



Type Checking

Cosmin E. Oancea

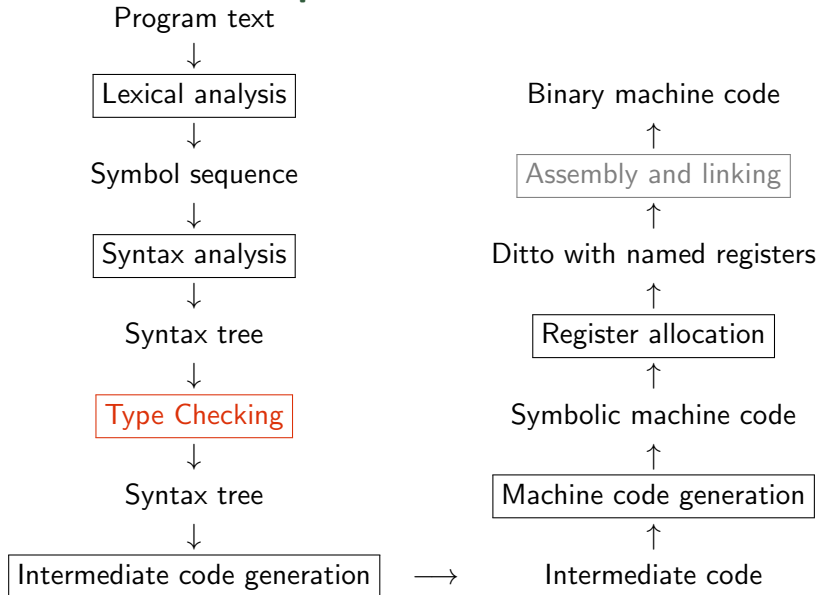
`cosmin.oancea@diku.dk`

Department of Computer Science (DIKU)
University of Copenhagen

February 2018 Compiler Lecture Notes

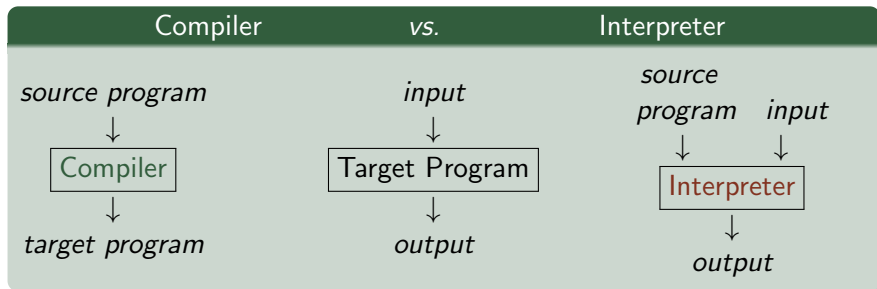


Structure of a Compiler



- 1 Interpretation Recap: Synthesized/Inherited Attributes
- 2 Type-System Characterization
- 3 Type Checker for FASTO Without Arrays (Generic Notation)
- 4 Advanced Concepts: Type Inference
- 5 Type Checker for FASTO With Arrays (F# Code)

Interpretation Recap



*The interpreter directly executes one by one the operations specified in the *source program* on the *input* supplied by the user, by using the facilities of its implementation language.*

Synthesized vs Inherited Attributes

A compiler phase consists of one or several traversals of the `ABSYN`. We formalize it via *attributes*:

Inherited: info passed downwards on the `ABSYN` traversal, i.e., from root to leaves. Think: helper structs. **Example?**

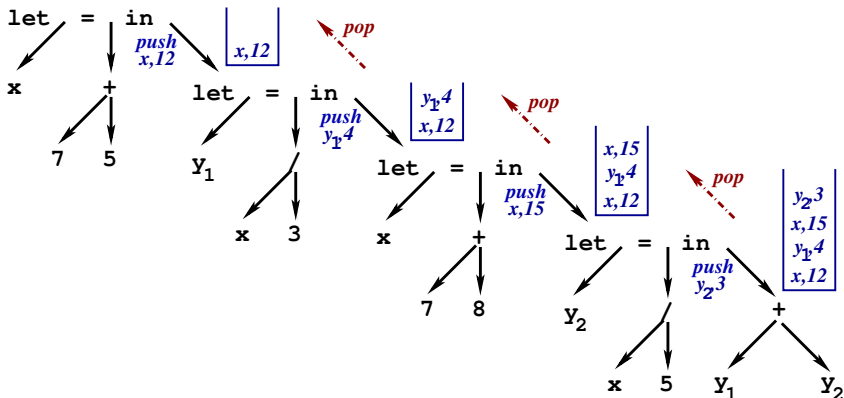
Synthesized: info passed upwards in the `ABSYN` traversal, i.e., from leaves to the root. Think: the result. **Example?**

Both: Information may be synthesized from one subtree and may be inherited/used in another subtree (or at a latter parse of the same subtree). **Example?**

Example of Inherited Attributes

The variable and function symbol tables, i.e., *vtable* and *ftable*, in the interpretation of an expression:

$$Eval_{Exp}(Exp, vtable, ftable) = \dots$$



Example of Synthesized Attributes

The interpreted value of an expression / program is synthesized.

Example of both *synthesized* and *inherited* attributes:

$$vtable = Bind_{Typelds}(Typelds, args)$$
$$ftable = Build_{ftable}(Funs)$$

and used in the interpretation of an expression.

Interpretation vs Compilation Pros and Cons

Interpretation vs Compilation Pros and Cons

- + Simple (good for impatient people).
- + Allows easy modification / inspection of the program at run time.
- Typically, it does not discover all type errors. **Example?**
- Inefficient execution:
 - Inspects the SYMTAB repeatedly, e.g., symbol table lookup.
 - Values must record their types.
 - The same types are checked over and over again.
 - No “global” optimizations are performed.

Idea: Type check and optimize as much as you can statically, i.e., before running the program, and generate optimized code.

- 1 Interpretation Recap: Synthesized/Inherited Attributes
- 2 Type-System Characterization**
- 3 Type Checker for FASTO Without Arrays (Generic Notation)
- 4 Advanced Concepts: Type Inference
- 5 Type Checker for FASTO With Arrays (F# Code)

Type System / Type Checking

Type System: a set of logical rules that a legal program must respect.

Type Checking verifies that the type system's rules are respected.

Example of type rules and type errors:

- $+$, $-$ expect integral arguments: `a + (b==c)`
- *if-branch expressions have the same type:*
`let a = (if (b == 3) then 'b' else 11) in ...`
- *the type and number of formal and actual arguments match:*
`fun int sum ([int] x) = reduce(op +, 0, x)`
`fun [bool] main() = map(sum, iota(4))`
- other rules?

Some language invariants cannot be checked statically: Examples?

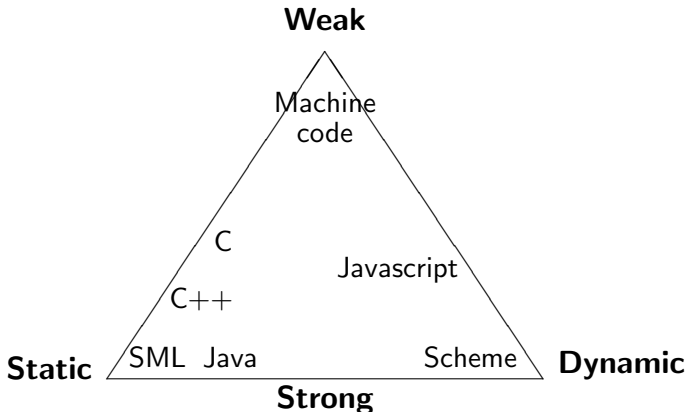
Type System

Static: Type checking is performed before running the program.

Dynamic: Type checking is performed while running the program.

Strong: All type errors are caught.

Weak: Operations may be performed on values of wrong types.



Type Rules

Specify the type constraints and a way to derive the type of an expression, based on the types of its constituent subexpressions.

`map` : $\forall \alpha. \forall \beta. ((\alpha \rightarrow \beta) * [\alpha]) \rightarrow [\beta]$. Type rule for `map(f, x)`:

Type Rules

Specify the type constraints and a way to derive the type of an expression, based on the types of its constituent subexpressions.

$\text{map} : \forall \alpha. \forall \beta. ((\alpha \rightarrow \beta) * [\alpha]) \rightarrow [\beta]$. Type rule for $\text{map}(f, x)$:

- compute t , the type of (arbitrary expression) x , and check that $t \equiv [t_{el}]$ for some t_{el} .
- get f 's signature from $f\text{table}$. IF f does not receive exactly one arg THEN $\text{error}()$ ELSE $f : t_{in} \rightarrow t_{out}$, for some t_{in} and t_{out} .
- IF $(t_{el} \equiv t_{in})$ THEN $\text{map}(f, x) : [t_{out}]$, ELSE $\text{error}()$

$\text{reduce} : \forall \alpha. (((\alpha * \alpha) \rightarrow \alpha) * \alpha * [\alpha]) \rightarrow \alpha$.

Type rule for $\text{reduce}(f, e, x)$:

Type Rules

Specify the type constraints and a way to derive the type of an expression, based on the types of its constituent subexpressions.

map : $\forall \alpha. \forall \beta. ((\alpha \rightarrow \beta) * [\alpha]) \rightarrow [\beta]$. **Type rule for map(f, x):**

- compute t , the type of (arbitrary expression) x , and check that $t \equiv [t_{el}]$ for some t_{el} .
- get f 's signature from `f.table`. IF f does not receive exactly one arg **THEN error()** **ELSE** $f : t_{in} \rightarrow t_{out}$, for some t_{in} and t_{out} .
- IF $(t_{el} \equiv t_{in})$ **THEN** `map(f, x) : [tout]`, **ELSE error()**

reduce : $\forall \alpha. (((\alpha * \alpha) \rightarrow \alpha) * \alpha * [\alpha]) \rightarrow \alpha$.

Type rule for reduce(f, e, x):

- compute t , the type of e , and t_x the type of x , and check that:
 - 1 $f : (t * t \rightarrow t)$, i.e., f is an operator that can reduce, and that
 - 2 $t_x = [t]$, i.e., x is an array of element type t .
- if so then `reduce(f, e, x) : t`.

- 1 Interpretation Recap: Synthesized/Inherited Attributes
- 2 Type-System Characterization
- 3 Type Checker for FASTO Without Arrays (Generic Notation)**
- 4 Advanced Concepts: Type Inference
- 5 Type Checker for FASTO With Arrays (F# Code)

What Is The Plan

The type checker builds (statically) unique types for each expression, and reports whenever a type rule is violated.

As before, we logically split the `ABSYN` representation into different *syntactic categories*: expressions, function decl, etc.,

and implement each syntactic category via one or several functions that use case analysis on the `ABSYN`-type constructors.

In practice we work on `ABSYN`, but here we keep implementation generic by using a notation that resembles the language grammar.

For symbols representing variable names, we use `name(id)` to get the name as a string. A type error is signaled via function `error()`.

Symbol Tables Used by the Type Checker

vtable binds variable names to their *types*,
e.g., `int`, `char`, `bool` or arrays, e.g., `[[[int]]]`.

ftable binds function names to their *types*. The type of a function is written $(t_1, \dots, t_n) \rightarrow t_0$, where t_1, \dots, t_n are the argument types and t_0 is the result type.

Type Checking an Expression (Part 1)

Inherited attributes: *vtable* and *fable*.

Synthesized attribute: the expression's type.

| $Check_{Exp}(Exp, vtable, ftable) = \text{case } Exp \text{ of}$ | |
|--|---|
| num | int |
| id | $t = \text{lookup}(vtable, \text{name}(\text{id}))$ if ($t == \text{unbound}$) then error(); int else t |
| $Exp_1 + Exp_2$ | $t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ if ($t_1 == \text{int}$ and $t_2 == \text{int}$) then int else error(); int |
| $Exp_1 = Exp_2$ | $t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ if ($t_1 == t_2$) then bool else error(); bool |
| ... | |

Note: In Fasto equality of arrays is not supported!

Type Checking a Function (Declaration)

- creates a *vtable* that binds the formal args to their types,
- computes the type of the function-body expression, named t_1 ,
- and checks that the function's return type equals t_1 .

| | |
|--|---|
| $Check_{Fun}(Fun, ftable) = \text{case } Fun \text{ of}$ | |
| $Type \text{ id } (Typelds) = Exp$ | $vtable = Check_{Typelds}(Typelds)$ $t_1 = Check_{Exp}(Exp, vtable, ftable)$ $if (t_1 \neq Type)$ $then \text{ error}(); \text{int}$ |

| | |
|--|---|
| $Check_{Typelds}(Typelds) = \text{case } Typelds \text{ of}$ | |
| $Type \text{ id}$ | $bind(SymTab.empty(), \text{id}, Type)$ |
| $Type \text{ id } , Typelds$ | $vtable = Check_{Typelds}(Typelds)$ $if (lookup(vtable, \text{id}) = unbound)$ $then bind(vtable, \text{id}, Type)$ $else \text{ error}(); vtable$ |

Type Checking the Whole Program

- builds the functions' symbol table,
- type-checks all functions,
- checks that a `main` function of no args exists.

| | |
|--|---|
| $Check_{Program}(Program) = \text{case } Program \text{ of}$ | |
| $Funs$ | $f_{table} = Get_{Funs}(Funs)$ $Check_{Funs}(Funs, f_{table})$ $\text{if } (lookup(f_{table}, main) \neq () \rightarrow \alpha)$ $\text{then } \mathbf{error}()$ |

| | |
|--|--|
| $Check_{Funs}(Funs, f_{table}) = \text{case } Funs \text{ of}$ | |
| Fun | $Check_{Fun}(Fun, f_{table})$ |
| $Fun Funs$ | $Check_{Fun}(Fun, f_{table})$ $Check_{Funs}(Funs, f_{table})$ |

Building the Functions' Symbol Table

| | |
|---|---|
| $Get_{Funs}(Funs) = \text{case } Funs \text{ of}$ | |
| Fun | $(f, t) = Get_{Fun}(Fun)$ $bind(SymTab.empty(), f, t)$ |
| $Fun Funs$ | $fTable = Get_{Funs}(Funs)$ $(f, t) = Get_{Fun}(Fun)$ $if (lookup(fTable, f) == unbound)$ $then bind(fTable, f, t)$ $else \text{error}(); fTable$ |

| | |
|--|---|
| $Get_{Fun}(Fun) = \text{case } Fun \text{ of}$ | |
| $Type \text{ id } (TypeIds) = Exp$ | $[t_1, \dots, t_n] = Get_{Types}(TypeIds)$ $(\text{id}, (t_1, \dots, t_n) \rightarrow Type)$ |

| | |
|--|--------------------------------|
| $Get_{Types}(TypeIds) = \text{case } TypeIds \text{ of}$ | |
| $Type \text{ id}$ | $[Type]$ |
| $Type \text{ id}, TypeIds$ | $Type :: Get_{Types}(TypeIds)$ |

- 1 Interpretation Recap: Synthesized/Inherited Attributes
- 2 Type-System Characterization
- 3 Type Checker for FASTO Without Arrays (Generic Notation)
- 4 Advanced Concepts: Type Inference**
- 5 Type Checker for FASTO With Arrays (F# Code)

Advanced Type Checking

- Data-Structures:** Represent the data-structure type in the symbol table and check operations on the values of this type.
- Overloading:** Check all possible types. If multiple matches, select a default typing or report errors.
- Type Conversion:** if an operator takes arguments of wrong types then, if possible, convert to values of the right type.
- Polymorphic/Generic Types:** Check whether a polymorphic function is correct for all instances of type parameters.
Instantiate the type parameters of a polymorphic function, which gives a monomorphic type.
- Type Inference:** Refine the type of a variable/function according to how it is used. If not used consistently then report error.

Polymorphic Functions: By Checking All Instances

In FASTO we have a fixed set of polymorphic functions of known types (signatures), e.g., `map`, `reduce`, etc., and the approach is to check individually each call, i.e., `map(f, exp)`!

Note that the type of `map` is not expressible in FASTO.

$$\text{map} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta],$$

$$\text{map}(f, [x_1, \dots, x_n]) \equiv [f(x_1), \dots, f(x_n)]$$

Type rule for `map`:

- compute t , the type of (arbitrary expression) x , and check that $t \equiv [t_{el}]$ for some t_{el} .
- get f 's signature from `f`table. IF f does not receive exactly one arg THEN `error()` ELSE $f : t_{in} \rightarrow t_{out}$, for some t_{in} and t_{out} .
- IF $(t_{el} \equiv t_{in})$ THEN $\text{map}(f, x) : [t_{out}]$,
- ELSE `error()`

Type Checking Map With Book Notations:

| $Check_{Exp}(Exp, vtable, ftable) = \text{case } Exp \text{ of}$ | |
|--|--|
| ... | |
| $\text{map}(\text{id}, Exp_{arr})$ | $t_{arr} = Check_{Exp}(Exp_{arr}, vtable, ftable)$ $t_{el} = \text{case } t_{arr} \text{ of}$ <ul style="list-style-type: none"> $Array(t_1) \rightarrow t_1$ $\text{ other} \rightarrow \text{error}()$ $t_f = \text{lookup}(ftable, \text{name}(\text{id}))$ $\text{case } t_f \text{ of}$ <ul style="list-style-type: none"> $\text{unbound} \Rightarrow \text{error}()$ $(t_{in} \rightarrow t_{out}) \Rightarrow$ <ul style="list-style-type: none"> $\text{if } t_{in} = t_{el} \text{ then } Array(t_{out})$ $\text{else } \text{error}()$ $\text{ otherwise} \Rightarrow \text{error}()$ |

Remember:

- $vtable$ maps variable names to their types
- $ftable$ maps function names to their (type) signature

Type Inference for Polymorphic Functions

Key difference: type rules check whether types can be “unified”, rather than type equality.

```
if ... then ([], [1,2,3], [])  
         else (['a','b'], [], [])
```

When we do not know a type we use a (fresh) **type variable**:

Type Inference for Polymorphic Functions

Key difference: type rules check whether types can be “unified”, rather than type equality.

```
if ... then ([], [1,2,3], [])
      else (['a','b'], [], [])
```

When we do not know a type we use a (fresh) **type variable**:

then: $\forall \alpha. \forall \beta. \text{list}(\alpha) * \text{list}(\text{int}) * \text{list}(\beta)$

else: $\forall \gamma. \forall \delta. \text{list}(\text{char}) * \text{list}(\gamma) * \text{list}(\delta)$

notation: use Greeks for type vars, omit \forall but use fresh names.

Types t_1 and t_2 can be unified $\Leftrightarrow \exists$ substitution $S \mid S(t_1) = S(t_2)$.

Most-General Unifier: the least specialized subs/type that still unifies

Type Inference for Polymorphic Functions

Key difference: type rules check whether types can be “unified”, rather than type equality.

```
if ... then ([], [1,2,3], [])
      else (['a','b'], [], [])
```

When we do not know a type we use a (fresh) **type variable**:

then: $\forall \alpha. \forall \beta. \text{list}(\alpha) * \text{list}(\text{int}) * \text{list}(\beta)$

else: $\forall \gamma. \forall \delta. \text{list}(\text{char}) * \text{list}(\gamma) * \text{list}(\delta)$

notation: use Greeks for type vars, omit \forall but use fresh names.

Types t_1 and t_2 can be unified $\Leftrightarrow \exists$ substitution $S \mid S(t_1) = S(t_2)$.

Most-General Unifier: the least specialized subs/type that still unifies
 $S = \{\alpha \leftarrow \text{char}, \gamma \leftarrow \text{int}, \delta \leftarrow \beta\} \Rightarrow \text{list}(\text{char}) * \text{list}(\text{int}) * \text{list}(\beta)$

Example: Inferring the Type of SML's length

```
fun length(x) = if null(x) then 0
                else length( tl(x) ) + 1
```

Example: Inferring the Type of SML's length

```
fun length(x) = if null(x) then 0
                else length( tl(x) ) + 1
```

| <i>EXPRESSION</i> | : <i>TYPE</i> | <i>UNIFY</i> |
|------------------------|---|--|
| length | : $\beta \rightarrow \gamma$ | |
| x | : β | |
| if | : $bool * \alpha_i * \alpha_j \rightarrow \alpha_i$ | |
| null | : $list(\alpha_n) \rightarrow bool$ | |
| null(x) | : $bool$ | $list(\alpha_n) \equiv \beta$ |
| 0 | : int | $\alpha_i \equiv int$ |
| + | : $int * int \rightarrow int$ | |
| tl | : $list(\alpha_t) \rightarrow list(\alpha_t)$ | |
| tl(x) | : $list(\alpha_t)$ | $list(\alpha_t) \equiv list(\alpha_n)$ |
| length(tl(x)) | : γ | |
| length(tl(x)) + 1 | : int | $\gamma \equiv int$ |
| if(..) then .. else .. | : int | |

Most-General Unifier Algorithm (MGU)

- a type expression is represented by a graph (typically acyclic),
- a set of unified nodes has one representative, REP, (initially each node is its own representative),
- `find(n)` returns the representative of node `n`.
- `union(m,n)` merges the equivalence classes of `m` and `n`:
 - if `m` is a type constructor (or basic type) then REP of all nodes in `n`'s equivalence class are set to `find(m)` (and similar for `n`),
 - otherwise pick one, e.g., `n`, and set the REP of all the nodes in `m`'s equivalence class to `find(n)`

boolean unify(Node m, Node n)

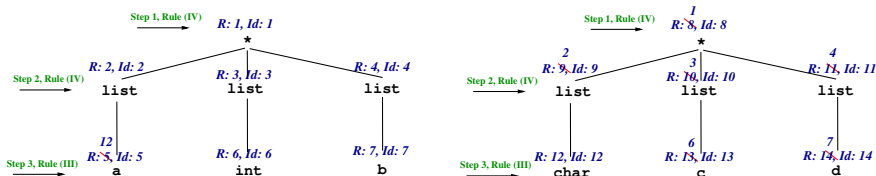
- ```

(I) if (find(m) = find(n)) then return true;
(II) else if (m and n are the same basic type) then return true;
(III) else if (m or n represent a type variable) then union(m, n); return true;
(IV) else if (m and n are the same type constructor
 with children m1, .., mk and n1, .., nk, $\forall k$) then
 union(m, n); return unify(m1, n1) and .. and unify(mk, nk);
(V) else return false;

```

# Most-General Unifier Example

Each node is annotated with two integer values:  
 – REP (R)  
 – node's identifier (Id)  
 Initially, Id = R, i.e., every node in its own equiv class.



**SUCCESS** (after three big horizontal steps), MGU is: `list(char) * list(int) * list(b)`

To construct the unified type (after MGU succeeds): start with any of the two type expressions, and write down the “representative” nodes, i.e., **the ones with Node Id = REP** (otherwise jump to the corresponding REP node and write it down).

## Structural-Equivalence Example

Intuitively, the names of the structs and fields, e.g., A, a, do NOT matter, but only the type constructors, e.g., struct, \* and basic types, e.g., int.

Under Structural Equivalence: types A, B and C Are Equivalent

```
struct A { struct B { struct C {
 int a; int b; int d;
 struct B* b; struct A* a; struct C* c;
}; }; };
```

Next slides compute the most-general unifier (MGU) of A and C, which both have **cyclic** graph representations of their types.

## Structural-Equivalence Example

Intuitively, the names of the structs and fields, e.g., A, a, do NOT matter, but only the type constructors, e.g., struct, \* and basic types, e.g., int.

Under Structural Equivalence: types A, B and C Are Equivalent

```

struct A {
 int a;
 struct B* b;
};

struct B {
 int b;
 struct A* a;
};

struct C {
 int d;
 struct C* c;
};

```

Next slides compute the most-general unifier (MGU) of A and C, which both have **cyclic** graph representations of their types.

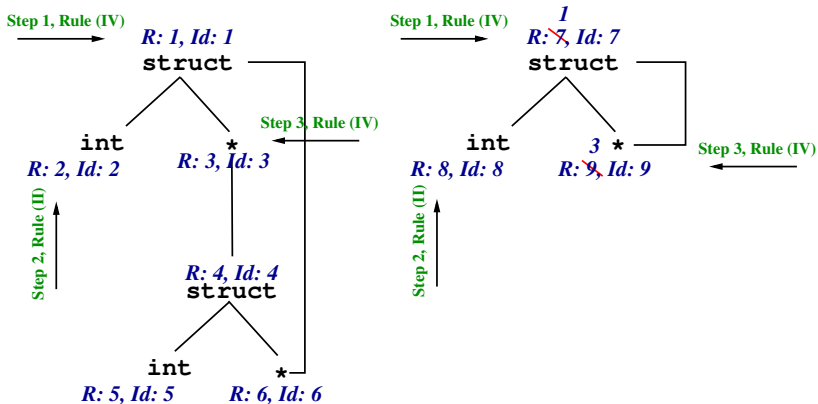
To construct the unified type (after MGU succeeds): start with any of the two type expressions, and write down the “representative” nodes, i.e., **the ones with Node Id = REP** (otherwise go to the corresponding REP node and write it down).

For **cyclic graphs**, a marking phase is necessary so that you do not visit the same node multiple times, i.e., infinite recursion.

## Structural Equivalence Example (2)

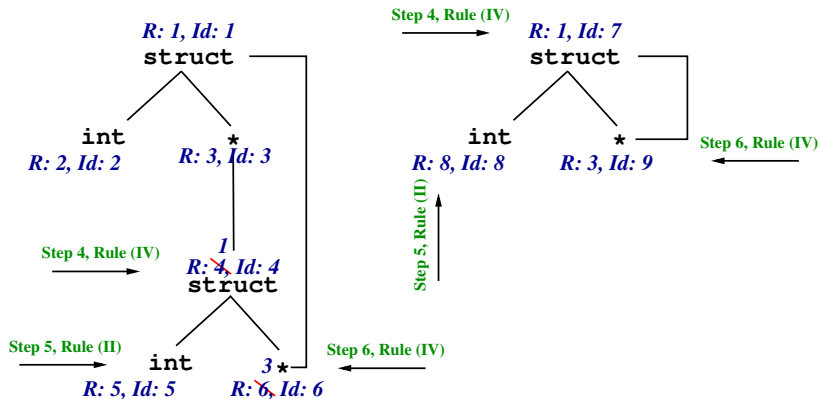
Each node is annotated  
with two integer values:

- REP
- node's identifier



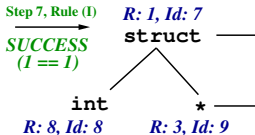
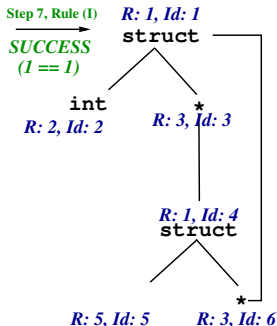
## Structural Equivalence Example (3)

Each node is annotated with two integer values: – REP  
– node's identifier



## Structural Equivalence Example (3)

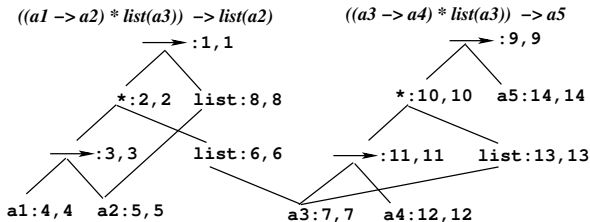
Each node is annotated  
with two integer values:      - REP  
                                                 - node's identifier



After MGU succeeds, to build the unified type when graph may be cyclic, a marking phase is necessary so that you do not visit the same node multiple times, i.e., infinite recursion.

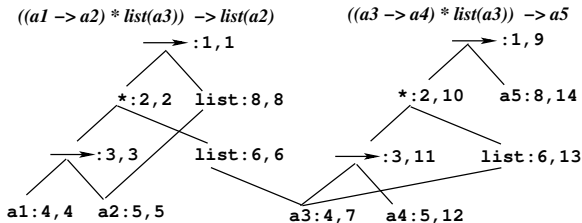
The unified type would be the structural type of struct C.

# Another Most-General Unifier Example



Each node is annotated with two integer values:

- REP
- node's identifier



The unifier is constructed by combining nodes' REPs:

$((a1 \rightarrow a2) * list(a2)) \rightarrow list(a2)$

To construct the unified type: start with any of the two type expressions; and write down the “representative” nodes, i.e., the ones with node id = REP (otherwise go to the REP node & write it).



- 1 Interpretation Recap: Synthesized/Inherited Attributes
- 2 Type-System Characterization
- 3 Type Checker for FASTO Without Arrays (Generic Notation)
- 4 Advanced Concepts: Type Inference
- 5 Type Checker for FASTO With Arrays (F# Code)

# What Changes When Adding Arrays? (part 1)

Polymorphic Array Constructors and Combinators:

$$\text{map: } \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta],$$
$$\text{map}(f, \{x_1, \dots, x_n\}) \equiv \{f(x_1), \dots, f(x_n)\}$$

$$\text{reduce: } \forall \alpha. ((\alpha * \alpha) \rightarrow \alpha) * \alpha * [\alpha] \rightarrow \alpha$$
$$\text{reduce}(g, e, \{x_1, \dots, x_n\}) \equiv g(..(g(e, x_1).., x_n)$$

Question 1: Do we need to implement type inference?

# What Changes When Adding Arrays? (part 1)

Polymorphic Array Constructors and Combinators:

$$\text{map: } \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta],$$

$$\text{map}(f, \{x_1, \dots, x_n\}) \equiv \{f(x_1), \dots, f(x_n)\}$$

$$\text{reduce: } \forall \alpha. ((\alpha * \alpha) \rightarrow \alpha) * \alpha * [\alpha] \rightarrow \alpha$$

$$\text{reduce}(g, e, \{x_1, \dots, x_n\}) \equiv g(..(g(e, x_1).., x_n)$$

Question 1: Do we need to implement type inference?

Answer 1: No! FASTO supports a fixed set of polymorphic function whose types are known (or if you like, very simple type inference).

# What Changes When Adding Arrays? (part 1)

Polymorphic Array Constructors and Combinators:

$$\text{map: } \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta],$$

$$\text{map}(f, \{x_1, \dots, x_n\}) \equiv \{f(x_1), \dots, f(x_n)\}$$

$$\text{reduce: } \forall \alpha. ((\alpha * \alpha) \rightarrow \alpha) * \alpha * [\alpha] \rightarrow \alpha$$

$$\text{reduce}(g, e, \{x_1, \dots, x_n\}) \equiv g(..(g(e, x_1).., x_n)$$

Question 2: Assuming type-checking is successful, can we forget the type of  $\text{map}(f, a)$ ?

# What Changes When Adding Arrays? (part 1)

Polymorphic Array Constructors and Combinators:

$$\text{map: } \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta],$$

$$\text{map}(f, \{x_1, \dots, x_n\}) \equiv \{f(x_1), \dots, f(x_n)\}$$

$$\text{reduce: } \forall \alpha. ((\alpha * \alpha) \rightarrow \alpha) * \alpha * [\alpha] \rightarrow \alpha$$

$$\text{reduce}(g, e, \{x_1, \dots, x_n\}) \equiv g(..(g(e, x_1).., x_n)$$

Question 2: Assuming type-checking is successful, can we forget the type of  $\text{map}(f, a)$ ?

Answer 2: No, the type of  $[t] \equiv \text{map}(f, a)$  needs to be remembered for machine-code generation, e.g.,  $t : \text{int}$  vs  $t : \text{char}$ .

Same for array literals, array indexing, map, reduce, etc.

## What Changes When Adding Arrays? (part 2)

$\text{map} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta]$ . Type rule for  $\text{map}(f, x)$ :

- compute  $t$ , the type of  $x$ , and check that  $t \equiv [t_{in}]$  for some  $t_{in}$ .
- check that  $f : t_{in} \rightarrow t_{out}$
- if so then  $\text{map}(f, x) : [t_{out}]$ .

ABSYN representation for  $\text{map}$ :  $\text{Exp}\langle 'T \rangle = \dots \mid$

- Map of  $\text{FunArg}\langle 'T \rangle * \text{Exp}\langle 'T \rangle * 'T * 'T * \text{Position}$
- **Before type checking, 'Ts are unknown:**  $\text{UntypedExp} = \text{Exp}\langle \text{unit} \rangle$
- **After type checking types are known:**  $\text{TypedExp} = \text{Exp}\langle \text{Type} \rangle$

1st  $T$  is the input-array element type, e.g.,  $t_{in}$ ,

2nd  $T$  is the output-array element type, e.g.,  $t_{out}$ .

## What Changes When Adding Arrays? (part 2)

$\text{map} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta]$ . Type rule for  $\text{map}(f, x)$ :

- compute  $t$ , the type of  $x$ , and check that  $t \equiv [t_{in}]$  for some  $t_{in}$ .
- check that  $f : t_{in} \rightarrow t_{out}$
- if so then  $\text{map}(f, x) : [t_{out}]$ .

ABSYN representation for  $\text{map}$ :  $\text{Exp}\langle 'T \rangle = \dots \mid$

- Map of  $\text{FunArg}\langle 'T \rangle * \text{Exp}\langle 'T \rangle * 'T * 'T * \text{Position}$
- **Before type checking, 'Ts are unknown:**  $\text{UntypedExp} = \text{Exp}\langle \text{unit} \rangle$
- **After type checking types are known:**  $\text{TypedExp} = \text{Exp}\langle \text{Type} \rangle$

1st  $T$  is the input-array element type, e.g.,  $t_{in}$ ,

2nd  $T$  is the output-array element type, e.g.,  $t_{out}$ .

$\text{checkProg} : \text{UntypedProg} \rightarrow \text{TypedProg}$

Type checking an expression/program now results in a new  $\text{exp}/\text{prg}$ , where all  $'T$  fields, initially unknown ( $'T = \text{unit}$ ) of an expression are filled with known types ( $'T = \text{Type}$ ).

## TypeChecker.fs: Entry-Point checkProg

```
(* function symbol table *)
type FunTable = SymTab.SymTab<(Type * Type list * Position)>

(*adds a (fun name,signature) to ftab*)
fun updateFunctionTable (ftab: FunTable) (fundec: UntypedFunDec) : FunTable =
 let (FunDec (fname, ret_type, args, _, pos)) = fundec
 let arg_types = List.map (fun (Param (_, ty)) -> ty) args
 match SymTab.lookup fname ftab with
 | Some _ -> raise (MyError ("Duplicate function!", pos))
 | None -> SymTab.bind fname (ret_type, arg_types, pos) ftab

let checkProg (funDecs : UntypedProg) : TypedProg =
 (* builds ftab from special funs and pgm functions *)
 let ftab0 = SymTab.fromList [("chr", (Char, [Int], (0,0)));
 ("ord", (Int, [Char], (0,0))]]
 ftab = List.fold updateFunctionTable ftab0 funDecs

 (* applies typing rules and fills in 'T with Types *)
 decorated_funDecs = List.map (checkFun ftab) funDecs

match SymTab.lookup "main" ftab with
| Some (_, [], _) -> decorated_funDecs (* all fine! *)
| _ -> raise (MyError("No Main, OR Main with Args", p))
```



# TypeChecker.fs: Type Checking a Function

```

type VarTable = SymTab.SymTab<Type> (* variable symbol table *)

checkFunWithVtable (vtab: VarTable) (ftab: FunTable) pos
 (fundec: UntypedFunDec) : TypedFunDec =
 let (FunDec (fname, rettype, params, body, fpos)) = fundec
 ... (* Expand vtab by adding the formal param bindings. *)
 paramtable = List.fold addParam (SymTab.empty()) params
 vtab' = SymTab.combine paramtable vtab

 (*type check fun's body \Rightarrow the type of and a type-annotated body'*)
 (body_type, body') = checkExp ftab vtab' body

 (* If return type matches body' type \Rightarrow type-annotated fun declaration *)
 if body_type = rettype
 then (FunDec (fname, rettype, params, body', pos))
 else raise (MyError ("Fun return type does NOT matches body type",fpos))

```

Isn't vtab always empty? Why pass it as param?

# TypeChecker.fs: Type Checking a Function

```

type VarTable = SymTab.SymTab<Type> (* variable symbol table *)

checkFunWithVtable (vtab: VarTable) (ftab: FunTable) pos
 (fundec: UntypedFunDec) : TypedFunDec =
 let (FunDec (fname, rettype, params, body, fpos)) = fundec
 ... (* Expand vtab by adding the formal param bindings. *)
 paramtable = List.fold addParam (SymTab.empty()) params
 vtab' = SymTab.combine paramtable vtab

 (*type check fun's body \Rightarrow the type of and a type-annotated body'*)
 (body_type, body') = checkExp ftab vtab' body

 (* If return type matches body' type \Rightarrow type-annotated fun declaration *)
 if body_type = rettype
 then (FunDec (fname, rettype, params, body', pos))
 else raise (MyError ("Fun return type does NOT matches body type",fpos))

```

Isn't vtab always empty? Why pass it as param?

Because we need to typecheck both named and anonymous (lambda) function declarations. (see checkFunArg, the `Lambda` case).

# TypeChecker.fs: Type Checking Simple Exprs

```
(* computes the type of and the type-annotated expression *)
checkExp (ftab : FunTable)
 (vtab : VarTable)
 (exp : UntypedExp) : (Type * TypedExp) =
 match exp with ...
 | Var (s, pos) -> match SymTab.lookup s vtab with
 | None -> raise (MyError ("Unknown var!"), pos))
 | Some t -> (t, Var (s, pos)))

(* e1, e2 must be of the same SCALAR type. The result type is Bool. *)
| Equal (e1, e2, pos) ->
 let (t1, e1') = checkExp ftab vtab e1
 (t2, e2') = checkExp ftab vtab e2
 match (t1 = t2, t1) with
 | (false, _) -> raise (MyError ("Equal different types!", pos))
 | (true, Array _) -> raise (MyError ("Cannot compare arrays", pos))
 | _ -> (Bool, Equal (e1', e2', pos))
```