Compilerconstructie

najaar 2013

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

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

college 5, dinsdag 8 oktober 2013

Static Type Checking

The Phases of a Compiler

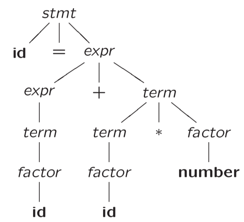
(from college 1)

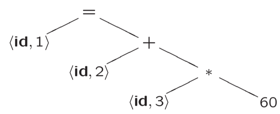
Token stream:

$\langle \mathbf{id}, 1\rangle \; \langle=\rangle \; \langle \mathbf{id}, 2\rangle \; \langle+\rangle \; \langle \mathbf{id}, 3\rangle \; \langle*\rangle \; \langle 60\rangle$

Syntax Analyser (parser)

Parse tree / syntax tree:





6.1 Variants of Syntax Trees

Directed Acyclic Graphs for Expressions

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

Syntax tree vs DAG...

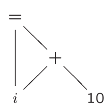
Producing Syntax Trees or DAG's

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

1) $p_1 =$ getLeaf(**id**, *entry*-a)
2) $p_2 =$ getLeaf(**id**, *entry*-a) $= p_1$
3) $p_3 =$ getLeaf(**id**, *entry*-b)
4) $p_4 =$ getLeaf(**id**, *entry*-c)
5) $p_5 =$ getNode$('-', p_3, p_4)$

Implementation: Value-Number Method

DAG for $i = i + 10$



1	**id**	→ to entry for i	
2	**num**	10	
3	+	1	2
4	=	1	3
5	...		

- Search array for (existing) node
- Use hash table

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
  - ...

6.3.1 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
  - $\to$ for function types: $s \to t$
  - Cartesian product $\times$: $s \times t$
  - ...

CFG for Function Declaration

$$
\begin{array}{lcl}
F & \rightarrow & B\ \textbf{id}\ (\mathit{OptL}) \\
B & \rightarrow & \textbf{int} \\
 & \mid & \textbf{float} \\
\mathit{OptL} & \rightarrow & \epsilon \\
 & \mid & \mathit{Ps} \\
\mathit{Ps} & \rightarrow & P \\
 & \mid & \mathit{Ps}, P \\
P & \rightarrow & T\ \textbf{id}
\end{array}
$$

CFG for Function Declaration

$$
\begin{array}{lcll}
F & \rightarrow & B\ \textbf{id}\ (\mathit{OptL}) & \{\ F.\mathit{type} = \rightarrow (\mathit{OptL.type}, B.\mathit{type});\ \} \\
B & \rightarrow & \textbf{int} & \{\ B.\mathit{type} = \mathit{integer};\ \} \\
 & \mid & \textbf{float} & \{\ B.\mathit{type} = \mathit{float};\ \} \\
\mathit{OptL} & \rightarrow & \epsilon & \{\ \mathit{OptL.type} = \mathit{void};\ \} \\
 & \mid & \mathit{Ps} & \{\ \mathit{OptL.type} = \mathit{Ps.type};\ \} \\
\mathit{Ps} & \rightarrow & P & \{\ \mathit{Ps.type} = P.\mathit{type};\ \} \\
 & \mid & \mathit{Ps}_1, P & \{\ \mathit{Ps.type} = \times(\mathit{Ps}_1.\mathit{type}, P.\mathit{type});\ \} \\
P & \rightarrow & T\ \textbf{id} & \{\ P.\mathit{type} = T.\mathit{type};\ \}
\end{array}
$$

6.3.2 Type Equivalence

$$
S \rightarrow \textbf{id} = E \ \ \{\textbf{if}\ (id.\mathit{type} == E.\mathit{type})\ \textbf{then}\ \ldots;\ \textbf{else}\ \ldots\}
$$

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;
```


6.3.3 Declarations

- We need symbol tables to record global and local declarations in procedures, blocks, and structs to resolve names
- Symbol table contains type and relative adress of names

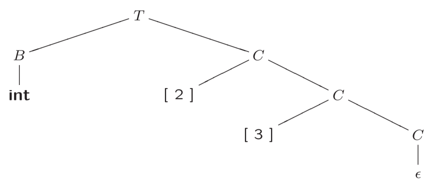
Example:

$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id}; D \ \mid\ \epsilon \\
T & \rightarrow & B\ C \ \mid\ \textbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \textbf{int} \ \mid\ \textbf{float} \\
C & \rightarrow & \epsilon \ \mid\ [\ \textbf{num}\ ]\ C
\end{array}
$$

Structure of Types (Example)

$$
\begin{array}{lcl}
T & \rightarrow & B\ C \ \mid\ \textbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \textbf{int} \ \mid\ \textbf{float} \\
C & \rightarrow & \epsilon \ \mid\ [\ \textbf{num}\ ]\ C
\end{array}
$$

`int[2][3]`



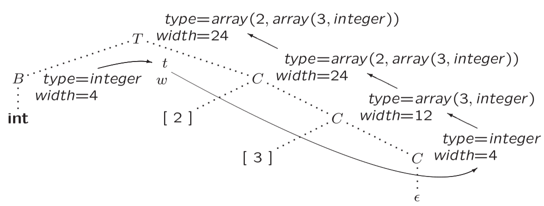
6.3.4 Storage Layout for Local Names

- Storage comes in blocks of contiguous bytes
- Width of type is number of bytes needed

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.\mathit{type};\ \ w = B.\mathit{width};\ \} \\
 & & C & \{\ T.\mathit{type} = C.\mathit{type};\ \ T.\mathit{width} = C.\mathit{width};\ \} \\
B & \rightarrow & \textbf{int} & \{\ B.\mathit{type} = \mathit{integer};\ \ B.\mathit{width} = 4;\ \} \\
B & \rightarrow & \textbf{float} & \{\ B.\mathit{type} = \mathit{float};\ \ B.\mathit{width} = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.\mathit{type} = t;\ \ C.\mathit{width} = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ C.\mathit{type} = \mathit{array}(\textbf{num}.\mathit{value},\ C_1.\mathit{type}); \\
 & & & \ \ C.\mathit{width} = \textbf{num}.\mathit{value} \times C_1.\mathit{width};\ \}
\end{array}
$$

Types and Their Widths (Example)

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.\mathit{type};\ \ w = B.\mathit{width};\ \} \\
 & & C & \{\ T.\mathit{type} = C.\mathit{type};\ \ T.\mathit{width} = C.\mathit{width};\ \} \\
B & \rightarrow & \textbf{int} & \{\ B.\mathit{type} = \mathit{integer};\ \ B.\mathit{width} = 4;\ \} \\
B & \rightarrow & \textbf{float} & \{\ B.\mathit{type} = \mathit{float};\ \ B.\mathit{width} = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.\mathit{type} = t;\ \ C.\mathit{width} = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ C.\mathit{type} = \mathit{array}(\textbf{num}.\mathit{value},\ C_1.\mathit{type}); \\
 & & & \ \ C.\mathit{width} = \textbf{num}.\mathit{value} \times C_1.\mathit{width};\ \}
\end{array}
$$



6.3.5 Sequences of Declarations

$$D \rightarrow T\ \mathbf{id};D \mid \epsilon$$

Use *offset* as next available address

$$\begin{array}{lll} P & \rightarrow & \{\ \mathit{offset}\ = 0;\} \\ & & D \\ D & \rightarrow & T\ \mathbf{id};\ \{\ \mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ \mathrm{T}.\mathit{type},\ \mathit{offset}); \\ & & \quad \mathit{offset}\ = \mathit{offset} + T.\mathit{width};\} \\ & & D_1 \\ D & \rightarrow & \epsilon \end{array}$$

6.3.6 Fields in Records and Classes

Example

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
x = p.x + q.x;
```

$$\begin{array}{lll} D & \rightarrow & T\ \mathbf{id};D \mid \epsilon \\ T & \rightarrow & \mathbf{record}\ '\{'\ D\ '\}' \end{array}$$

- Fields are specified by sequence of declarations
  - Field names within record must be distinct
  - Relative address for field is relative to data area for that record

Fields in Records and Classes

Stored in separate symbol table t

Record type has form *record*(t)

$$\begin{array}{lll} T & \rightarrow & \mathbf{record}\ '\{'\quad \{\ \mathit{Env.push}(\mathit{top});\ \mathit{top} = \mathbf{new}\ \mathit{Env}(); \\ & & \qquad\qquad \mathit{Stack.push}(\mathit{offset});\ \mathit{offset} = 0;\} \\ & & D\ '\}'\quad \{\ T.\mathit{type} = \mathit{record}(\mathit{top});\ T.\mathit{width} = \mathit{offset}; \\ & & \qquad\qquad \mathit{top} = \mathit{Env.pop}();\ \mathit{offset} = \mathit{Stack.pop}();\} \end{array}$$

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 $f(x)$ has type t
- Sound type system
- Strongly typed implementation of language

A Simple Type Checker

A language example (Pascal-like)

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

A Simple Type Checker

Translation scheme for saving type of identifier

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

A Simple Type Checker

Type Checking of Expressions

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

A Simple Type Checker

Type Checking of Statements

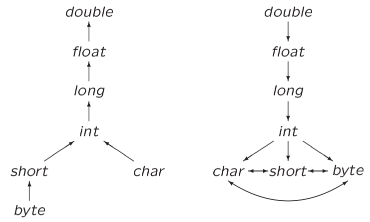
$$\begin{array}{llll} S & \rightarrow & \mathbf{id} := E & \{\mathbf{if}\ (\mathit{id.type} == E.\mathit{type}) \\ & & & \ \mathbf{then}\ S.\mathit{type} = \mathit{void};\ \mathbf{else}\ S.\mathit{type} = \mathit{type\_error};\} \\ S & \rightarrow & \mathbf{if}\ E\ \mathbf{then}\ S_1 & \{\mathbf{if}\ (E.\mathit{type} == \mathit{boolean}) \\ & & & \ \mathbf{then}\ S.\mathit{type} = S_1.\mathit{type};\ \mathbf{else}\ S.\mathit{type} = \mathit{type\_error};\} \\ S & \rightarrow & \mathbf{while}\ E\ \mathbf{do}\ S_1 & \{\mathbf{if}\ (E.\mathit{type} == \mathit{boolean}) \\ & & & \ \mathbf{then}\ S.\mathit{type} = S_1.\mathit{type};\ \mathbf{else}\ S.\mathit{type} = \mathit{type\_error};\} \\ S & \rightarrow & S_1;S_2 & \{\mathbf{if}\ (S_1.\mathit{type} == \mathit{void})\ \mathbf{and}\ (S_2.\mathit{type} == \mathit{void}) \\ & & & \ \mathbf{then}\ S.\mathit{type} = \mathit{void};\ \mathbf{else}\ S.\mathit{type} = \mathit{type\_error};\} \end{array}$$

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 $E \to E_1 + E_2$ 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)

$E \to E_1 + E_2$ { 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> typevar

%%
decl : typevar ID               { addType($2, $1); }
     | typevar ID '[' NUM ']'   { addType($2, mkarr($1,$4)); }
     ;
typevar : INT                   { $$ = mkint(); }
        | typevar '^'           { $$ = 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)

```
%{
typedef struct Node
{ Type *typ;
  int islval;
} Node;
%}
%union
{ Node *rec;
}
%type <rec> expr
%%
```

```
expr : expr '+' expr:
       { if ($1->typ->type != Tint ||
             $3->typ->type != Tint )
           semerror ("non-int operands in +");
         $$->typ = mkint();
         $$->islval = FALSE;
         gen(...);
       }
     | expr '=' 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
- Staat al online
- Inleveren 28 oktober

Over twee weken

- 's Ochtends hoorcollege
- 's Middags werkcollege

Compiler constructie

college 5
Static Type Checking

Chapters for reading: 6.1, 6.3, 6.5.1, 6.5.2