Type Checking

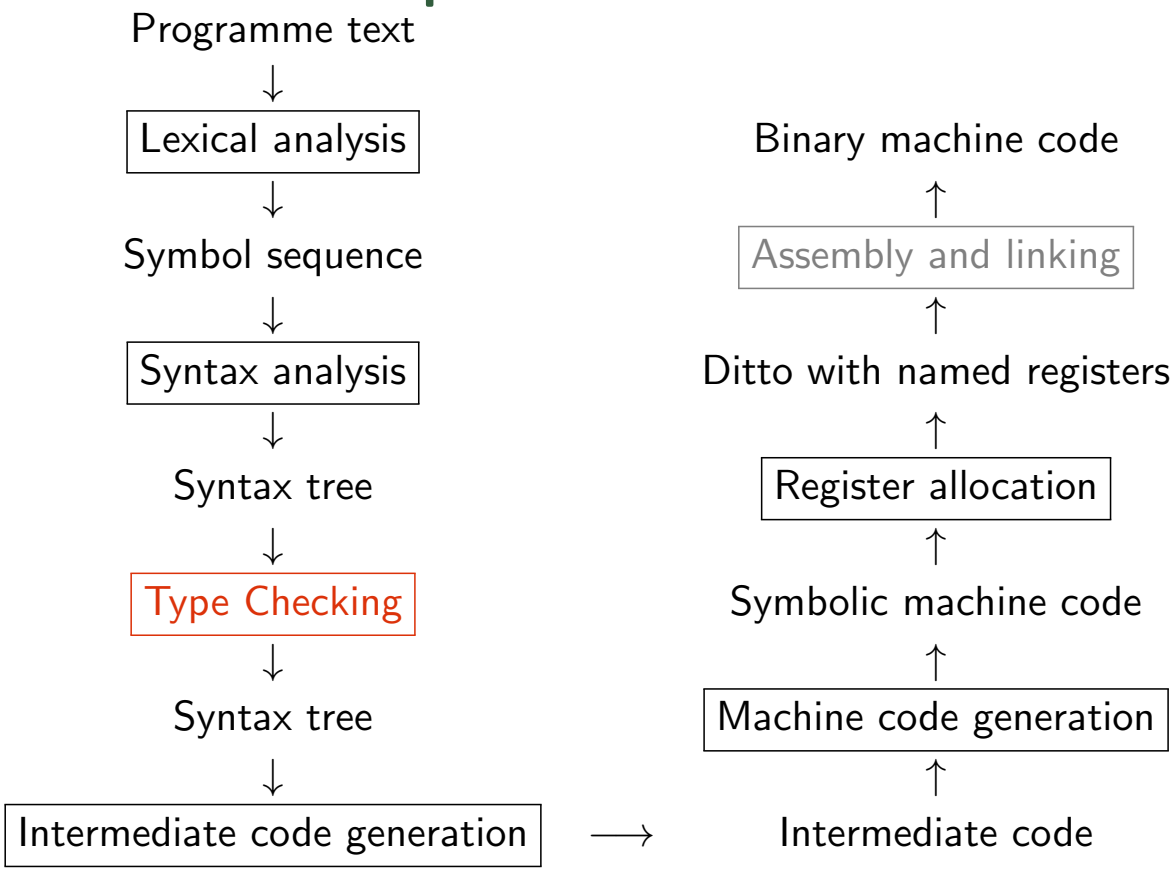
A significant part of this lecture notes uses Torben Mogensen's material. Many Thanks!

Cosmin Oancea
cosmin.oancea@diku.dk

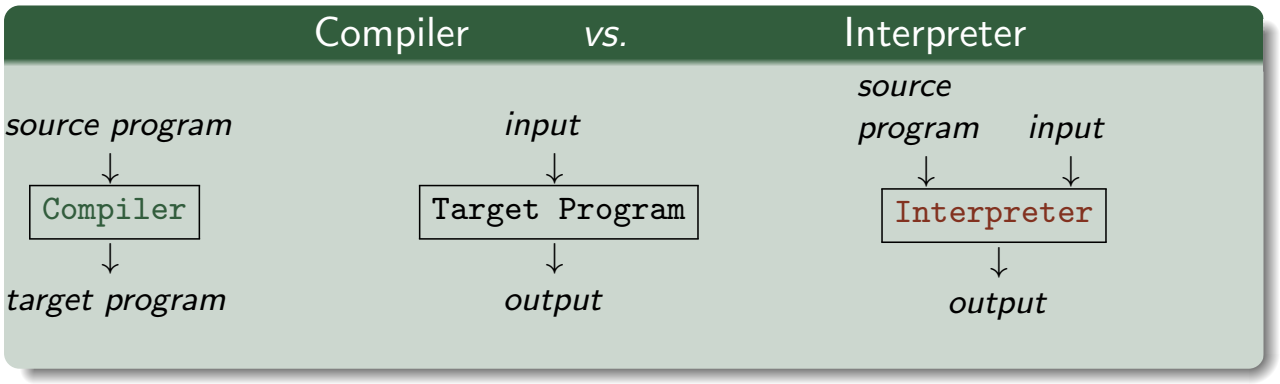
Department of Computer Science
University of Copenhagen

December 2012

Structure of a Compiler



Interpretation Recap



Why interpret? Debugging, Prototype-Language Implementation, etc.

Synthesized and 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?

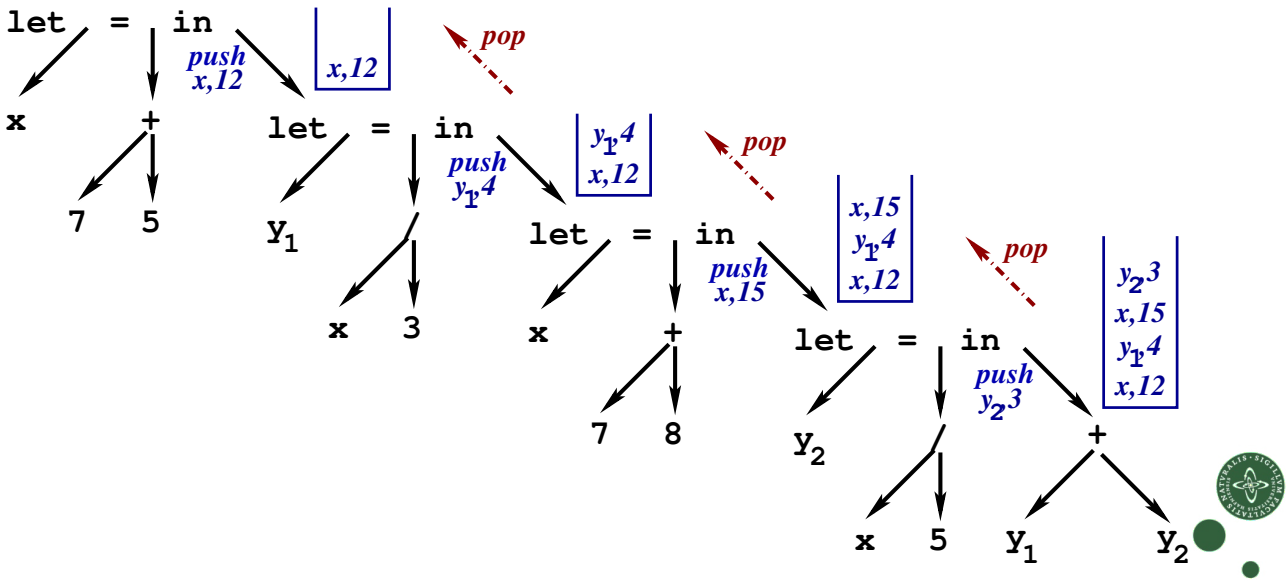
Synthesized: info passed upwards in the AbSyn traversal, i.e., from leaves to the root. Think: the result. Example?

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 *ftble*, in the interpretation of an expression:

$Eval_{Exp}$ (*Exp*, *vtable*, *ftable*) = ...



Example of Synthesized Attributes

The interpreted value of an expression / program is synthesized.

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

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

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 AbSyn 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.

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:

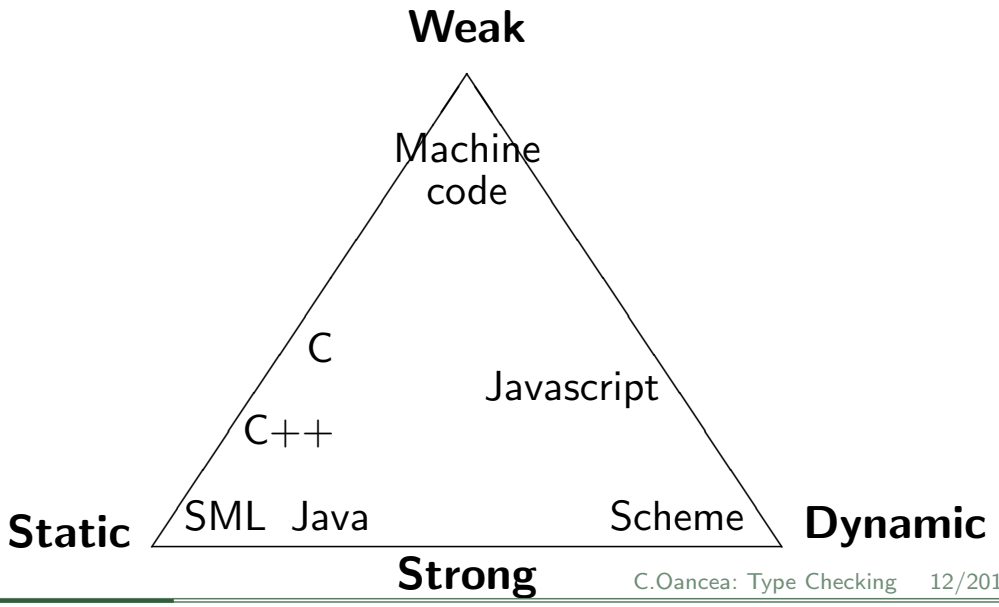
- `+, -` *expect integral arguments:* `a + (b=c)`
- `if`*-branch expressions have the same type:*
  `let a = ( if (b = 3) then 'b' else 11 ) in ...`
- *the type and number of formal and actual arguments match:*
  `fun int sum ([int] x) = reduce(op +, 0, x)`
  `fun [bool] main() = map(sum, iota(4))`
-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

Some language invariants cannot be checked statically: Examples?

Type Systems

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.



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 $(t_1, ..., t_n) \rightarrow t_0$, 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* **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* **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`
...	

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	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, name(\textbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$
id (*Exps*)	$t = lookup(ftable, name(\textbf{id}))$ *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)

- 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.

$Check_{Fun}(Fun, ftable)$ = `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**(); `int`

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

$Check_{Program}(Program)$ = `case` *Program* `of`	
Funs	$ftable = Get_{Funs}(Funs)$ $Check_{Funs}(Funs, ftable)$ *if* $lookup(ftable, \texttt{main}) \neq ( ) \rightarrow \alpha$ *then* **error**()

$Check_{Funs}(Funs, ftable)$ = `case` *Funs* `of`	
Fun	$Check_{Fun}(Fun, ftable)$
Fun Funs	$Check_{Fun}(Fun, ftable)$ $Check_{Funs}(Funs, ftable)$

Building the Functions' Symbol Table

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

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

$Get_{Types}(TypeIds) = \texttt{case}\ TypeIds\ \texttt{of}$	
$Type\ \textbf{id}$	$[Type]$
$Type\ \textbf{id}\ ,\ TypeIds$	$Type::Get_{Types}(TypeIds)$

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.

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: $\forall\alpha.\forall\beta.list(\alpha) * list(int) * list(\beta)$

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

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

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	:	$\beta \rightarrow \gamma$	
x	:	β	
if	:	$bool * \alpha_i * \alpha_i \rightarrow \alpha_i$	
null	:	$list(\alpha_n) \rightarrow bool$	
$null(x)$	:	$bool$	$list(\alpha_n) = \beta$
0	:	int	$\alpha_i = int$
$+$	:	$int * int \rightarrow int$	
tl	:	$list(\alpha_t) \rightarrow list(\alpha_t)$	
$tl(x)$	:	$list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
$length(tl(x))$	:	γ	$\gamma = int$
$length(tl(x)) + 1$	:	int	
if(...)	:	int	

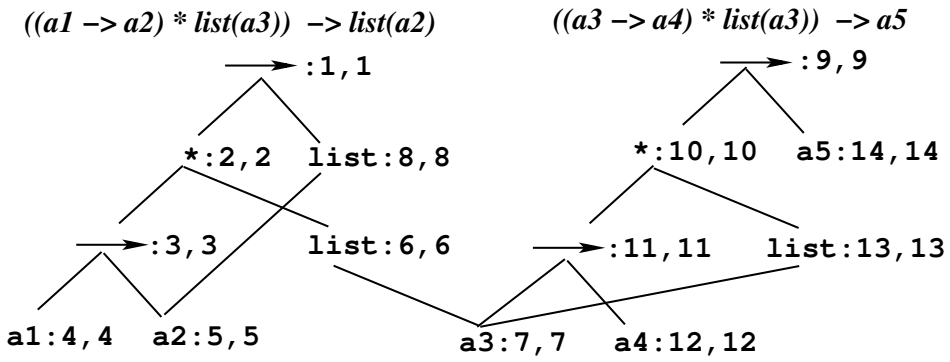
Most-General Unifier Algorithm

- 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 `n` and `m`:
  - if `n` is a type constructor then REP = `n`, (similar for `m`)
  - otherwise REP = either `n` or `m`

boolean unify(Node m, Node n)

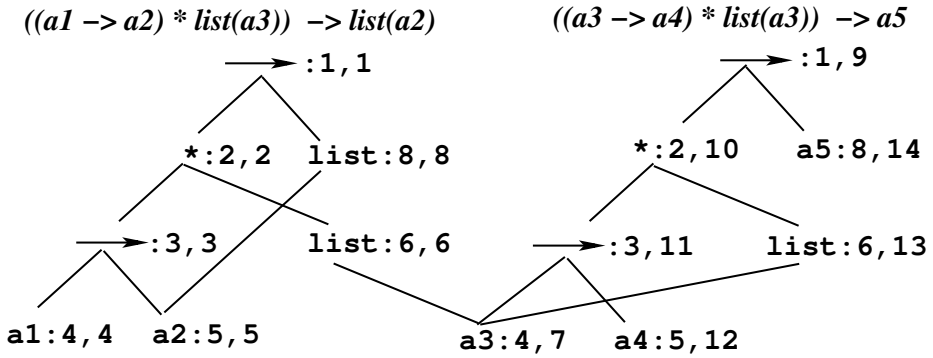
```
s = find(m); t = find(n);
if     ( s = t )                                  then return true;
else if ( s and t are the same basic type) then return true;
else if ( s or  t represent a type variable ) then
                                        union(s,t); return true;
else if ( s and t are the same type-constructor
          with children s_1,..,s_k and t_1,..,t_k, ∀ k ) then
    union(s,t); return unify(s_1,t_1) and .. and unify(s_k,t_k);
else                                                return false;
```

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(a2)) -> list(a2)

What Changes When Adding Arrays? (part 1)

Polymorphic Array Constructors and Combinators:

$\texttt{replicate} : \forall\alpha.\mathit{int} * \alpha \rightarrow [\alpha]$,
$\texttt{replicate(3, a)} \equiv \{\texttt{a, a, a}\}$.

$\texttt{map} : \forall\alpha.\forall\beta.(\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta]$
$\texttt{map(f, } \{\texttt{x}_1\texttt{,..,x}_n\}\texttt{)} \equiv \{\texttt{f(x}_1\texttt{),..,f(x}_n\texttt{)}\}$

$\texttt{reduce} : \forall\alpha.(\alpha * \alpha \rightarrow \alpha) * \alpha * [\alpha] \rightarrow \alpha$
$\texttt{reduce(g,e,}\{\texttt{x}_1\texttt{,..,x}_n\}\texttt{)} \equiv \texttt{g(..(g(e, x}_1\texttt{)..,x}_n\texttt{)}$

Question 1: Do we need to implement type inference?

Answer 1: No! 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:

$\texttt{replicate} : \forall\alpha.\mathit{int} * \alpha \rightarrow [\alpha]$,
$\texttt{replicate(3, a)} \equiv \{\texttt{a, a, a}\}$.

$\texttt{map} : \forall\alpha.\forall\beta.(\alpha \rightarrow \beta) * [\alpha] \rightarrow [\beta]$
$\texttt{map(f, } \{\texttt{x}_1\texttt{,..,x}_n\}\texttt{)} \equiv \{\texttt{f(x}_1\texttt{),..,f(x}_n\texttt{)}\}$

$\texttt{reduce} : \forall\alpha.(\alpha * \alpha \rightarrow \alpha) * \alpha * [\alpha] \rightarrow \alpha$
$\texttt{reduce(g,e,}\{\texttt{x}_1\texttt{,..,x}_n\}\texttt{)} \equiv \texttt{g(..(g(e, x}_1\texttt{)..,x}_n\texttt{)}$

Question 2: Assuming type-checking is successful, can we forget the type of `replicate(3, a)`?

Answer 2: No, the type of `a` needs to be remembered for machine-code generation, e.g., `a` : *int* vs `a` : *char*.

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

What Changes When Adding Arrays? (part 2)

`map`: $\forall\alpha.\forall\beta.(\alpha \to \beta) * [\alpha] \to [\beta]$. 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` : $t_{in} \to t_{out}$
- if so then `map(f, x)` : $[t_{out}]$.

AbSyn representation for `map`:

- `Exp =...| Map of string * Exp * Type * Type * pos`,
- *Before* type checking, both `Type`s are `UNKNOWN`. *After:*
- the first `Type` is the input-array element type, e.g., t_{in},
- the second `Type` is the output-array element type, e.g., t_{out}.

Type checking an expression/program now results in a new exp/prg, where all the `Type` fields of an expression are filled with known types

The Gist of Type.sml: Whole Program

Type-Checker Entry Point is Function `CheckProgram`

```
type TabEntry = string * (Type list * Type)
val  funTab : TabEntry list ref = ref []

fun checkProgram(funDecs : Fasto.FunDec list) : Fasto.FunDec list =
  let val tab = ("ord",([Fasto.Char (0,0)], Fasto.Int  (0,0)))::
                ("chr",([Fasto.Int  (0,0)], Fasto.Char (0,0)))::
                (List.map getType funDecs) (*ftable for declared funs*)
      (* Oversimplified: what did I omit to check? *)

      val () = funTab := tab (*global,to avoid passing it as param*)

      (* type checking each FunDec results in a new FunDec *)
      val decorated = List.map checkAndDecorate funDecs

      (* check main function exists and has type () -> int *) ...
  in decorated
  end
```

The Gist of Type.sml: Type Checking a Function

Compute the type of fun's body, check that it matches the result type

```
fun checkAndDecorate (fid, ret_type, args, body, pos) =

  (* args : (string * Type) list  can be used as vtable *)
  let val (body_type, newbody) = expType body args

  (* type rule: type of body equals the type of function's result *)
  in  (fid, typesMatch(body_type,ret_type), args, newbody, pos)
  end


fun typesMatch( Fasto.Int  p1, Fasto.Int  p2 ) = Fasto.Int  p1
  | typesMatch( Fasto.Bool p1, Fasto.Bool p2 ) = Fasto.Bool p1
  | typesMatch( Fasto.Char p1, Fasto.Char p2 ) = Fasto.Char p1
  | typesMatch( Fasto.Array(t1,p1), Fasto.Array(t2,p2) ) =
                            Fasto.Array(typesMatch(t1,t2), p1)
  | typesMatch( t1 , t2 ) = raise Error("Type error!")
```

The Gist of Type.sml: Type Checking an Expression

Map Type Rule: the type of array's elem equals the type of fun's arg.

```
fun expType exp vtab = case exp of
    Fasto.Num (n, pos) => (Fasto.Int pos, exp) | ...
  | Fasto.Map (fid, arr, argtype, restype, pos) =>
      let val (arr_type, arr_new) = expType arr vtab

          val el_type = case arr_type of
                Fasto.Array (t,p) => t
              | other => raise Error ("Map argument not an array")

          val (f_arg_type, f_res_type) =
              case SymTab.lookup fid (!funTab) of
                NONE => raise Error ("Unknown identifier!")
              | SOME ([a1],res) => (a1,res)
              | SOME (args,res) => raise Error("Map: not unary fun!")

      in ( Fasto.Array(f_res_type, pos),
           Fasto.Map( fid, arr_new, typesMatch(el_type, f_arg_type),
                      f_res_type, pos ) )
      end
```