- variable decl type var: type-id ...
 - the type of the variable has to match type-id
- array subscript must be int
- if $exp1$ then $exp2$ else $exp3$ – $exp1$ must be integer, $exp2$ and $exp3$ must be of same type
- ...
- see Appendix

Recursive Type Declarations

- How to convert the following declaration into the internal type representations ?
`type list = {first : int, rest : list}`
- **Problem:** when we do the conversion of the r.h.s., “list” is not defined in the tenv yet.
- **Solution:** use the special **Name** type
`datatype ty = NAME of Symbol.symbol * ty option ref |`
- First, enter a “header” type for list `val tenv' = enter(tenv,name, NAME(name,ref NONE))`
- Then, we process the body (i.e., r.h.s) of the type declarations, and assign the result into the reference cell in the **NAME** type

- **Problem:** when we process the right hand side of function declarations, we may encounter symbols that are not defined in the env yet

```
function do_nothing1(a: int, b: string)= do_nothing2(a+1)
function do_nothing2(d: int) = do_nothing1(d, “str”)
```

- **Solution:** first put all function names (on the l.h.s.) with their header information (e.g., parameter list, function name, type, etc., all can be figured out easily) into the env ----- then process each function’s body in this augmented env.

BREAK

- add another attribute – boolean whether or not you are in a loop, whether or not you can break.
- perform corresponding checks