Tuples

Compiler Construction '9 Final Report

Mélissa Gehring   Tibor Vaughan   Kilian Schneiter   Capucine Berger

EPFL

melissa.gehring@epfl.ch   tibor.vaughan@epfl.ch   kilian.schneiter@epfl.ch   capucine.berger@epfl.ch

1. Introduction

We started by implementing a lexer. The goal was to take as an input a list of characters and output a list of words. These words were then parsed into tokens, to know what they were referring to. Once we implemented the parser, the words needed to be analyzed, which was done with the name analyzer provided to us. Next, we made a type checker to make sure that that there were no unexpected type relations. Finally, we needed a way to transpose the structure into web assembly, which was done by `codeGen.scala`.

In our extension, we wanted to add the possibility to use tuples, a widely used feature in all programming language. We had to modify all the steps but the lexer and the code generator, and add a new stage in the pipeline, as we will explain more in details in the following sections.

2. Examples

Here are some examples of tuples and how we can use them.

```
object Tuple {
  def maybeNeg(v: (Int, Boolean)): (Int, Boolean) =
  { // type
    v match {
      case (i, false) => // pattern
        (i, false) // literal
      case (i, true) =>
        (−i, false)
    }
  }
}
```

This first example illustrates pattern matching on a couple. We can see how tuples are used as types in both the parameters and the return types of the function, but also as patterns and as literals.

```
object LongTuple {
  def adder(v: (Int, Int, Int, Int, Int, Int, Int)): Int = {
    v match {
        case (v1, v2, v3, v4, v5, v6, v7) =>
            v1 + v2 + v3 + v4 + v5 + v6 + v7
    }
  }
}
```

The second example demonstrates the use of a tuple of size 7.

```
object TupleCeption {
  def adder(v: (Int, Int)): (Int, (Int, Int)) = {
    v match {
        case (v1, v2) =>
            val res : Int = v1 + v2;
            (res, (v1, v2))
    }
  }
}
```

This last example shows that our implementation supports nested tuples and allows us to use operators on their elements.

3. Implementation

We are going to explain how we implemented the tuples in Amy.

3.1 Theoretical Background

Since tuples were not a new concept, all the theoretical background we needed was already provided to us in the course of our studies at EPFL. Here is the Wikipedia definition of tuples: *"A tuple is a finite ordered list (sequence) of elements: single, couple/double, triple, quadruple, quintuple, sextuple, septuple, octuple, ..., n-tuple."* [1].

Therefore, tuples can be represented as fixed-size vectors containing different types of elements, making it possible to declare tuples of integer and boolean or even tuples of tuples as seen previously in our examples.

3.2 Implementation Details

We had to make a choice between two different implementations of tuples, either *"treat them as an built-in feature of Amy, or desugar them into Case classes"* [2]. The first way required us to modify every step of our compiler, but ensured that tuples would be handled as a complete implemented feature. On the other hand, the second necessitated fewer modifications in all the steps before code generation, but tuples wouldn't be distinguishable from case classes after this last phase, which implied that the debugging might get trickier. Moreover, we believed it was more limited in the sense that we had to define an upper limit for the size of the tuple.

We chose the latter as it was in a way easier to implement. At the beginning, we thought that it was very rare to use tuples of size bigger than 5 so we decided to support tuples of size 2 to 5. However, when we started to implement the desugarization, we realized that we could have tuples of arbitrary size. Then, it made no sense anymore to bound the size of our tuples.

3.2.1 Tree Module

The first step of our implementation of tuples was to add three new classes in the Tree Module: `TupleType` to represent types, `TupleExpr` for expressions and `TuplePattern` for patterns. Each of these three modules contains a list respectively of `TypeTree`, `Expr` and `Pattern` and is defined by its size.

3.2.2 Parser

We added `TupleType`, `TupleExpr` and `TuplePattern` support. For the first one, we wrote a new `val` for tuple types. It contains a list of types, separated by commas, between parenthesis. For the expressions, we implemented a case in the `val parenthesizedExpr`. If the list of expressions is bigger than 1, the expression becomes a `TupleExpr`. For the last one, we did the same as with types but for patterns.

3.2.3 Name Analyzer

The name analyzer transforms the types of the nominal tree into types of symbolic tree. Therefore, we added a case for `TupleType` in the `transformType` function. All types contained in the attribute of the `N.TupleType` are then mapped to their transformed type (considering their module) in the symbolic tree and wrapped into a `S.TupleType`.

3.2.4 Type Checker

In the `genConstraints` function, we added a case for `TupleExpr`. A constraint is generated for each type of the expected `TupleType`. If the expected type is not a `TupleType`, an error occurs.

3.2.5 Tuples

Finally, we decided to add a new stage to the pipeline to desugar the tuples into case classes. We first tried to insert this step in between the name analyzer and the type checker but for that, we'd have had to create a new class for each of the different types of tuples without knowing beforehand the type of the given tuples. We quickly realized that this was not the solution.

It became quite clear that the only place to insert this new stage was just after the type checker and before the code generator. That way, we did not have to worry about the type of the tuple as is was already established by the type checker. All that was left to do was traversing all the modules to change each instance of tuples into case classes.

To do so, we copied parts of the NameAnalyzer to traverse the program and while traversing we modified every instance of either, `TupleType`, `TupleExpr` or `TuplePattern` into their case class equivalents (respectively `ClassType`, `Call`, `CaseClassPattern`). As we traversed the program, we added Types and Constructors for the different sizes of Tuples we encountered, it allowed us to have Tuples of arbitrary size while keeping the output size to the minimum needed. The types are added to a new module called `$amycTuples`.

3.2.6 Difficulty

At first, it was difficult to determine where to start. With the help of the TAs, we figured out what we needed to do. The support for tuples in the four first parts was pretty straightforward. The real difficulty was in the desugarization part. We encountered many difficulties as we explained earlier.

4. Possible Extensions

4.1 Individual access

One possible extension we could add would be to support individual access to the elements of a tuple. Right now, the only way to access the elements inside of a tuple is to use a pattern match. Implementing this extension would make the use of tuples simpler for the programmer using our compiler. Indeed, the `TupleCeption` example would become more straightforward:

```
object TupleCeption2 {
  def adder(v: (Int, Int)): (Int, (Int, Int)) = {
    val res = v._1 + v._2
    (res, (v._1, v._2))
  }
}
```

4.2 Value equality

In our extension, since tuples are treated as case classes, their equality is done by reference. It would make sense to implement equality by value. For example, this piece of code would work as intended:

```
object TupleEquality {
    def equals(t1: (Int, Int), t2: (Int, Int)): Boolean = {
        t1 == t2
    }
}
```

This would return true if and only if the contents of t1 and t2 are the same.

4.3 Variable size tuples

Considering that variable arguments are a common feature of computer languages, we thought that it would be possible to do the same with tuples. Concretely, it would be possible to create single-type tuples of undefined size. Here is an example to show how it could be used:

```
object VarargsTuple {
    def subst(t: (Int...)): Int = {
        t match {
            case (v1, v2) => v1 − v2
            case (v1, v2, v3) => v1 − v2 − v3
            case _ => 0
        }
    }
}
```

However, it would be more practical to implement it with the individual access, so we would not need to always use pattern matching to work with `VarTuples`.

References

[1] Wikipedia. Tuple, 2020.

[2] Georg Schmid. Compiler Extensions for Amy, 2020.