Compilerconstructie

najaar 2012

http://www.liacs.nl/home/rvvliet/coco/

Rudy van Vliet
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs.nl

college 5, dinsdag 9 oktober 2012

Static Type Checking

6.1 Variants of Syntax Trees

Directed Acyclic Graphs for Expressions

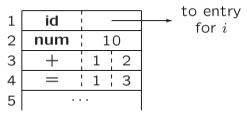
$a + a * (b - c) + (b - c) * d$

Syntax tree vs DAG

Producing Syntax Trees or DAG's

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (\ E\ )$	$T.node = E.node$
5) $T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**$,$**id**$.entry)$
6) $T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**$,$**num**$.val)$

Value-Number Method



Static Checking

- Type checks:
  Verify that type of a construct matches the expected one
- Flow-of-control checks:
  Example: break-statement must be enclosed in while-, for- or switch-statement
- ...

6.3 Types and Declarations

Types can be used for

- Type checking
- Translation
  Type information useful
  - to determine storage needed
  - to calculate address of array element
  - to insert explicit type conversions
  - to choose right version of operator
  - ...

Type Expressions

Types have structure

Example: array type `int[2][3]`

- Basic types: boolean, char, integer, float, void
- Type names: typedefs in C, class names in C++
- Type constructors:
  - array
  - record: data structure with named fields
  - $\rightarrow$ for function types: $s \rightarrow t$
  - Cartesian product $\times$: $s \times t$
  - ...

6.5 Type Checking

- Type system contains information about
  - Syntactic constructs of language
  - Notion of types
  - Logical rules to assign types to language constructs
    - if both operands of $+$ are integers, then result is integer
    - if f has type $s \rightarrow t$ and x has type s, then expression $f(s)$ has type t
- Sound type system
- Strongly typed implementation of language

A Simple Type Checker

A language example (Pascal-like)

- $P \rightarrow D; S$
- $D \rightarrow D; D \mid \textbf{id} : T$
- $T \rightarrow \textbf{boolean} \mid \textbf{char} \mid \textbf{integer} \mid \textbf{array}\,[\textbf{num}\,]\,\textbf{of}\,T \mid \hat{}\,T$
- $S \rightarrow \textbf{id} := E \mid \textbf{if}\ E\ \textbf{then}\ S \mid \textbf{while}\ E\ \textbf{do}\ S \mid S; S$
- $E \rightarrow \textbf{true} \mid \textbf{false} \mid \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E\ \textbf{and}\ E \mid E\ \textbf{mod}\ E \mid E[E] \mid E\hat{}$

A Simple Type Checker

Translation scheme for saving type of identifier

P	$\rightarrow$	$D; S$	
D	$\rightarrow$	$D; D$	
D	$\rightarrow$	**id** : T	$\{addType(\textbf{id}.entry, T.type);\}$
T	$\rightarrow$	**boolean**	$\{T.type = boolean;\}$
T	$\rightarrow$	**char**	$\{T.type = char;\}$
T	$\rightarrow$	**integer**	$\{T.type = integer;\}$
T	$\rightarrow$	$\hat{}\,T_1$	$\{T.type = pointer(T_1.type);\}$
T	$\rightarrow$	**array** [**num**] **of** T_1	$\{T.type = array(1 \ldots \textbf{num}.val, T_1.type);\}$

A Simple Type Checker

Type Checking of Expressions

E	$\rightarrow$	**true**	$\{E.type = boolean;\}$
E	$\rightarrow$	**false**	$\{E.type = boolean;\}$
E	$\rightarrow$	**literal**	$\{E.type = char;\}$
E	$\rightarrow$	**num**	$\{E.type = integer;\}$
E	$\rightarrow$	**id**	$\{E.type = lookup(\textbf{id}.entry);\}$
E	$\rightarrow$	E_1 **and** E_2	{**if** $(E_1.type == boolean)$ **and** $(E_2.type == boolean)$ **then** $E.type = boolean$; **else** $E.type = type\_error$;}
E	$\rightarrow$	E_1 **mod** E_2	{**if** $(E_1.type == integer)$ **and** $(E_2.type == integer)$ **then** $E.type = integer$; **else** $E.type = type\_error$;}
E	$\rightarrow$	$E[E]$	{**if** $(E_2.type == integer)$ **and** $(E_1.type == array(s,t))$ **then** $E.type = t$; **else** $E.type = type\_error$;}
E	$\rightarrow$	$E_1\hat{}$	{**if** $(E_1.type == pointer(t))$ **then** $E.type = t$; **else** $E.type = type\_error$;}

A Simple Type Checker

Type Checking of Statements

S	$\rightarrow$	**id** $:= E$	{**if** $(id.type == E.type)$ **then** $S.type = void$; **else** $S.type = type\_error$;}
S	$\rightarrow$	**if** E **then** S_1	{**if** $(E.type == boolean)$ **then** $S.type = S_1.type$; **else** $S.type = type\_error$;}
S	$\rightarrow$	**while** E **do** S_1	{**if** $(E.type == boolean)$ **then** $S.type = S_1.type$; **else** $S.type = type\_error$;}
S	$\rightarrow$	$S_1; S_2$	{**if** $(S_1.type == void)$ **and** $(S_2.type == void)$ **then** $S.type = void$; **else** $S.type = type\_error$;}

Type Equivalence

$S \rightarrow \textbf{id} := E$ {**if** $(id.type == E.type)$ **then** $S.type = void$; **else** $S.type = type\_error$;}

When are type expressions equivalent?

- Structural equivalence
- Name equivalence
- Use graph representation of type expressions to check equivalence
  - Leaves for basic types and type names
  - Interior nodes for type constructors
  - Cycles in case of recursively defined types. . .

Structural Equivalence

- Same basic type:
  integer is equivalent to *integer*
- Formed by applying same constructor to structurally equivalent types
  pointer($integer$) is equivalent to *pointer*($integer$)
- One is type name of other

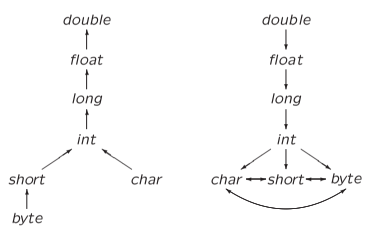
```
type  link = ^cell;
var   next : link;
      last : link;
      p    : ^cell;
      q, r : ^cell;
```


Type Conversions

$y = x + i$ with x float and i integer

- widening conversion
- narrowing conversion
- explicit conversion (= cast)
- implicit conversion (= coercion), automatically by compiler

Conversions in Java



Widening conversions Narrowing conversions

Coercion (Example)

Semantic action for $E \rightarrow E_1 + E_2$ uses two functions:

- $max(t_1, t_2)$
- $widen(a, t, w)$

```
Addr widen(Addr a, Type t, Type w)
{     if (t==w) return a;
      else if (t == integer and w == float)
           { temp = new Temp();
             gen(temp '=' '(float)' a);
             return temp;
           }
           else error;
}
```


Coercion (Example)

$E \rightarrow E_1 + E_2$ { $E.type = max(E_1.type, E_2.type)$;
$a_1 = widen(E_1.addr, E_1.type, E.type)$;
$a_2 = widen(E_2.addr, E_2.type, E.type)$;
$E.addr$ = **new** $Temp()$;
$gen(E.addr\ '='\ a_1\ '+'\ a_2)$;
}

Constructing Type Graphs in Yacc

```
enum Types {Tint, Tfloat, Tpointer, Tarray, ...};
typedef struct Type
{ Types type;
  struct Type *child
} Type;
```

- Type *mkint() construct int node if not already constructed
- Type *mkfloat() construct float node if not already constructed
- Type *mkarray(Type*, int) construct array-of-type node if not already constructed
- Type *mkptr(Type*)construct pointer-of-type node if not already constructed

Yacc Specification (Example)

from college 4

```
expr   : expr '+' term    { $$ = $1 + $3; }
       | term
       ;
term   : term '+' factor  { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'     { $$ = $2; }
       | DIGIT
       ;

%%
/* auxiliary functions section */
yylex()
{   int c;
    c = getchar();
    if (isdigit(c))
    {   yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```


Constructing Type Graphs in Yacc

Syntax-directed definitions

```
%union
{ Symbol *sym;
int num;
Type *typ;
}
%token INT
%token <sym> ID
%token <num> NUM
%type  <typ> type

%%
decl : type ID              { addType($2, $1); }
     | type ID '[' NUM ']'  { addType($2, mkarr($1,$4)); }
     ;
type : INT                  { $$ = mkint(); }
     | type '^'             { $$ = mkptr($1); }
     | /* empty */          { $$ = mkint(); }
     ;
```


Type Checking in Yacc

Syntax-directed definitions

```
%{
enum Types {Tint, Tfloat, Tpointer, Tarray, ...};
typedef struct Type
{ Types type;
  struct Type *child
} Type;
%}
%union
{ Type *typ;
}
%type <typ> expr

%%
expr : expr '+' expr { if ($1->type != Tint || $3->type != Tint )
                         semerror("non-int operands in +");
                       else
                       { $$ = mkint();
                         gen(int-add instruction for $1 and $3);
                       }
                     }
```


Type Coercion in Yacc

Syntax-directed definitions

```
%{ ... %}
%%
expr : expr '+' expr
       { if ($1->type == Tint && $3->type == Tint)
         { $$ = mkint(); gen(int-add instruction for $1 and $3);
         }
         else if ($1->type == Tfloat && $3->type == Tfloat)
         { $$ = mkfloat(); gen(float-add instruction for $1 and $3);
         }
         else if ($1->type == Tfloat && $3->type == Tint)
         { $$ = mkfloat(); gen(int2float instruction for $3);
           gen(float-add instruction for $1 and $3);
         }
         else if ($1->type == Tint && $3->type == Tfloat)
         { $$ = mkfloat(); gen(int2float instruction for $1);
           gen(float-add instruction for $1 and $3);
         }
         else
         { semerror ("type error in +");
           $$ = mkint();
         }
       }
```


L-Values and R-Values

- $E_1 = E_2$;
- What can E_1 and E_2 be?
  $i = i + 1$;
  $i = 5$;
- L-value: left side of assignment, location
  Example: identifier i, array acces $a[2]$
- R-value: right side of assignment, value
  Example: identifier i, array acces $a[2]$, constant 5, addition $i + 1$

L-Values and R-Values in Yacc

Syntax-directed definitions (C)

```
%{                          expr : expr '+' expr:
typedef struct Node                { if ($1->typ->type != Tint ||
{ Type *typ;                            $3->typ->type != Tint )
  int islval;                         semerror ("non-int operands in +");
} Node;                            $$->typ = mkint();
%}                                 $$->islval = FALSE;
%union                             gen(...);
{ Node *rec;                       }
}                                | expr '=' expr
%type <rec> expr                   { if ( !$1->islval || $1->typ != $3->typ )
%%                                     semerror ("invalid assignment");
                                   $$->typ = $1->typ;
                                   $$->islval = FALSE;
                                   gen(...);
                                   }
                                 | ID
                                   { $$->typ = lookup($1);
                                   $$->islval = TRUE;
                                   gen(...);
                                   }
```


Volgende week

- Practicum over opdracht 2
- Direct naar 302/304
- Komt spoedig online
- Inleveren 29 oktober

Compiler constructie

college 5

Static Type Checking

Chapters for reading: 6.1, 6.3.1, 6.3.2, 6.5.1, 6.5.2