Semantic Analysis for μComp-lang

Letterio Galletta

Remember µComp-lang

µComp-lang is essentially an imperative language with components

Many simplifications

- datatypes: only int, char, bool variables, arrays, and references
- no compound data types, doubles, function pointers, ...
- no initializers in variable declarations
- Functions can return only int, char, void, bool
- References to scalar and array references are not interchangeable
- No dynamic allocation of memory
- No automatic type coercion

Scoping rules (1)

µComp-lang is a language with static scoping:

- Interfaces and components define their own scope
- Interfaces and components cannot be nested
- There is a global scope that contains interface and component declarations
- The declarations inside the global scope are mutually recursive
- The declarations inside a component are mutually recursive

Scoping rules (2)

µComp-lang is a language with static scoping:

- Blocks can be nested
- A variable `x` in a inner block hides a variables with the same name in an outer block
- Function declarations cannot be nested
- No function overloading

Component and interface rules (1)

- A component can only provides and uses interfaces
- A component must implement all the members defined in the interfaces it provides (the types of the members must be the same)
- The name of an interface must occur only once in a `provides` and `uses` list
- A component may provide two interfaces that have members with the same name as long as these members have the same type

Component and interface rules (2)

- The interfaces used by a component must not lead to **ambiguous** names, i.e. names provided by different used interfaces

Main function

µComp-lang does not support separate compilation:

Each source file must provide the definition of a component which provides the App interface

```
interface App {

  def main(): int;

}
```

this interface can be provided by only one component and never used

Library functions

We assume there exists a Prelude interface:

```
interface Prelude {

  def print(v : int);   // output an integer on the stdout

  def getint() : int;   // read an integer from the stdin

}
```

This interface is implicitly used by all components, and cannot be provided by any component

Types (1)

The types are given by the following grammar

$$T ::= int \mid bool \mid char \mid void \mid T\& \mid T[n] \mid T[]$$

$$\mid T1 \times \ldots \times Tn \rightarrow T \mid I[name] \mid C[name]$$

There are no type coercion:

- Booleans and characters cannot be converted to integers
- References to arrays and references to scalars are different types

Types (2)

- Arithmetic operators expect only integers
- Logical operators expect only boolean values
- _[_] operator expects an array and an integer as index
- Guards in `if` and `while` expect boolean values
- only functions can be invoked
- a function call must provides a number of arguments equals to the parameters of the function, and the types must agree

Other semantic rules

- Array should have a size of at least 1
- Array cannot be at the left of an assignment, i.e., array1 = arrayB, is not allowed
- Array references can occur only as formal parameter
- void variables are not allowed
- References T& require T to be a scalar type
- No partial application of functions
- Functions are not expressible values and there are no function pointers
- Functions cannot return references or arrays
- No multi-dimensional arrays

Automatic dereference (1)

Assume `x : int&` and `y : int`

```
y = x + 1;
```

The statement is well typed because `x` is automatically dereferenced, so `x + 1` has type `int`

Automatic dereference (2)

Assume `x : int&` and `y : int`

```
x = &y;
```

The statement is well typed, and it will assign the address of `y` to `x`

Automatic dereference (3)

Assume `x : int&`

```
x = x + 1;
```

The statement is well typed, and it will increment the value `x` points to by 1

At the end of semantic analysis

The result of this phase is

- a typed AST, where all member are qualified with the name of the corresponding interface, if the program is well typed
- an error, otherwise