Type Checking

Slides by Cosmin Oancea
cosmin.oancea@diku.dk

(with tweaks by Robert Glück, glueck@di.ku.dk)

Department of Computer Science (DIKU)
University of Copenhagen

May 2023, IPS Lecture Slides

Structure of a Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree

↓

Type checking

↓

Syntax tree

↓

Intermediate code generation ⟶ Intermediate code

↑

Machine code generation

↑

Symbolic machine code

↑

Register allocation

↑

Ditto with named registers

↑

Assembly and linking

↑

Binary machine 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)

Interpretation Recap

Interpreter *vs.* **Compiler**

Interpreter	Compiler	Target Program
program, *input* $\downarrow$	*program* $\downarrow$	*input* $\downarrow$
Interpreter	Compiler	Target Program
$\downarrow$ *output*	$\downarrow$ *target program*	$\downarrow$ *output*

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 the interpreter's implementation language.

An interpreter performs a *1-stage* computation. A compiler and the generated target program perform the same computation in *2 stages*.

Synthesized vs. Inherited Attributes

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

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

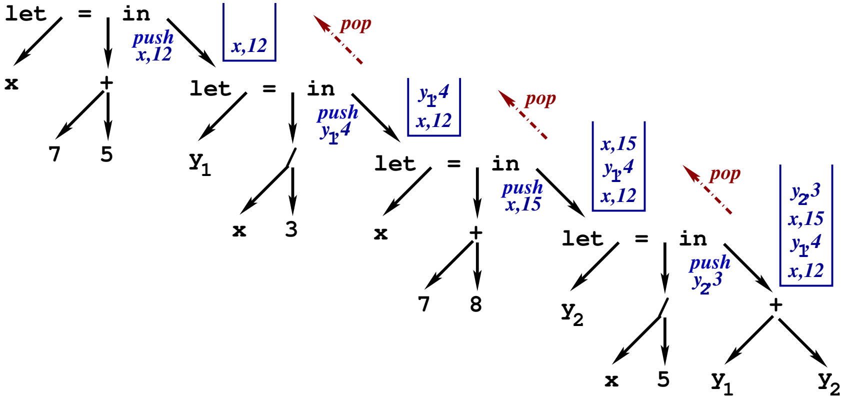
Synthesized: info passed upwards in the AbsSyn traversal, i.e., from leaves to the root. Think: the result. Example?

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 in Interpreter

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

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



Example of Synthesized Attributes in Interpreter

The interpreted value of an expression is *synthesized*:

```
let x = 7 + 5 ... in y1 + y2
```

Example of both *synthesized and inherited* attributes:

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

and used in the interpretation of an expression. They are synthesized by a declaration and inherited by the scope of the declaration.

Interpretation vs. Compilation Pros and Cons

Interpretation vs. Compilation Pros and Cons

+ Simple (good for impatient people, good for prototyping).

+ 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:

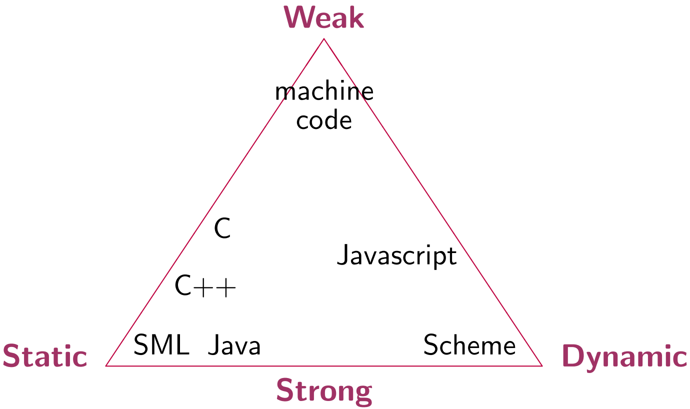
- `+`, `-` *expect integer arguments:*
  `a + (b == c)`
- `if`-*branch expressions have the same type:*
  `let a = (if (b == 3) then 'b' else 11) in ...`
- *type and number of formal and actual arguments match:*
  ```
  fun int sum ([int] x) = reduce(op +, 0, x)
  fun [bool] main()     = map(sum, iota(4))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

`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 `ftable`. `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)` $:\ [t_{out}]$ `ELSE error()`

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 `ftable`. `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)` : $[t_{out}]$ `ELSE error()`

`reduce` : $\forall\,\alpha.\,(((\alpha\ *\ \alpha)\ \rightarrow\ \alpha)\ *\ \alpha\ *\ [\alpha])\ \rightarrow\ \alpha$.
Type rule for `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` $: \forall\, \alpha.\, \forall\, \beta.\, ((\alpha \to \beta) \; * \; [\alpha]) \; \to \; [\beta]$. 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 `ftable`. `IF` f does not receive exactly one arg `THEN error() ELSE` $f \; : \; t_{in} \to t_{out}$, for some t_{in} and t_{out}.
- `IF` $(t_{el} \equiv t_{in})$ `THEN map(f, x)` $:$ $[t_{out}]$ `ELSE error()`

`reduce` $: \forall\, \alpha.\, (((\alpha \; * \; \alpha) \; \to \; \alpha) \; * \; \alpha \; * \; [\alpha]) \; \to \; \alpha.$

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

- compute the type t of e, the type t_x of `x`, and check that:
  1. $f \; : \; (t \; * \; t \; \to t)$, i.e., `f` is an operator that can reduce, and
  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 constructors.

In practice, we work on AbSyn, but here we keep the 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 $(t_1, ..., t_n) \rightarrow t_0$, where t_1, ..., t_n are the argument types and t_0 is the result type.

Type Checking an Expression (Part 1)

Inherited attributes: *vtable* and *ftable*.

Synthesized attribute: the expression's type.

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

Note: In Fasto equality of arrays is not supported!

Type Checking an Expression (Part 2)

$Check_{Exp}(Exp, vtable, ftable)$ = `case` Exp `of`	
...	
`if` Exp_1 `then` Exp_2 `else` Exp_3	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $t_3 = Check_{Exp}(Exp_3, vtable, ftable)$ *if* ($t_1 ==$ `bool` *and* $t_2 == t_3$) *then* t_2 *else* **error**(); t_2
`let` **id** = Exp_1 `in` Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, name(\mathbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$
id ($Exps$)	$t = lookup(ftable, name(\mathbf{id}))$ *if* ($t == unbound$) *then* **error**(); `int` *else* $((t_1, \ldots, t_n) \rightarrow t_0) = t$ $[t'_1, \ldots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ *if* ($m == n$ *and* $t_1 == t'_1, \ldots, t_n == t'_n$) *then* t_0 *else* **error**(); t_0

Type Checking a Function Declaration

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

$Check_{Fun}(Fun, ftable) =$ case Fun of	
$Type$ **id** ($TypeIds$) $=$ Exp	$vtable = Check_{TypeIds}(TypeIds)$ $t_1 = Check_{Exp}(Exp, vtable, ftable)$ $if\ (\ t_1 \neq Type\ )$ $then$ **error()**

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

Type Checking the Whole Program

1. builds the complete functions' symbol table,
2. type-checks each function,
3. checks that a `main` function of no args exists.

$Check_{Program}(Program) = \texttt{case}\ Program\ \texttt{of}$	
$Funs$	$ftable = Get_{Funs}(Funs)$ "1st pass" $Check_{Funs}(Funs, ftable)$ "2nd pass" $if\ (\ lookup(ftable, \texttt{main}) \neq (\ ) \to \alpha\ )$ $then$ **error()**

$Check_{Funs}(Funs, ftable) = \texttt{case}\ Funs\ \texttt{of}$	
Fun	$Check_{Fun}(Fun, ftable)$
$Fun\ Funs$	$Check_{Fun}(Fun, ftable)$ $Check_{Funs}(Funs, ftable)$

Type Checking the Whole Program

1. builds the complete functions' symbol table,
2. type-checks each function,
3. checks that a `main` function of no args exists.

$Check_{Program}(Program) = \texttt{case}\ Program\ \texttt{of}$	
Funs	$ftable = Get_{Funs}(Funs)$ "1st pass" $Check_{Funs}(Funs, ftable)$ "2nd pass" $if\ (\ lookup(ftable, \texttt{main}) \neq (\ ) \to \alpha\ )$ $then$ **error()**

$Check_{Funs}(Funs, ftable) = \texttt{case}\ Funs\ \texttt{of}$	
Fun	$Check_{Fun}(Fun, ftable)$
Fun Funs	$Check_{Fun}(Fun, ftable)$ $Check_{Funs}(Funs, ftable)$

Q: Why is the complete ftable built *before* checking the functions?

Type Checking the Whole Program

1. builds the complete functions' symbol table,
2. type-checks each function,
3. checks that a `main` function of no args exists.

$Check_{Program}(Program) = \texttt{case}\ Program\ \texttt{of}$	
$Funs$	$ftable = Get_{Funs}(Funs)$ "1st pass" $Check_{Funs}(Funs, ftable)$ "2nd pass" $if\ (\ lookup(ftable, \texttt{main}) \neq (\ ) \to \alpha\ )$ $then$ **error()**

$Check_{Funs}(Funs, ftable) = \texttt{case}\ Funs\ \texttt{of}$	
Fun	$Check_{Fun}(Fun, ftable)$
$Fun\ Funs$	$Check_{Fun}(Fun, ftable)$ $Check_{Funs}(Funs, ftable)$

Q: Why is the complete ftable built *before* checking the functions?

A: To type check mutually recursive functions.

Building the Functions' Symbol Table

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

$Get_{Fun}(Fun) =$ case Fun of	
$Type$ **id** ($TypeIds$) $=$ Exp	$[t_1,\ldots,t_n] = Get_{Types}(TypeIds)$ (**id**, $(t_1,\ldots,t_n) \rightarrow Type$)

$Get_{Types}(TypeIds) =$ case $TypeIds$ of	
$Type$ **id**	$[Type]$
$Type$ **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 error.

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`. The approach is to check individually each call, i.e., `map(f,exp)`.

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

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

Type rule for `map`:

- compute t, the type of (arbitrary expression) `exp`, and check that $t \equiv [t_{el}]$ for some t_{el}.
- get f's signature from `ftable`. `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,exp)` : $[t_{out}]$
  `ELSE error()`

Type Checking Map With Book Notations

$Check_{Exp}(Exp, vtable, ftable) = \texttt{case}\ Exp\ \texttt{of}$	
$\ldots$	
`map(` **id**, Exp_{arr}`)`	$t_{arr} = Check_{Exp}(Exp_{arr}, vtable, ftable)$ $t_{el} = case\ t_{arr}\ of$ $\quad Array(t_1) \Rightarrow t_1$ $\quad \mid\ otherwise \Rightarrow$ **error()** $t_f = lookup(ftable, name(\textbf{id}))$ $case\ t_f\ of$ $\quad unbound \Rightarrow$ **error()** $\quad \mid\ (t_{in} \rightarrow t_{out}) \Rightarrow$ $\qquad if\ t_{in} == t_{el}\ then\ Array(t_{out})$ $\qquad else$ **error()** $\quad \mid\ otherwise \Rightarrow$ **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-branch: $\forall\alpha.\forall\beta.\ \ list(\alpha) \ast list(int) \ast list(\beta)$

else-branch: $\forall\gamma.\forall\delta.\ \ list(char) \ast list(\gamma) \ast list(\delta)$

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 subst/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-branch: $\forall\alpha.\forall\beta.\ \mathit{list}(\alpha) * \mathit{list}(\mathit{int}) * \mathit{list}(\beta)$

else-branch: $\forall\gamma.\forall\delta.\ \mathit{list}(\mathit{char}) * \mathit{list}(\gamma) * \mathit{list}(\delta)$

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 subst/type that still unifies.

$S = \{\alpha \leftarrow \mathit{char}, \gamma \leftarrow \mathit{int}, \delta \leftarrow \beta\} \Rightarrow \mathit{list}(\mathit{char}) * \mathit{list}(\mathit{int}) * \mathit{list}(\beta)$

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
```

Expression	:	Type	Unify
length	:	$\beta \to \gamma$	
x	:	β	
if	:	$bool * \alpha_i * \alpha_i \to \alpha_i$	$\alpha_i \equiv \gamma$
null	:	$list(\alpha_n) \to bool$	
null(x)	:	$bool$	$list(\alpha_n) \equiv \beta$
0	:	int	$\alpha_i \equiv int$
$+$	:	$int * int \to int$	
tl	:	$list(\alpha_t) \to list(\alpha_t)$	
$tl(x)$	:	$list(\alpha_t)$	$list(\alpha_t) \equiv list(\alpha_n)$
$length(tl(x))$	:	γ_1	
$length(tl(x)) + 1$	:	int	$\gamma_1 \equiv int, \gamma \equiv int$
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 `find(m)` is a basic type or type constructor then set the REP of all nodes in `n`'s equivalence class to `find(m)` (similar for `n`),
  - otherwise, pick one, e.g., `n`, and set the REP of all nodes in `m`'s equivalence class to `find(n)`.

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 `find(m)` is a basic type or type constructor then set the REP of all nodes in `n`'s equivalence class to `find(m)` (similar for `n`),
  - otherwise, pick one, e.g., `n`, and set the REP of all nodes in `m`'s equivalence class to `find(n)`.

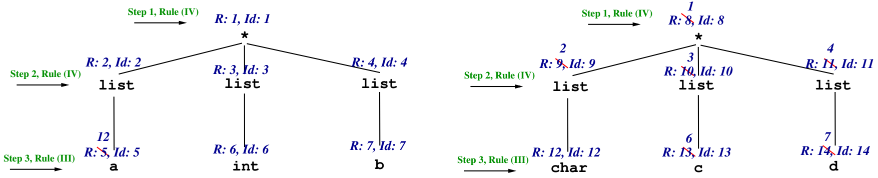
boolean unify(Node m, Node n): s = `find`(m); t = `find`(n)

(I) **if** ($s = t$) **then return** *true*;
(II) **else if** (*s and t are the same basic type*) **then return** *true*;
(III) **else if** (*s or t represent a type variable*) **then** `union`(s, t); **return** *true*;
(IV) **else if** (*s and t are the same type constructor with children $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$*) **then**
`union`(s, t); **return** `unify`(s_1, t_1) **and** … **and** `unify`(s_k, t_k);
(V) **else return** *false*;

Most-General Unifier Example

Each node is annotated with two integer values: – REP (R) – node's identifier (Id)

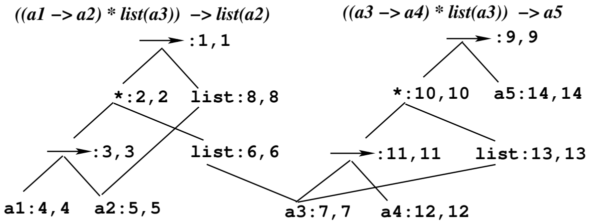
Initially, Id = R, i.e., every node in its own equiv class.



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

After MGU succeeds, construct the unified type: start with any of the two type expressions, and write down the "representative" nodes, i.e., the nodes with Id $=$ REP (otherwise go to the corresponding REP node and write it down).

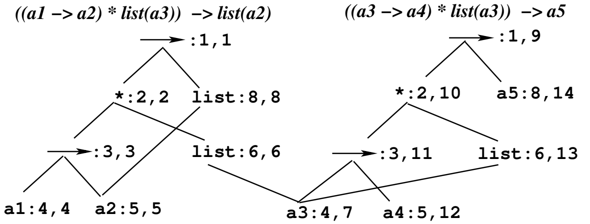
Another Most-General Unifier Example



Each node is annotated with two integer values:

– REP

– node's identifier



The unifier is constructed by combining nodes' REPs:

((a1 –> a2) * list(a1)) –> list(a2)

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

Advanced: 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.

Advanced: 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;
};                        };                                  };
```

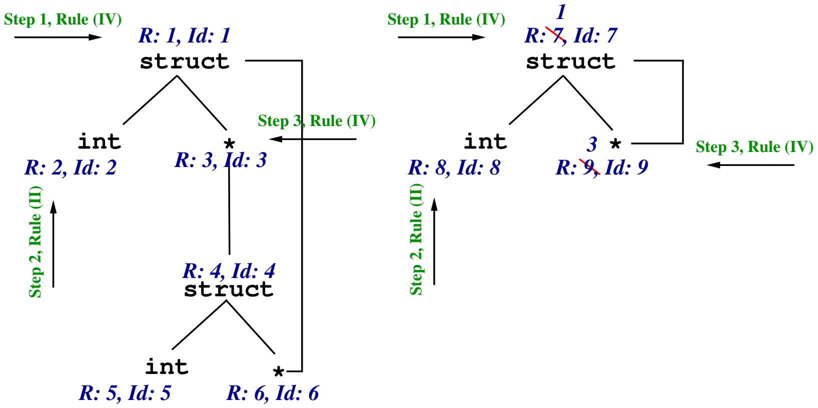
Next slides compute the most-general unifier (MGU) of `A` and `C`, which both have cyclic graph representations of their types.

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 nodes with 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.

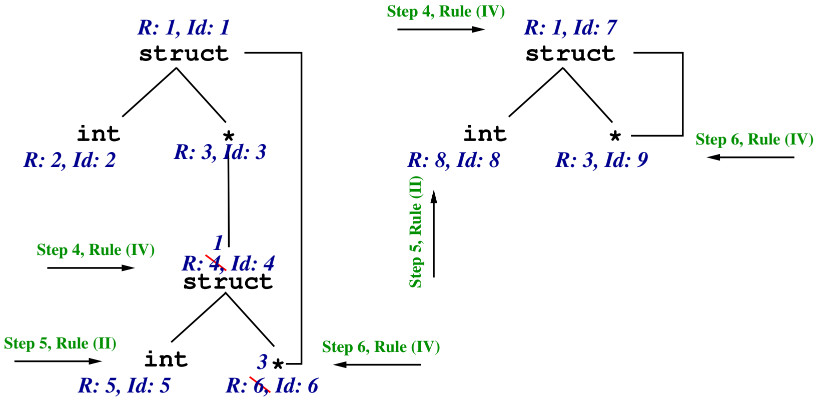
Advanced: Structural Equivalence Example (2)

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



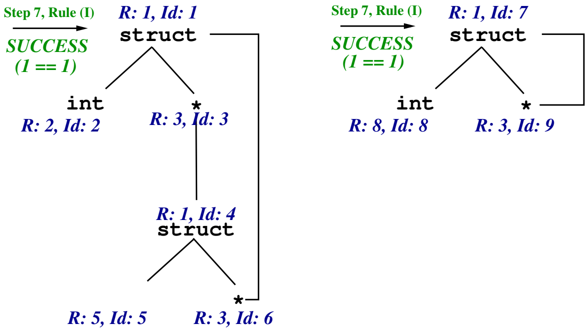
Advanced: Structural Equivalence Example (3)

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



Advanced: 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`.

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:

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

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

Question 1: Do we need to implement type inference in FASTO?

What Changes When Adding Arrays? (part 1)

Polymorphic Array Constructors and Combinators:

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

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

Question 1: Do we need to implement type inference in FASTO?

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:

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

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

Question 2: Assuming type-checking is successful, can we forget the type of `map(f,a)`?

What Changes When Adding Arrays? (part 1)

Polymorphic Array Constructors and Combinators:

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

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

Question 2: Assuming type-checking is successful, can we forget the type of `map(f,a)`?

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

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

What Changes When Adding Arrays? (part 2)

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

AbSyn representation for map: `Exp<'T> = ... |`

- `Map of FunArg<'T> * Exp<'T> * 'T * 'T * Position`
- Before type checking, all `'T` are unknown: `UntypedExp=Exp<unit>`
- After type checking, types are known: `TypedExp=Exp<Type>`

1st `T` is the input-array element type, i.e., t_{in},
2nd `T` is the output-array element type, i.e., t_{out}.

What Changes When Adding Arrays? (part 2)

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

ABSYN representation for map: `Exp<'T> = ... |`

- `Map of FunArg<'T> * Exp<'T> * 'T * 'T * Position`
- Before type checking, all `'T` are unknown: `UntypedExp=Exp<unit>`
- After type checking, types are known: `TypedExp=Exp<Type>`

1st `T` is the input-array element type, i.e., t_{in},
2nd `T` is the output-array element type, i.e., t_{out}.

`checkProg : UntypedProg -> TypedProg`

Type checking an expression/program now results in a new exp/prg, where all `'T` fields of an expression, initially unknown (`'T=unit`), are filled with known types (`'T=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    (* 1st pass *)

        (* applies typing rules and fills in 'T with Types *)
        decorated_funDecs = List.map (checkFun ftab) funDecs   (* 2nd pass *)

    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

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

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 ⇒ the type of and a type-annotated body' *)
        (body_type, body') = checkExp ftab vtab' body

    (* if return type matches body' type ⇒ 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

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

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 ⇒ the type of and a type-annotated body' *)
        (body_type, body') = checkExp ftab vtab' body

    (* if return type matches body' type ⇒ 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))
    ...
```