/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CS4212 -- Compiler Design

T Y P E C H E C K E R F O R A C - L I K E

L A N G U A G E

The toy language has global and local scopes where variables and

procedures can be defined. All data declarations must be typed.

There is only one base type, namely 'int'. Composite types such as

pointers, arays, structures, and functions taking arguments of any

types and returning any type can be created. Due to restrictions

imposed by Prolog operators, there are significant syntactic differences

w.r.t. C, however, very simple systematic translation schemes between

the two languages exist.

The following are the type constructors:

int x ; : declares x to be of base type integer

T $ x ; : x is a pointer to type T

T x @ N ; : x is an array of size N,

with elements of type T

T x#(T1 arg1; : x is a function that takes n arguments,

...;Tn argn) ; of types T1,...,Tn. Arguments must be

separated by semicolon

Str ~~~ { ... } ; : declares the structure Str, having the

fields declared as normal 'variables'

within the curly braces.

Str ~~~ x, y, z ; : declare x, y, z to be of the structure

type Str. Str must have been declared

in advance.

Type-related operators:

$ : pointer dereference -- ($p) same as (\*p) in C

& : address-of operator, similar to C

\_@N : array indexing -- a@2 same as a[2] in C

\_#\_ : function application -- f#(2;3) same as f(2,3) in C

\_~~\_ : field selector from a structure -- s~~f same as s.f in C

The purpose of this code is to illustrate how types are recorded by the

compiler, and how type checking is performed so as to disallow

arbitrary interpretations of data. The current type system is slightly

more restrictive than the one of C, performing fewer casts. For

instance, an array will not be cast to its address when appearing on

the rhs of an assignment, and neither would a function. The address-of

operator must be used explicitly in such cases, to allow the assignment

to type-check. Arrays, functions, and addresses are not r-values, and

cannot be assigned to.

The following is an example of correct program. You will find a test

of the type-checker on this program at the end of the file, as an

illustration oh how the type-checker is expected to be used.

int x, % x is an integer

$y, % y is a pointer to an integer

z@10, % z is an array of 10 integers

$ $ $ zzz ; % zzz is a pointer to pointer to pointer to integer

int ($fptr)#( int aa; % fptr is a pointer to a function that

int bb; % takes 3 arguments and returns an integer

int ($pp)#(int x)) ; % -- the third arg is a pointer to function too

s ~~~ { int f1,$ f2@10 ; % s is a structure type, with 4 fields: an integer,

int e1,e3@10 } ; % an array of pointers to intgers, a second integer,

% and an array of integers.

s ~~~ p,q ; % p and q are variables of type s

int proc # ( int a; % proc is a procedure that takes 3 arguments

int b; % and returns an int. The third arg has a function type

int ($p)#(int x)) :: {

int c,d; % two local integer variables

int f # (int w) :: % a local (nested) function, can only be called from inside proc

{x = w ; return 1 } ;

if a > 100 then {

return a+1 % the type of the return expression will be checked against

} ; % the type declared to be returned by the procedure

ss ~~~ { int aa,bb } ; % declarations can appear anywhere; another structure with 2 fields

ss ~~~ sss ; % sss is a structure of type ss

sss ~~ aa = % assignment to the field 'aa' of structure 'sss'

2 + c +

f#d \* proc#(x;$y;p) ; % calling f and then proc, recursively

return ($pp)#(3) ; % calling the pointer to function, and returning its value from proc

} ;

int qq#(int d) :: % another function

{ return d+1 } ;

fptr = & proc ; % assignment of the address of proc to a pointer to function

z@8 = ($fptr)#(6;20;&qq) ; % calling a pointer to function

x = 1 ; y = & x ; z@1 = $y ; % examples of assignments, some involving function calls

x = proc#(3;4;&qq) ; % -- all data transfers are int

$ y = proc#(z@2;$y;&qq) ;

p ~~ f1 = q ~~ e1 ; % assignment of and to structure fields

x = $ (z+2) ; % pointer arithmetic, z+2 is a legal pointer

$ $ zzz = y ; % multiple dereferencing, a pointer to int is assigned

y = (p~~f2)@2 ; % access to an array element inside a structure

$ $ $ zzz = $((p~~f2)@2) % multiple dereferencing of pointers

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

:- op(1099,yf,;).

:- op(960,fx,if).

:- op(959,xfx,then).

:- op(958,xfx,else).

:- op(960,fx,for).

:- op(960,fx,while).

:- op(959,xfx,do).

:- op(960,fx,switch).

:- op(959,xfx,of).

:- op(1060,xfx,::). % procedure definition operator

:- op(100,xfx,#). % procedure call operator

:- op(950,fx,return).

:- op(1050,fx,int). % procedure return types

:- op(1050,fx,void). %

:- op(110,fy,$). % pointer dereference operator

:- op(105,yfx,@). % array access operator

:- op(107,fx,&). % address-of operator

:- op(115,yfx,~~). % structure access operator

:- op(1050,xfx,~~~). % structure definition operator

/\*

\* This is the main predicate, which computes the type of an

\* expression/statement. For expressions (i.e. syntactic constructs

\* that have \_value\_, the type is the type of the value. For type

\* declarations, the type returned is the atom 'type', and for all

\* other statements, the type is 'stmt'. The dummy types 'type' and

\* 'stmt' are important so as to allow us to check that structures

\* do not have any code in their definition.

\*

\* The arguments are as follows:

\* arg1 (input) : expression whose type is being calculated

\* arg2 (output) : the type returned for arg1

\* arg3 (input) : the input attributes

\* arg4 (output) : the output attributes

\*

\* Types are encoded as an "inside-out" representation of

\* the declaration. For instance, a declaration of the form

\*

\* int ($fptr)#( int aa; int bb; int ($pp)#(int x)) ;

\*

\* associates the following type representation with variable fptr :

\*

\* $(int#(int;int;$(int#int)))

\*

\* This representation has the syntactic tree

\*

\* $

\* |

\* #

\* / \

\* int ;

\* / \

\* int ;

\* / \

\* int $

\* |

\* #

\* / \

\* int int

\*

\* making it very clear that the type is a pointer to a

\* function that returns int, and takes 3 arguments: two ints

\* and a pointer to a function that returns int and takes an

\* argument of type int.

\*/

% Type checking of expressions (does not change the attributes,

% thus input and output attrs are always the same)

typecheck(X,T,A,A) :- atom(X), !, get\_assoc(type\_env,A,Local), member((X->T),Local),!.

% if X is a variable, then it must have been declared, so its name

% and associated type must be stored in the type\_env attribute

typecheck(N,int,A,A) :- integer(N), !.

% an integer constant has type 'int'

typecheck(E,($T),A,A) :- E =.. [Op,E1,E2], member(Op,[+,-]),

typecheck(E1,TE,A,\_), % pointer arithmetic: if p is a pointer, then p+k and p-k are legal pointers

same\_type(TE,$T), % same\_type checks if a cast is possible, so that E1 is interpreted as ptr

typecheck(E2,int,A,\_). % array names will be cast to pointers.

typecheck(E,int,A,A) :- E =.. [Op,E1,E2], % arithmetic expressions have integer operands and result

member(Op,[+,-,\*,/,rem,/\,\/,==,\=,=<,>=,<,>]),!, % binary operators are checked here

typecheck(E1,int,A,\_), typecheck(E2,int,A,\_).

typecheck(E,int,A,\_) :- % arithmetic expressions have integer operands and result

E =.. [Op,E1], member(Op,[\,-,+]), !, % unary operators are checked here

typecheck(E1,int,A,\_).

typecheck(($E),T,A,A) :- !, % type of a dereferenced expression: the type of E must be

typecheck(E,TE,A,\_), % equivalent to that of pointer to some type T

same\_type($T,TE). % and then the type of the result will be T.

typecheck((E1@E2),T,A,A) :-!, % array subscript expression

typecheck(E1,TE,A,\_), % the type of E1 must be equivalent to that of array with

same\_type(TE,@(T)), % elements of some type T, and then the result will be

typecheck(E2,int,A,\_). % of type T. The subscript E2vmust be of type int

typecheck((&E),($T),A,A) :-!, % address-of expression

( atom(E) ; E = \_ @ \_ % operator & can only be applied to a variable, an array element,

; E = \_ ~~ \_ ; E = $\_ ), % a structure field, or a dereferenced pointer

typecheck(E,T,A,\_).

typecheck((P#L),T,A,A) :- !, % function call expression

typecheck(args(L),LT1,A,\_), % compute the type of all args as a tuple LT1

typecheck(P,(T#LT2),A,\_), % the type of P recorded in type\_env must be of the pattern

same\_type(LT1,LT2). % T#LT2, where T is some return type, and LT2 is some tuple type

% Then, LT1 and LT2 must be cast-able to each other, and T will be

% the type of the function call expression.

typecheck(args(L),TL,A,A) :- !, % tuple of arguments. 'args' is just a wrapper to avoid confusion

( L = (H;T), TL = (TH;TT) % with statement sequencing { S1 ; S2 }, because the ';' is used there too

-> typecheck(H,TH,A,\_), % if the tuple has more than 1 component, compute the types by

typecheck(args(T),TT,A,\_)% recursing through all components

; typecheck(L,TL,A,\_) ). % otherwise, just compute the type of the singleton argument

typecheck((S ~~ F),T,A,A) :- !, % structure field access; F must be an atom, and S must be

atom(F), typecheck(S,~~~(TS),A,\_), % a declared structure. 'check\_struct\_acc' will check if

check\_struct\_acc(TS,F,T,A). % the field was declared, and retrieve its type.

% Type checking for statements. The returned type is either the atom

% 'type', that represents the fact that the program fragment subjected

% to type checking has only declarations (useful to check that the

% inside of a structure does not have any executable code -- which would

% not be syntactically acceptable); or the atom 'stmt', to represent

% that it does contain some executable code.

typecheck((E1 = E2),stmt,A,A) :- !, % assignment

typecheck(E1,T1,A,\_), % the types of the two sides must be cast-able to each other

typecheck(E2,T2,A,\_),

( atom(E1) ; E1 = \_ @ \_ % the LHS must be a l-value, i.e. a variable, array element,

; E1 = $ \_ ; E1 = \_ ~~ \_ ), % dereferenced expression, or structure field

same\_type(T1,T2),

writeln( 'Assignment' : E1=E2 | T1 = T2 ). % We print all the assignments, to have some verification

% 'if' and 'while' statements are straightforward. The inner scopes may

% have their own declarations, but these do not need to survive as we

% return to the outer scope, thus we do not care about the output attrs

typecheck((if B then { S }),stmt,A,A) :- !,

typecheck(B,int,A,\_), typecheck(S,stmt,A,\_).

typecheck((if B then { S1 } else { S2 }), stmt, A, A) :-

typecheck(B,int,A,\_), typecheck(S1,stmt,A,\_), typecheck(S2,stmt,A,\_).

typecheck((while B then { S }),stmt,A,A) :- !, typecheck(B,int,A,\_), typecheck(S,stmt,A,\_).

% Declarations must be stored in the type environment. All

% variable declarations start with 'int' or have ~~~ at the top level,

% so they are easy to identify

typecheck(int Decl, type, Ain, Aout) :- !,

process\_decl(int,Decl,Ain,Aout,0). % places the declared identifier in the type environment

typecheck( Str ~~~ { Fields } , type, Ain, Aout) :- !, % structure declaration

get\_assoc(type\_env,Ain,Orig\_env,A1,[]), % start with a fresh type environment, but preserve the old one

typecheck(Fields,type,A1,A2), % the {...} block of the structure must not contain executable code

% -- this is enforced by expected the atom 'type' as return type

get\_assoc(type\_env,A2,StructFields,A3,Orig\_env), % convert the computed type environment into a list of fields for the structure

% and restore the original type environment

get\_assoc(structures,A3,Structures,Aout,[(Str->StructFields)|Structures]).

% The output attributes are important here and must he handled with care

typecheck(Str ~~~ Decls, type, Ain, Aout) :- !, % variable declaration, whose type is a previously defined structure

get\_assoc(structures,Ain,Structures), % check that the structure has been declared

member(Str->\_,Structures),

process\_decl((~~~(Str)),Decls,Ain,Aout,0). % process the declaration, placing the declared variable in the type environment

typecheck({S},T,A,A) :- !, typecheck(S,T,A,\_). % handle local scope

typecheck((S;),T,A,A) :- !, typecheck(S,T,A,\_). % handle statements terminated with ';'

typecheck((S1 ; S2), T, Ain, Aout) :- !, % sequence of statements, attributes must be handled properly

typecheck(S1,T1,Ain,Aaux),

typecheck(S2,T2,Aaux,Aout),

( T1 = type, T2 = type, T = type, ! % compute the type of the statement; return 'type' only if both S1 and S2

; T = stmt ) . % have returned 'type' -- means there's no executable code in S1 and S2

typecheck(Head::{Body},stmt,Ain,Aout) :- !, % procedure definition

Head =.. [T,Rest],

process\_decl(T,Rest,Ain,A1,0), % compute the type of the function id, and place it in the type env

get\_assoc(type\_env,A1,[\_->Type#\_|\_]), % retrieve the return type of the function, and store it in the

get\_assoc(return\_type,A1,OrigRetType,A2,Type), % 'return\_type' attribute; save the original value so it can be returned later

typecheck({Body},stmt,A2,A3), % type-check the body; return expressions will be matched against val of return\_type

put\_assoc(return\_type,A3,OrigRetType,Aout). % restore the original return type, in case we're in a nested function

typecheck(return Expr,stmt,A,A) :- !, % return statement

typecheck(Expr,Type,A,\_), % type check the return expression, and match it against the return\_type attribute

get\_assoc(return\_type,A,T),

same\_type(Type,T). % The function's declared return type and the type of the return expression must be cast-able

/\*

\* Utility predicates, that compute types from declarations, check if

\* types are cast-able, and process structure declarations and accesses

\*/

% check structure access: retrieves a structure definition,

% checks if the field given as argument exists, and returns

% its type

check\_struct\_acc(TS,Field,Type,Attr) :-

get\_assoc(structures,Attr,Structs),

member((TS->Fields),Structs),

member((Field->Type),Fields).

% Helper that processes a declaration. A typical declaration has

% the format

% int <... id\_1 ... id\_n ...>

% or

% void <... id\_1 ... id\_n ...>

% (in which case, a function that returns void must be defined).

% The <... id\_1 ... id\_n ...> is an expression that has several

% identifiers embedded inside: these ar the identifiers being declared

% by the current declaration.

%

% It happens to be more convenient to separate the syntactic structure

% of the declaration into 2 parts: the "int" or "void", passed as the

% first argument, and the rest, passed as the second. The second

% argument may be a tuple, in which case every component must be

% processed individually, through recursion. Each declaration, if

% correct, is added to the 'type\_env' attribute.

%

% The Lev argument is increased every time we go into a nested tuple.

% This helps us achieve an important effect. When we process tuples of

% formal arguments in procedure definitions, we need these arguments to

% be available in the environment while we process the procedure's body.

% However, some of the arguments may be functions themselves, so they

% have their own tuples of arguments in the argument definition. For

% those nested tuples, we do not want to add the arguments to the

% current environment, since those are not variables with a legal scope

% in the body.

%

% To understand this issue better, look at the definition of 'proc' in

% the code example at the beginning of the file. This procedure has

% 3 arguments: a, b, and p. All these 3 arguments have a legal scope

% inside the body of 'proc', and must be available in the environment

% while processing the body. However, argument x is a formal argument

% that is part of the definition of p, and should not be allowed to

% appear in the body of 'proc'. This is enforced by increasing the Lev

% argument of the predicate as we go into the \_nested\_ tuple (actually,

% it's not really a tuple, since the function has only one argument, but

% it could be a tuple in the general case) that is part of the

% definition of p. Tuple components are added to the environment only

% when Lev is 0, and so a, b, and p will be added, while x will not.

%

% The 'inside-out' effect in representing a type is achieved by

% stripping type constructors from the second argument, and adding

% them to the first argument. When the second arg has become an atom

% (i.e. a variable name), the first argument has accumulated the type

% of that variable.

process\_decl(IntVoid,(Decl,DeclRest),Ain,Aout,Lev) :- !,

process\_decl(IntVoid,Decl,Ain,Aaux,Lev), % With a tuple of arguments, process each argument individually

process\_decl(IntVoid,DeclRest,Aaux,Aout,Lev). % and recurse through the rest of arguments.

process\_decl(AccumType,X,Ain,Aout,\_) :- atom(X), !, % Second arg is variable -> first arg has its type

get\_assoc(type\_env,Ain,Env,Aout,[(X->AccumType)|Env]). % Record the declaration in the type env

process\_decl(AccumType,$ T,Ain,Aout,Lev) :- % strip a pointer dereference operator; simply move it from the

process\_decl($AccumType,T,Ain,Aout,Lev).% second arg to the first

process\_decl(AccumType,T@\_,Ain,Aout,Lev):- % strip an array subscript operator; simply move it from the

process\_decl(@(AccumType),T,Ain,Aout,Lev). % second arg to the first. We do not record the size, though we should.

process\_decl(AccumType,T#L,Ain,Aout,Lev) :- % For function declarations, turn the tuple of arguments into a tuple of

get\_assoc(type\_env,Ain,Orig,A1,[]), % types first. Do that by starting with a fresh type env (but preserve orig one)

Lev1 is Lev+1, % Since it's a nested tuple, increase Lev

process\_tuple(L,A1,A2,Lev1), % Upon return, type\_env would have accumulated all argument types

get\_assoc(type\_env,A2,ArgTypes,Aaux,New), % Before using the argument types in building the current type,

( Lev == 0 % place the arguments in the environment if they don't come from a nested

-> append(ArgTypes,Orig,New) % tuple, so as these identifiers appear declared when processing the body.

; New = Orig ),

reverse(ArgTypes,RAT), % Arg types have been stored in reverse order, restore the order.

to\_tuple(RAT,Tuple), % Args have been stored into a list, convert back to tuple

process\_decl(AccumType#Tuple,T,Aaux,Aout,Lev). % Move function call to first arg and continue

% Process tuples of arguments for procedure definitions.

% Distinguish between non-singleton and singleton tuples.

% Pass on the current Lev to the recursive calls, so as to

% avoid nested arguments to be made available to the procedure's

% body

process\_tuple((Decl;Decls),Ain,Aout,Lev) :- !, % Non-singleton tuples

Decl =.. [Head,Rest],

process\_decl(Head,Rest,Ain,Aaux,Lev), % Each component must be processed as an independent declaration

process\_tuple(Decls,Aaux,Aout,Lev). % Recurse through the rest of the components

process\_tuple(Decl,Ain,Aout,Lev) :- % singleton tuples

Decl =.. [Head,Rest],!,

process\_decl(Head,Rest,Ain,Aout,Lev).

process\_tuple(void,A,A,\_). % Argument list can also be empty

% Convert a list of types into a tuple of types, for compatibility

% with the way types are encoded.

to\_tuple([],void) :- !.

to\_tuple([\_->H],H) :- !.

to\_tuple([\_->H|T],(H;Rest)) :- !, to\_tuple(T,Rest).

% Check if two types are cast-able. They must be if they are identical,

% but we also allow automatic casts between arrays and pointers

same\_type(int,int) :- !.

same\_type(void,void) :- !.

same\_type((~~~(X)),(~~~(X))) :- !.

same\_type(T1#Args1,T2#Args2) :- !,

same\_type(T1,T2), same\_type(Args1,Args2).

same\_type(($T1),($T2)) :- !, same\_type(T1,T2).

same\_type(($T1),@(T2)) :- !, same\_type(T1,T2). % an array identifier can be used as a pointer

same\_type(@(T1),($T2)) :- !, same\_type(T1,T2). % a pointer identifier can be used as an array

same\_type(@(T1),@(T2)) :- !, same\_type(T1,T2).

same\_type((T1H,T1L),(T2H,T2L)) :- !,

same\_type(T1H,T2H), same\_type(T1L,T2L).

same\_type((T1H;T1L),(T2H;T2L)) :- !,

same\_type(T1H,T2H), same\_type(T1L,T2L).

/\* test program, if there's no failure, the the program has type-checked successfully \*/

:- Program = (

int x, $y, z@10, $ $ $ zzz ;

int ($fptr)#(int aa; int bb;int ($pp)#(int x)) ;

s ~~~ { int f1,$ f2@10 ; int e1,e3@10 } ;

s ~~~ p,q ;

int proc#(int a;int b; int ($p)#(int x)) :: {

int c,d;

int f#(int w) :: {x = w ; return 1 } ;

if a > 100 then { return a+1 } ;

ss ~~~ { int aa,bb } ;

ss ~~~ sss ;

sss ~~ aa = 2+c+f#d \* proc#(x;$y;p) ;

return ($pp)#(3) ;

} ;

int qq#(int d) :: { return d+1 } ;

fptr = & proc ;

z@8 = ($fptr)#(6;20;&qq) ;

x = 1 ; y = & x ; z@1 = $y ;

x = proc#(3;4;&qq) ;

$ y = proc#(z@2;$y;&qq) ;

p ~~ f1 = q ~~ e1 ;

x = $ (z+2) ;

$ $ zzz = y ;

y = (p~~f2)@2 ;

$ $ $ zzz = $((p~~f2)@2)

),

empty\_assoc(E),

put\_assoc(type\_env,E,[],A1), % initialize attributes

put\_assoc(structures,A1,[],A2),

put\_assoc(return\_type,A2,none,A3),

typecheck({Program},\_,A3,\_). % type\_check program

:- same\_type($(@A),@($ (int)#(int))), writeln('A' = A).