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

Program text
↓
Lexical analysis
↓
Symbol sequence
↓
Syntax analysis
↓
Syntax tree
↓
Type Checking
↓
Syntax tree
↓
Intermediate code generation

Binary machine code
↑
Assembly and linking
↑
Ditto with named registers
↑
Register allocation
↑
Symbolic machine code
↑
Machine code generation
↑
Intermediate 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)
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., \textit{vtable} and \textit{ftable}, in the interpretation of an expression:

\[ \text{Eval}_{\text{Exp}}(\text{Exp}, \text{vtable}, \text{ftable}) = \ldots \]
Example of Synthesized Attributes

The interpreted value of an expression / program is synthesized.

Example of both synthesized and inherited attributes:

\[
\begin{align*}
  vtable &= \text{Bind}_{\text{TypeIds}}(\text{TypeIds, args}) \\
  ftable &= \text{Build}_{\text{ftable}}(\text{Funs})
\end{align*}
\]

and used in the interpretation of an expression.
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.

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

---

Machine code

- C
- C++
- Javascript
- Scheme
- SML
- Java

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

\[
\begin{align*}
\text{compute } t, \text{ the type of } (\text{arbitrary expression}) \text{ } x, \\
\text{and check that } t \equiv [t]\text{ for some } tel.
\end{align*}
\]

\[
\begin{align*}
\text{get } f\text{'s signature from } \text{ftable}. \text{ IF } f \text{ does not receive exactly one arg THEN error() ELSE } f: t\text{ in } \rightarrow t\text{ out}, \text{ for some } t\text{ in } \text{ and } t\text{ out}. \text{ IF } (t\text{ el} \equiv t\text{ in}) \text{ THEN } \text{map}(f, x): [t\text{ out}], \text{ ELSE error()}
\end{align*}
\]

reduce : \( \forall \alpha. ((\alpha \ast \alpha \rightarrow \alpha) \ast \alpha \ast [\alpha]) \rightarrow \alpha. \) Type rule for \( \text{reduce}(f, e, x) \):

\[
\begin{align*}
\text{compute } t, \text{ the type of } e, \text{ and } t\text{x the type of } x, \text{ and check that: } \\
1 f: (t \ast t \rightarrow t), \text{ i.e., } f \text{ is an operator that can reduce, and that } \\
2 t\text{x} = [t], \text{ i.e., } x \text{ is an array of element type } t.
\end{align*}
\]

if so then \( \text{reduce}(f, e, x): t. \)
Type Rules

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

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

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

\[
\text{reduce} : \forall \alpha. (((\alpha \ast \alpha) \rightarrow \alpha) \ast \alpha \ast [\alpha]) \rightarrow \alpha. \text{ Type rule for reduce}(f, e, x):
\]
Type Rules

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

\[
\text{map} : \forall \alpha. \forall \beta. \left((\alpha \rightarrow \beta) \ast [\alpha]\right) \rightarrow [\beta]. \text{ Type rule for map}(f, x): \\
\circ \text{ compute } t, \text{ the type of (arbitrary expression) } x, \text{ and check that } t \equiv [t_{el}] \text{ for some } t_{el}. \\
\circ \text{ get } f's \text{ signature from } ftable. \text{ IF } f \text{ does not receive exactly one arg THEN error()} \text{ ELSE } f : t_{in} \rightarrow t_{out}, \text{ for some } t_{in} \text{ and } t_{out}. \\
\circ \text{ IF } (t_{el} \equiv t_{in}) \text{ THEN } \text{ map}(f, x) : [t_{out}], \text{ ELSE error()}
\]

\[
\text{reduce} : \forall \alpha. \left(((\alpha \ast \alpha) \rightarrow \alpha) \ast \alpha \ast [\alpha]\right) \rightarrow \alpha. \text{ Type rule for reduce}(f, e, x): \\
\circ \text{ compute } t, \text{ the type of } e, \text{ and } t_x \text{ the type of } x, \text{ and check that:} \\
1 \ f : (t \ast t \rightarrow t), \text{ i.e., } f \text{ is an operator that can reduce, and that} \\
2 \ t_x = [t], \text{ i.e., } x \text{ is an array of element type } t. \\
\circ \text{ if so then reduce}(f, e, x) : t.
\]
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().
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: \textit{vtable} and \textit{ftable}.
Synthesized attribute: the expression’s type.

\begin{tabular}{|c|c|}
\hline
\textit{Check}_{Exp}(Exp, vtable, ftable) = \text{case } Exp \text{ of} \\
\hline
\textit{num} & \text{int} \\
\hline
\textit{id} & \text{\text{\texttt{t = lookup(vtable, name(id))}}} \\
& \text{if } (\text{\texttt{t == unbound}}) \text{ then } \text{\texttt{error(); int}}} \\
& \text{else } \text{\texttt{t}} \\
\hline
\textit{Exp}_1 + \textit{Exp}_2 & \text{\texttt{t}_1 = Check}_{Exp}(\textit{Exp}_1, vtable, ftable)} \\
& \text{\texttt{t}_2 = Check}_{Exp}(\textit{Exp}_2, vtable, ftable)} \\
& \text{if } (\text{\texttt{t}_1 == \texttt{int} and \texttt{t}_2 == \texttt{int}}) \text{ then } \text{\texttt{int}}} \\
& \text{else } \text{\texttt{error(); int}}} \\
\hline
\textit{Exp}_1 = \textit{Exp}_2 & \text{\texttt{t}_1 = Check}_{Exp}(\textit{Exp}_1, vtable, ftable)} \\
& \text{\texttt{t}_2 = Check}_{Exp}(\textit{Exp}_2, vtable, ftable)} \\
& \text{if } (\text{\texttt{t}_1 == \texttt{t}_2}) \text{ then } \text{\texttt{bool}}} \\
& \text{else } \text{\texttt{error(); bool}}} \\
\hline
\end{tabular}

Note: In Fasto equality of arrays is not supported!
Type Checking an Expression (Part 2)

\[ \text{Check}_{\text{Exp}}(\text{Exp}, \text{vtable}, \text{ftable}) = \text{case Exp of} \]

\[ \ldots \]

| if \ Exp_1 then \ Exp_2 else \ Exp_3 | \[ t_1 = \text{Check}_{\text{Exp}}(\text{Exp}_1, \text{vtable}, \text{ftable}) \]
| \[ t_2 = \text{Check}_{\text{Exp}}(\text{Exp}_2, \text{vtable}, \text{ftable}) \]
| \[ t_3 = \text{Check}_{\text{Exp}}(\text{Exp}_3, \text{vtable}, \text{ftable}) \]
| if \ (t_1 == \text{bool} \text{ and } t_2 == t_3) \text{ then } t_2 \]
| \[ \text{else } \text{error}(); t_2 \]

| let \ \text{id} = \ Exp_1 \text{ in } \ Exp_2 | \[ t_1 = \text{Check}_{\text{Exp}}(\text{Exp}_1, \text{vtable}, \text{ftable}) \]
| \[ \text{vtable}' = \text{bind}(\text{vtable}, \text{name}(\text{id}), t_1) \]
| \[ \text{Check}_{\text{Exp}}(\text{Exp}_2, \text{vtable}', \text{ftable}) \]

| \text{id } (\text{Exps} ) | \[ t = \text{lookup}(\text{ftable}, \text{name}(\text{id})) \]
| if \ (t == \text{unbound}) \text{ then } \text{error}(); \text{int} \]
| else \ ((t_1, \ldots, t_n) \rightarrow t_0) = t \]
| \[ [t'_1, \ldots, t'_m] = \text{Check}_{\text{Exps}}(\text{Exps}, \text{vtable}, \text{ftable}) \]
| if \ (m == n \text{ and } t_1 == t'_1, \ldots, t_n == t'_n) \text{ then } t_0 \]
| \[ \text{else } \text{error}(); t_0 \]
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 $t_1$.

\[
\text{Check}_{\text{Fun}}(\text{Fun}, \text{ftable}) = \text{case } \text{Fun} \text{ of}
\]

\[
\text{Type id } (\text{Typelds} ) = \text{Exp} & \quad \text{vtable} = \text{Check}_{\text{Typelds}}(\text{Typelds}) \\
t_1 = \text{Check}_{\text{Exp}}(\text{Exp}, \text{vtable}, \text{ftable}) \\
\text{if } ( t_1 \neq \text{Type} ) \\
\text{then error(); int}
\]

\[
\text{Check}_{\text{Typelds}}(\text{Typelds}) = \text{case } \text{Typelds} \text{ of}
\]

\[
\text{Type id} & \quad \text{bind(SymTab.empty(), id, Type)} \\
\text{Type id , Typelds} & \quad \text{vtable} = \text{Check}_{\text{Typelds}}(\text{Typelds}) \\
& \quad \text{if } ( \text{lookup(vtable, id) = unbound} ) \\
& \quad \text{then bind(vtable, id, Type)} \\
& \quad \text{else error(); vtable}
\]
Type Checking the Whole Program

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

\[
\text{Check}_{\text{Program}}(\text{Program}) = \text{case Program of}
\]

| Funs | ftable = Get_{Funs}(Funs) \\
|------|-------------------------|
|      | Check_{Funs}(Funs, ftable) \\
|      | if ( lookup(ftable, main) \neq () ) \rightarrow \alpha ) \\
|      | then error() |

\[
\text{Check}_{Funs}(Funs, ftable) = \text{case Funs of}
\]

<table>
<thead>
<tr>
<th>Fun</th>
<th>Check_{Fun}(Fun, ftable)</th>
</tr>
</thead>
</table>
| Fun Funs | Check_{Fun}(Fun, ftable) \\
|         | Check_{Funs}(Funs, ftable) |
Building the Functions’ Symbol Table

\[ \text{Get}_{\text{Funs}}(\text{Funs}) = \text{case } \text{Funs} \text{ of} \]

| Fun  | \((f, t) = \text{Get}_{\text{Fun}}(\text{Fun}) \)  
|      | \(\text{bind} (\text{SymTab}.\text{empty}(), f, t)\)  |
| Fun Funs  | \(\text{ftable} = \text{Get}_{\text{Funs}}(\text{Funs}) \)  
|      | \((f, t) = \text{Get}_{\text{Fun}}(\text{Fun}) \)  
|      | \(\text{if } (\text{lookup}(\text{ftable}, f) == \text{unbound}) \)  
|      | \(\text{then } \text{bind}(\text{ftable}, f, t)\)  
|      | \(\text{else } \text{error}(); \text{ftable}\)  |

\[ \text{Get}_{\text{Fun}}(\text{Fun}) = \text{case } \text{Fun} \text{ of} \]

| Type id (Typelds) = Exp | \([t_1, \ldots, t_n] = \text{Get}_{\text{Types}}(\text{Typelds}) \)  
|                         | \((\text{id}, (t_1, \ldots, t_n) \rightarrow \text{Type})\)  |

\[ \text{Get}_{\text{Types}}(\text{Typelds}) = \text{case } \text{Typelds} \text{ of} \]

| Type id | \([\text{Type}]\)  |
| Type id, Typelds | \(\text{Type} :: \text{Get}_{\text{Types}}(\text{Typelds})\)  |
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.

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

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

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 \( \text{ftable} \). 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 \( \text{map}(f, x) : [t_{out}] \),

- ELSE error()
Type Checking Map With Book Notations:

<table>
<thead>
<tr>
<th>$\text{Check}_{\text{Exp}}(\text{Exp}, \text{vtable}, \text{ftable}) = \text{case Exp of}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
| map( | $t_{arr} = \text{Check}_{\text{Exp}}(\text{Exp}_{\text{arr}}, \text{vtable}, \text{ftable})$
| \text{id}, | $t_{el} = \text{case } t_{arr} \text{ of}$
| $\text{Exp}_{\text{arr}}$ | \hspace{2cm} $\text{Array}(t_{1}) \rightarrow t_{1}$
| | \hspace{2cm} other $\rightarrow \text{error()}$
| | $t_{f} = \text{lookup}(\text{ftable}, \text{name}(\text{id}))$
| | $\text{case } t_{f} \text{ of}$
| | \hspace{2cm} \text{unbound} $\Rightarrow \text{error()}$
| | \hspace{2cm} if $t_{in} = t_{el}$ then $\text{Array}(t_{out})$
| | \hspace{2cm} \text{else } \text{error()}$
| | \hspace{2cm} otherwise $\Rightarrow \text{error()}$
| |

Remember:
- $\text{vtable}$ maps variable names to their types
- $\text{ftable}$ 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: ∀α.∀β.list(α) * list(int) * list(β)
else: ∀γ.∀δ.list(char) * list(γ) * list(δ)

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

Types $t_1$ and $t_2$ can be unified $\iff \exists$ 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: $\forall \alpha.\forall \beta.\text{list}(\alpha) \ast \text{list}(\text{int}) \ast \text{list}(\beta)$
else: $\forall \gamma.\forall \delta.\text{list}(\text{char}) \ast \text{list}(\gamma) \ast \text{list}(\delta)$

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

Types $t_1$ and $t_2$ can be unified $\iff \exists$ substitution $S$ s.t. $S(t_1) = S(t_2)$.

Most-General Unifier: the least specialized subs/type that still unifies
$S = \{ \alpha \leftarrow \text{char}, \gamma \leftarrow \text{int}, \delta \leftarrow \beta \}$ $\Rightarrow \text{list}(\text{char}) \ast \text{list}(\text{int}) \ast \text{list}(\beta)$
Example: Inferring the Type of SML’s `length`

```plaintext
fun length(x) = if null(x) then 0
    else length( tl(x) ) + 1
```
Example: Inferring the Type of SML’s \texttt{length}

\begin{verbatim}
fun length(x) = if null(x) then 0 else length( tl(x) ) + 1
\end{verbatim}

\begin{tabular}{lll}
\textbf{EXPRESSION} & \textbf{TYPE} & \textbf{UNIFY} \\
length & $\beta \rightarrow \gamma$ & \\
x & $\beta$ & \\
if & $bool \times \alpha_i \times \alpha_i \rightarrow \alpha_i$ & \\
null & $list(\alpha_n) \rightarrow bool$ & \\
null(x) & $bool$ & $list(\alpha_n) \equiv \beta$ \\
0 & $int$ & $\alpha_i \equiv int$ \\
+ & $int \times int \rightarrow int$ & \\
tl & $list(\alpha_t) \rightarrow list(\alpha_t)$ & $list(\alpha_t) \equiv list(\alpha_n)$ \\
tl(x) & $list(\alpha_t)$ & \\
length(tl(x)) & $\gamma$ & $\gamma \equiv int$ \\
length(tl(x)) + 1 & $int$ & \\
if(\ldots) then \ldots else \ldots & $int$ & \\
\end{tabular}
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)

```java
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 m\(_1\), .., m\(_k\) and n\(_1\), .., n\(_k\), \(\forall k\) ) then
    union(m, n); return unify(m\(_1\), n\(_1\)) and .. and unify(m\(_k\), n\(_k\));
(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)

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

```c
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.
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 B* 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

Step 1, Rule (IV)

Step 2, Rule (II)

Step 3, Rule (IV)
Structural Equivalence Example (3)

Each node is annotated with two integer values:

- REP
- node’s identifier

Step 4, Rule (IV)

Step 5, Rule (II)

Step 6, Rule (IV)

```
struct int *
R: 2, Id: 2
R: 3, Id: 3
1
struct
R: 4, Id: 4
R: 5, Id: 5
R: 6, Id: 6
R: 8, Id: 8
struct
R: 1, Id: 1
R: 3, Id: 3
R: 8, Id: 8
struct
R: 1, Id: 7
R: 3, Id: 9
1
3
```
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:

\[ ((a1 \to a2) \times \text{list}(a3)) \to \text{list}(a2) \]
\[ ((a3 \to a4) \times \text{list}(a3)) \to a5 \]

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:

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

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

Question 1: Do we need to implement type inference?
What Changes When Adding Arrays? (part 1)

Polymorphic Array Constructors and Combinators:

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

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

Question 1: Do we need to implement type inference?

Answer 1: No! \textsc{Fasto} supports a fixed set of polymorphic function whose types are know (or if you like, very simple type inference).
What Changes When Adding Arrays? (part 1)

Polymorphic Array Constructors and Combinators:

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

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

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

\[
\text{reduce}(g, e, \{x_1, \ldots, x_n\}) \equiv g(\ldots(g(e, x_1)\ldots, x_n)
\]

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:

map: \( \forall \alpha. \forall \beta. (\alpha \to \beta) \ast [\alpha] \to [\beta] \),
\[
\text{map}(f, \{x_1, \ldots, x_n\}) \equiv \{f(x_1), \ldots, f(x_n)\}
\]

reduce: \( \forall \alpha. ((\alpha \ast \alpha) \to \alpha) \ast \alpha \ast [\alpha] \to \alpha \)
\[
\text{reduce}(g, e, \{x_1 , \ldots, x_n\}) \equiv g(\ldots(g(e, x_1)\ldots, x_n)
\]

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)

map : ∀ α. ∀ β. (α → β) * [α] → [β]. Type rule for map(f, x):

- compute t, the type of x, and check that t ≡ [t_{in}] for some t_{in}.
- check that f : t_{in} → 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, ’Ts are unknown: UntypedExp=Exp<unit>
- After type checking types are known: TypedExp=Exp<Type>

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)

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

- 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 \( \text{map}(f, x) : [t_{out}] \).

**AbSyn** representation for map: \( \text{Exp<'T>} = \ldots \ |
\]

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

1st \( T \) is the input-array element type, e.g., \( t_{in} \),
2nd \( T \) is the output-array element type, e.g., \( t_{out} \).

checkProg : UntypedProg -> TypedProg

Type checking an expression/program now results in a new exp/prg,
where all ’Ts fields, initially unknown (’T=unit) of an expression 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

(* 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 ⇒ 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

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 ⇒ 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))