Implicit Conversions for Amy Compiler

Compiler Construction '13 Final Report

Alexandre Falardeau

EPFL

{alexandre.falardeau}@epfl.ch

1. Introduction

During the semester, we were tasked with building a compiler that handled a subset of the Scala language. This subset language included the following features:

- Literal support for String, Integer, Unit and Boolean types
- Algebraic data type definition and instantiation
- A multitude of binary operators
- function definition and application
- If-else-else statements
- A match case structure

Amy is a statically typed language and for the purposes of this report, we will only consider the compiled version of the language (as opposed to the interpreted part). As such, the compiler contained 5 major sections. Execution of the compiler consisted in the each of these sections transforming the result of the previous, with the final output being executable WebAssembly code.

The first section is the lexer. Its role is to take the human written code and to transform it into a list of relevant token. It completes these by taking an ordered list of regex expression which transform key code expressions (such as "if", "else", parentheses, comments, etc.) into tokens. By finding matches to these regex in a greedy fashion, giving priority to matches defined first in the list, we get a list of tokens that we can use in the parser. Useless tokens (such as comments and whitespace) are filtered at this step. The second section is the parser. Its role is to transform the list of tokens into an AST following the rules defined by Amy's grammar. As to parse in linear time, the grammar defined by the parser needed to be LL1. The parser was built using the scallion library.

The third section is the name analyser. This section verifies that the code conforms to Amy's different naming rules (for example making sure a variable is declared before being used). The name analyser does this by first discovering ADT and function declerations and filling the symbol table. It then verifies the program and transforms a valid program into a symbolic AST, where each named element of a node refers to a unique element in the symbol table.

The fourth section is the type checker. In this section, we first iterate through every node of the AST to generate type constraints for the code. Then, using a unification algorithm, we make sure that no constraint causes a type conflict.

The fifth section is the WebAssembly code generator. This section takes an AST and converts to WebAssembly code using defined rules to convert every AST node.

The goal of this extension is to allow the developer write implicit conversion functions which can then be used to resolve some type conflicts without cluttering the code. A good example use case would be printing elements of code to console. It can become very tedious to make calls to a conversion function every time a code wants to print a non string variable to the console. If an implicit function exists mapping that data type to a string, the code will type check and will convert that variable using the defined implicit function.

2. Examples

2.0.1 Valid examples

The first code example demonstates the main use case for the feature.

```
object Base {
    implicit def int2bool(i: Int): Boolean = {i == 1}
    val b: Integer = true;
    2 || false;
    2 || b;
    2 == false;
    if (true) {
        2
    } else {
        true
    }
}
```

Given that a certain expression in the code needs to be a certain type, an implicit conversion can be used to transform an expression with the incorrect type to an expression with the correct type according to a defined rule (the implicit function). There are two main cases of this feature being helpful.

Given an expression that has an absolute type requirement (i.e. it needs to be a given type for the code to type check), implicit functions can be used to rectify that type. In the above example, the "——" operator requires all operands to be of boolean type, therefore the operator would create an absolute type requirement. Because of this, the int operand would be converted to a boolean expression with the int2bool function.

Given an expression that has a relative type requirement (i.e. one that depends on context, such as an expression that needs to be the same type as another, where any type is valid), the compiler will analyse the AST and apply the implicit function at the correct location so that the code type checks.

2.0.2 Error cases

For the above features to work as expected, we have found a few edge cases which demonstrate incorrect use of the implicit function.

```
object BadForm {
    implicit def invert(b: Boolean): Boolean = {!b}
    implicit def bool2int(b: Boolean, n: Int): Int = {
        if (b) {n} else {−1}
    } // Bad conversion error
}
```

The first error case refers to a faulty implicit function signature. Because these functions can only be used for type conversion, they must only take one operand and have a return type that is different that the operand's type.

```
object Ambiguous {
    implicit def bool2int(b: Boolean): Int = {
        if (b) { 3 }
        else { −1 }
    }
    implicit def bool2int2(b: Boolean): Int = {
        if (b) { 4 }
        else { −9 }
    }
    2 || false // Ambiguous conversion error
}
```

The second error case throws an ambiguous conversion error. If there exists two implicit functions in the same module with the same type signature, the compiler would never know which to use and as such throws an error. This is important to avoid random behavior.

```
object Transitive {
    implicit def int2Bool(i: Int): Boolean = {i != 0}
    implicit def bool2Str(b: Boolean): String = {if (b) {"true"}
    "Hello World" ++ 3 // Type error
}
```

The third error case simply demonstrates a limitation of the implicit function. Implicit functions cannot be applied transitively. If a function from type A to B and B to C exist, that does not imply that we can converts expressions of type A to one of type C. The above case would still not type check despite its validity if we could apply two implicit functions to the faulty expression.

```
object Ambiguous {
    implicit def bool2int(b: Boolean): Int = {
        if (b) { 3 }
        else { −1 }
    }
    implicit def int2bool(i: Int): Boolean = {i == 3}
    4 == true // Ambiguous conversion error
}
```

The fourth error cases demonstrates another case of ambiguous error. When two expressions' types must be equal for the program to type check, but can be any type, the compiler throws an error when these two expressions have different types and implicit conversions exist to convert expression to either of these types to either of the other. We have decided to have this limitation instead of prioritising some implicit conversion to facilitate debugging of Amy code.

3. Implementation

3.1 Theoretical Background

3.2 Implementation Details

Modifications in the lexer, parser and the code generator are very trivial and therefore will not be explained in depth. The lexer just needs to be updated to identify the ”implicit” keyword, the parser updated to differentiate functions and implicit functions and the code generator does not change because no new AST node is required for this extension.

The only important note is that I created a new ADT to facilitate the implementation of the extension. When the parser identifies an implicit function, it will generate a ”ConversionDef” instead of a ”FuncDef”, which both extend the ”ClassOrDef” abstract class. The two case classes' fields are identical, but the new type is useful to differentiate functions in the Name analysis step.

3.2.1 Name analysis modifications

In this section, implicit functions are ”discovered” by traversing the program AST in the same way normal functions are class definitions are discovered. Nevertheless, we had to add features to the symbol table to support a different type of function. The symbol table holds a map that relates a tuple containing a module, and two types (first the from type and second the to type) to the identifier of the conversion function signature. When adding a conversion function's type signature is verified to make sure it accepts only one parameter and returns a value of a different type. Analogously, we have added a getter for conversion functions that accepts the above tuple as parameters. Otherwise, conversion functions are treated like normal functions and are registered as such. Because the new map tracks which functions are conversion functions, the ”ConversionDef” ADTs are converted to ”FuncDef” when rebuilding the AST.

3.2.2 Type checker

The first major modification in the type checker affects the way the unification algorithm checks code. We have defined a new ”Conversion” ADT containing information about the expression in the AST which needs to be converted and the function signature of the conversion function. The unification algorithm now returns a list of all the conversions that are necessary for the program to type check. When it identifies a conflict, instead of immediately throwing an error, it checks if there exists a conversion function that maps the found type to the expected type. If there exists such a conversion, the unification will return it along with all other conversions. If it does not, a type error is thrown just like before.

If the unification function finishes and returns a list of conversion functions, this means that the program will type check after the AST transformation. The AST transformation is the second major modification to the type checker. When all conversions are created, they are passed to an object named the ”ConversionMapper” which will keep a copy of these conversions in a ListBuffer. This change of data structure is necessary because conversions will be deleted as the reconstruction of the AST executes. When an expression contained within a conversion is found, it will be wrapped in a ”Call” AST node, with the previous expression being used as the operand. For relative operators (such as ”if-then-else” and ”==”), both children are converted first and then verified to make sure they have not wrapped in conversion functions with opposing types. Whenever a conversion is used to wrap a node, it is deleted from the ListBuffer. This process continues until the ”ConversionMapper” iterates through the entire AST.

4. Possible Extensions

The implicit function currently has one big problem.

While both of these programs should type check, only the first does. Because of the way the constraints are generated (constraints for left expressions are generated before constraints for right) and the fact that the unification algorithm eliminates the constraints in order, when the compiler is faced with a (Expects: Int, Found: Bool) constraint, it does not know to go look for the sibling AST node that can be corrected. This feature does not work because I was unable to find a general way to track the proper expression to be converted while completing the elimination of constraints.

This extension also does not consider recursive implicit function application. Because a given conversion function could get reconstructed to have a call to itself, our extension can right now infinite loops into the code.

If I were to continue the implementation of this compiler (after fixing all of the existing bugs), I would most likely implement more implicit features to aid the reduction of clutter in code. An another feature could be for functions to accept implicit parameters. I believe that the challenge in that extension would be properly scope implicit values as to pick the right one for the function application.

References