Phase 2: Semantic Analyses

Compiler Construction
Radboud University

1 Background

The middle end of the compiler performs the different semantic analyses on the abstract syntax tree that the compiler produced in the first phase. The analyses include: binding time analysis (Section 6), type checking (Section 7), return type checking, and possibly much more.

2 Learning Outcome

Implement type inference and other semantic analyses on a non-standard language. If the specification is incomplete, make and document the well-founded design decisions about the changes.

3 Instructions

Implement at least type checking/inference and return path analysis for your compiler. Design and implement typing rules for expressions, statements and functions in Simple Programming Language (SPL). Other analyses may be included such as constant folding, specialisation of overloaded functions etc.

4 Deliverables

Presentation A third of the groups will present this assignment. Which groups have to present this assignment is announced on Brightspace.

In the presentation of this phase you tell the other students how your semantic analyses are set up and you demonstrate it. We all know the language and the theory, so you should just mention things specific to your analyses, like how you handled some interesting language constructions, and other interesting points.

Report sections Furthermore, finish the up to and including the third chapter of the report[1]. The provided skeleton contains example questions that you can answer but make sure the report is self contained.

5 Discusison

During the course you make your own implementation of a complete SPL compiler accompanied by a report that will be tried on the oral exam. The semantic analyses phase on its own is not

[1]The provided skeleton for the report is mandatory and can be found here: `https://gitlab.science.ru.nl/compilerconstruction/material`

graded but only as part of the total project. Comments are provided on the report sections that were handed in for guidance and affirmation.

Criteria for this phase can be found in Table 1.

6 Binding time analysis

Here you should check whether all applied occurrences of identifiers are associated with a proper definition.

There are three kinds of variables in SPL:

1. Global variables.
2. Function arguments, which can be used like local variables in the function body.
3. Local variables, which can be defined at the beginning of a function body.

In most programming languages local variables hide function arguments and global variables with the same name. Function arguments hide global variables with the same name. It is sensible to follow these rules in SPL.

Since SPL only has first order functions, the namespaces for functions and variables can be disjoint so that it is not a problem to use functions with the same name as variables.

Enforcement of scoping rules is usually not a separate phase in the semantic analysis, but a consequence of how environments are handled during type checking. Think about how you want to deal with name clashes and make sure that your implementation behaves accordingly. Do you want to produce warnings when a name hides another? Decide, document and test it! Write tests!

6.1 Polymorphism

Functions can be polymorphic, for example

```
firstTwo (x : [a]) : [a] {
    return (x.hd : x.tl.hd : []);
}
```

Polymorphic functions work for arguments of any type.

6.2 Overloading

Some operations are overloaded, for example the equality operator $==:: a \times a \rightarrow Bool$ can compare two values of the same type, and $print :: a \rightarrow Void$ can show values of any type. Most likely you need different implementations of these functions for different types. Your compiler needs to find a way to select the appropriate implementation based on the type of the argument.

Are there situations in SPL where solving overloading at compile time is very difficult. For instance, what happens if you use an overloaded operation on a *polymorphic* argument.

```
idPrint (x : a) : a {
    print (x);
    return x;
}
```

Since type information is needed during code generation, it is not sufficient to just check that programs are well typed. You also need to decorate the syntax tree with type information so that code generation can make use of it. Of course you do not have code generation yet, so just let your pretty printer print the decorated syntax tree.

Producing fancy error messages is difficult.

7 Type checking

If you just implement type checking, type annotations for functions are mandatory, and the **var** keyword for variable declarations cannot be used. You have to change the grammar for that.

In an imperative language like SPL you need to do more than just check if every expression is well typed. For instance, you need to check that a **Void** function never returns a value, and that a function that should return a value indeed returns a value of the proper type in every code path. It does help when you know the proper binding of variables and functions.

7.1 Type inference

The grammar of SPL allows leaving out type annotations for functions. Furthermore, when declaring a local or global variable, the **var** keyword can be used instead of a type.

```
firstTwo (x) {
    var r = x.tl;
    return (x.hd : r.hd : []);
}
```

7.2 Polymorphic Type Checking

Polymorphic type checking still requires programmers to always specify type signatures, but type signatures can contain type variables. The changed rules in the grammar are as follows.

In this language it is possible to write polymorphic functions, but the programmer always has to specify type signatures. Type checking is still depth-first tree traversal, and types are just compared for equality. Type variables are nothing special, the type checker treats them like base types. Different type variables are different types. You must implement generalization and instantiation, but only for global functions. The value restriction still applies, which means you never generalize local variables.

```
// Internally, the type gets generalized to ∀a.a → a
id (x : a) : a  {
    return x;
}

// You should now be able to write the polymorphic list reversal
reverse (list : [a] ) : [a]  {
    // ...
}

main ()  {
    // The type of id must be instantiated differently for each occurrence
    print(id(5));
    print(id(True));
```

```
}

// This function should give a type error
const (x : a, y : b) : a {
    return y;
}
```

7.3 Polymorphic Type Inference

Functions can be polymorphic, and the types of local variables can be inferred. You need all the machinery you have seen in the lecture with substitutions, unification, generalization and instantiation. The following program should typecheck and work as expected.

```
id (x) {
    return x;
}

main () {
    var x = 10;
    var y = x;
    print(id(y));
    print(id(True));
}
```

Property	Insufficient	Sufficient	Good	Optional
Design	No motivated design decisions.	Motivated design decisions.		
Errors	No error handling.	Errors that refer to an AST node.	Localised errors that are as precise as possible.	Reparations.
Type checking	Monomorphic type checking or monomorphic type inference	Polymorphic type inference.	Polymorphic type checking.	
Overloading	No support for overloaded builtins.	Support for overloaded builtins. Overloaded functions work for basic types only.	Overloaded equality and printing work for any type.	Overloaded builtins may make a polymorph function overloaded.
Other	Variables and functions must be defined before they are used.	Variables and functions may be used before they are defined. Return path analysis and binding time analysis.	Mutual recursion is supported.	
General	Non-working code.	Working code with a good I/O interface.	Well structured modular code.	Use of CI and issue tracker, other useful extensions regarding this phase.

Table 1: Assignment 2 rubric