Lancaster University

Unit 11: Semantic Checking

SCC 312 Compilation

John Vidler

j.vidler@lancaster.ac.uk

Lancaster University

Aims

- In this unit we take a short look at the very important area of semantic checking.
- We focus on a core issue : type checking.
- We look at the spectrum of type checking available, from none to weak to strong.

Lancaster University

Aims

- We can type check statically (within the compiler and during compile-time) and we can check dynamically (during run-time).
- We focus on static checking.

- We look at the problems languages with powerful data structure modelling capabilities present type checking.
- We look at name and structure equivalence.

Lancaster University

Semantic Checking

- the major issue in semantic analysis of a program source text is *type checking*.
- we need to check that all identifiers and constants are used in accordance with the type rules of the language
- and we need to detect cases of implicit type conversion (if allowed by the language) and insert the appropriate conversion code.

Lancaster University

Semantic Checking

- early high-level languages tended to have only a fixed set of possible types (perhaps fixed and floating point numbers, and character strings)
- though even these languages would typically allow the declaration of arrays and sub-routines (with particular parameter and return value types)
- but modern languages will have much more extensive facilities for declaring new types of objects, including records/structures, pointers, etc.

Lancaster University

Dynamic Type Checking

- some languages have *dynamic type checking*
- any variable holds information about the type of its current value
- and the (run-time) operations of the language recognise the types of their operands and apply the appropriate version of the operation (or report an error).
- Polymorphism

Lancaster University

The Spectrum of Type Checking

- None
- Weak
- Strong

Lancaster University

No Type Checking

- Some languages have no type checking
- A value of any type can be used anywhere which is allowed by the syntax.
- This is especially the case with assembly languages
- The problem is that most bugs are thus discovered at run time rather than compile time, which is a much more difficult and expensive situation

No Type Checking

Memories from SCC.150 - Printing strings with syscalls

```
.data
message: .asciiz "Demo Text"
...

.text
      ...
      lw $a0, message
      addi $v0, $zero, 4
      syscall
      ...
```

```
.data
message: .asciiz "Demo Text"
...

.text
      ...
      la $a0, message
      addi $v0, $zero, 4
      syscall
      ...
```

The compiler doesn't care that what we're doing makes no sense!

Lancaster University

No Type Checking

- Total Freedom
  - The compiler will do literally whatever you ask of it
- Total Responsibility
  - The compiler will do **literally** whatever you ask of it
- In systems programming, this can be extremely useful
  - The usage of particular parts of memory can change based on the state of the system, not the semantics of the program (you're not in a sandbox anymore)
  - No interpretation of what the programmer means can remove the chance of 'translation errors'

Lancaster University

Weakly-Typed Languages

- Here it is easy to override the type checking of the language.
- For example in C characters can be treated as integers, and we can call functions without matching the formal parameters in number or type.
- This is ok if we know what we are doing, but it may lead to obscure bugs.

Weakly-Typed Languages

```
int a = 4;
char b = 'a';

a = a + b;

printf( "Output: %d\n", a );

$> Output: 101
```

The compiler probably won't even notice you're doing this... let alone warn you

```
function demo(){ ... }

void (* ref)(void) = demo;

ref = ref + 10;

*ref(); // Probably SEGFAULT
```

A modern compiler will probably ask you if you're sure if you do this...

But it will still do it, if pushed :)

Note: When writing C/C++, remember your compiler flags!
-Wall -Werror can be extremely helpful sometimes

Lancaster University

Strongly-Typed Languages

- Here type-checking is enforced in all areas of the language and any type conversions have to be explicitly indicated.
  - There is no mechanism for *coercion*
- The standard example is Ada; we avoid certain classes of bug, but there may be inflexibilities in programming.

Strongly-Typed Languages

```
type BASE is Integer;
type DERRIVED is A;
type OTHER is Float;

DERRIVED a := 4;
BASE b := 2;

a = a + b;
```

The compiler will happily allow this, as we declared that there is an explicit path from DERRIVED to BASE ...

```
type BASE is Integer;
type DERRIVED is A;
type OTHER is Float;

DERRIVED a := 4;
OTHER b := 1;

b = b + a;
```

... but will refuse to compile this, as although logically OTHER can fit a DERRIVED (aka. BASE, aka. Integer), there is no explicit path from one to the other

Lancaster University

Varieties of Semantic Checking

- We are concerned here with static type checking
  - That is, that performed by the compiler at compile time

- ...and we are interested in type equivalence
  - That is, when do two entities have the same type (for example, the left-hand side of an assignment and the expression on the right)?

Lancaster University

Simple Type Equivalence

- For simple types there is no problem:
  - We hold the type of each variable in the symbol table
  - We can hopefully deduce from the form of a constant (the literal) what type it is
  - We know for each operator what type its operands must be and what is the consequential result type
  - ...and we can apply the tests as we build the parse tree.

Simple Type Equivalence

(Example 1 - Declarations)

```
let first = "some text";

let second = first;
```

Identifier	Type
first	<STRING>

We get the type for 'first' from the details of the token from the lexical analyser, and add that to the symbol table

Simple Type Equivalence
(Example 1 - Declarations)

```
let first = "some text";

let second = first;
```

Identifier	Type
first	<STRING>
second	

When 'second' is declared, we see it references 'first' in its initialisation - so go find out what 'first' is

Simple Type Equivalence
(Example 1 - Declarations)

```
let first = "some text";

let second = first;
```

Identifier	Type
first	<STRING>
second	<STRING>

When 'second' is declared, we see it references 'first' in its initialisation - so go find out what 'first' is and use that.

Lancaster University

Type Conversion

- Sometimes we may wish to convert a value from one type to another (if this is possible).

- We may require the programmer to indicate this *explicitly* with what is sometimes called a cast
  - such as "`a := (float) b`" or "`a := float(b)`".

- Or it can be done *implicitly* by *type coercion*
  - if an operator requires a `float` and we have an `integer`, then the compiler could automatically insert code to do the conversion.

Simple Type Equivalence

(Example 2 - Type Coercion)

```
1. let a = 4;
2. let b = 10;
3. let c = 0.0;
4.
5. c = a + b;
```

Identifier	Type
a	<INTEGER>
b	<INTEGER>
c	<FLOAT>

- All the types here are known beforehand (we infer them from the declaration)
- At line 5, when the compiler reaches the completion of the <expression> block "a + b" it can assess if this is a possible operation.
  - In this case, it should be fine, as internally we can have INT->FLOAT conversion routines that can be inserted here
  - Note: This can be done in stages, with each <expression> chunk handled separately... and this has its own problems.
    - BODMAS? PEMDAS?

Lancaster University

Type Coercions

- Some languages (for example Ada) require any coercions to be explicitly indicated by the programmer

- Others (for example PL/1) had very elaborate mechanisms for allowing implicit coercions from one type to another.

Lancaster University

More Complex Types

- So far, each of the types discussed could be expressed as a simple enumeration (STRING|INTEGER|FLOAT| ... etc)

- Now consider more complicated types, in particular those which are specified by the programmer

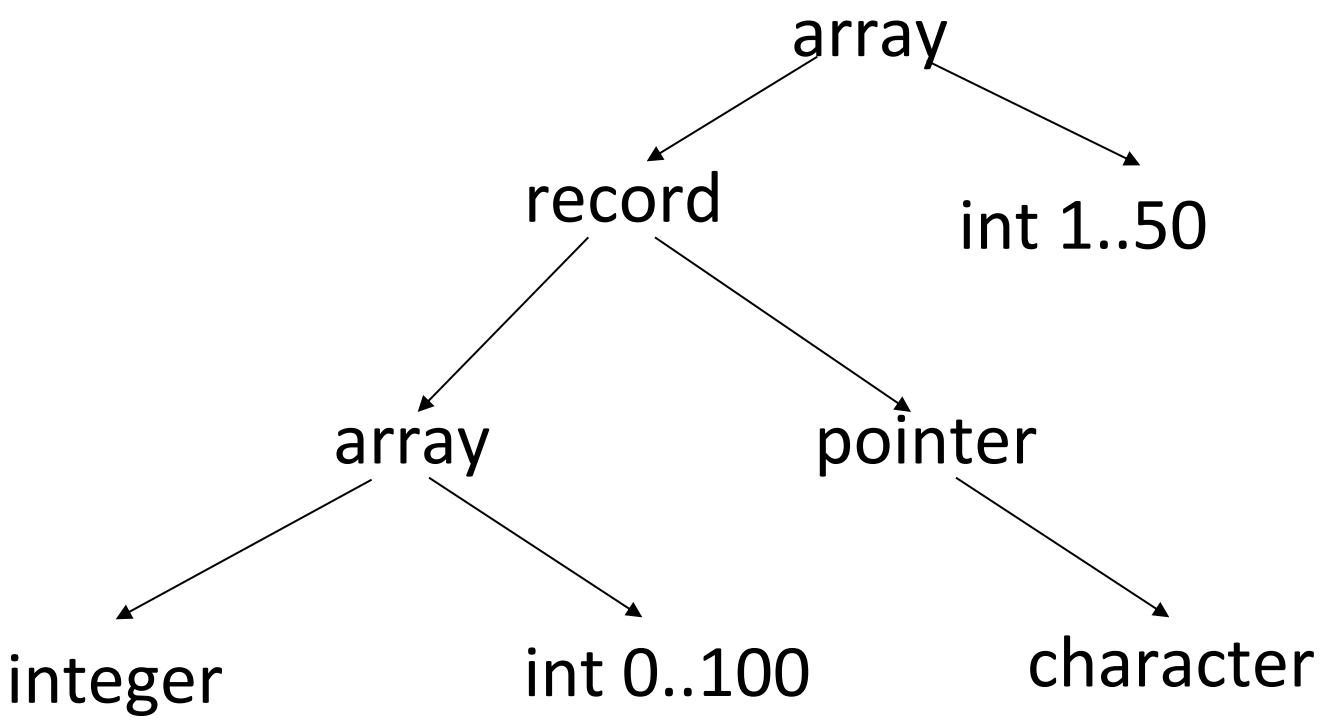
  - Arrays of records of sequences of types (arrays, records, pointers, etc)

  - Functions of various parameters with their own types and delivering another type, etc.

More Complex Types

- We could hold the *type* as a tree, so that our example would be held as something like shown.

```
array [1..50] of record
      array [0..100] of integer ;
      pointer to character ;
end record ;
```



Lancaster University

Type Checking : structure equivalence

- The type field in the symbol table entry is now a pointer to a structure of this form

- We check type equivalence by walking the two type trees and recursively checking for a match (or an allowable type conversion).

- Note that this structure may hold references to further structures in the interest of saving space - so walking the tree may become extremely expensive
  - Time/Memory trade-off

Lancaster University

Type Equivalence : two ways to check

- Many languages allow us to name new types, so that we can refer to the types (for example, in declarations) by their names rather than by repeating their definitions.

- But this means now there are two ways of checking type equivalence

Lancaster University

Name and Structure Equivalence

- *Name Equivalence :* we store a reference to the type name in the symbol table, and check the compatibility of two variables by simply checking the equivalence of the two names.

- *Structure Equivalence :* we use the type names to access the type definitions, and walk the two trees to check equivalence.

- These two mechanisms may give different results.

Lancaster University

Name and Structure Equivalence

- For example if we declare:

```
type someArray is array [1..100] of integer;
array1 : someArray ;
array2 : array [1..100] of integer ;
```

- Then the two arrays have the same type under structure equivalence, but not under name equivalence.

- The second array is said to have an anonymous type.

- A language design may (should) specify which type of equivalence is to be used.

Lancaster University

Structure Equivalence and Recursion

- If we are doing **structure** equivalence, notice that we have to be careful with recursive definitions.

- Consider

```
type node is record
     value : integer ;
     next : pointer to node ;
end record ;
```

- Here the type tree becomes a graph, so our type checking procedure must be capable of dealing with this (perhaps by using name equivalence for pointer types).
  - Remember our previous discussions regarding unspecified identifier types and forward declarations.

Lancaster University

Other Semantic Checking

- In discussing semantic checking we have concentrated on type checking.

- Other things we might want to check are *declaration checks* - we may want to enforce (if appropriate) declaration before use of variables.
  - C89-style C, for example

- We also need to deal correctly with the scope of variables; for example a variable in the body of a package may not be visible outside the package.

Scopes and Indentifiers

Identifier	Type
a	<STRING>
b	<INTEGER>

Inner Scope

Identifier	Type
b	<FLOAT>
c	<INTEGER>

- When our compiler is in the inner scope, the semantics of 'b' have changed, as the code redeclares a *local* 'b' with a new type (and presumably, a new memory location)
- If we want to access the outer 'b', a language feature must exist to support this.
  - "super.b" for example.
- 'a' is still accessable from the inner scope, of course.

Lancaster University

Other Semantic Checking

- *Flow of control checks* - certain types of statement may occur only inside certain constructions (for example, in Ada an "exit" statement must be inside a loop).

- This could be enforced with syntactic constraints, but often it is easier to do it as a separate "semantic" check.
  - To support this, metadata beyond the basic types might also have to be included in the scope structures.
    - What type of scope is this?
    - What operations are legal/valid for this location?
    - Any other constraints on access/operations here?

Lancaster University

Learning Outcomes

- You should understand the concept of semantic checking and how compilers support this.
- We've focused on type equivalence, so you should know about the spectrum of type checking, be aware of dynamic type checking, have an understanding of type equivalence and explicit/implicit coercions.
- You should understand the difference between structural and naming equivalences.

THE END