View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Scala AST Persistence

Technical Report

Mathieu Demarne Adrien Ghosn
Eugene Burmako
EPFL, Switzerland
{firstname.lastname}@epfl.ch

Abstract

The Scala compiler uses ASTs (abstract syntax trees) as an intermediate representation before generating bytecode. With the development of Scala macros which expand trees at compile time, being able to access, modify and recompose ASTs within the compilation scope is becoming more and more important.

One of the common scenarios of using macros is inspecting abstract syntax trees within reach in order to learn more about the code being transformed, to apply more powerful optimizations, etc. However, arguments to macros can depend on third-party libraries, which are precompiled as bytecode and don't have their ASTs available. It would therefore be great to have a way to publish ASTs along with the bytecode. The publishing of those ASTs should be a choice of the programmer and should take as little space as possible in order to be transparent to the user.

The AST persistence compiler plugin has been developed to address this problem by intercepting ASTs produced by the typechecker, compressing them with a dedicated tree compression algorithm and storing the result along with the generated bytecode. An accompanying library is provided that can retrieve stored trees for accesses in macros or even elsewhere. Finally, the AST persistence SBT plugin makes this experience as smooth as possible by automating packaging, distribution and fetching of stored trees.

In order to persist abstract syntax trees, one needs to store their structure (types of AST nodes and the edges between them) along with additional metadata, such as names, symbols and types. In the report we present a compression algorithm that targets just the structure of trees and an algorithm to store arbitrary data alongside. We have implemented and benchmarked these mechanisms to persist trees and names leaving support of other kinds of metadata to future work.

Keywords: Scala, tree compression, compile time reflection

Main repository: github.com/scalareflect/persistence

Table of Contents

Abstract....1
1 Introduction....3
1.1 Motivation....3
1.2 The challenge of compressing ASTs....3
2 Compression Algorithm....3
2.1 Overview....3
2.2 Pseudocode....4
2.3 Compression Example....5
2.4 Most Frequent Patterns....6
3 Implementation....6
3.1 Overview....6
3.2 Internal Data Representation....7
3.3 Compiler Plugin Structure....8
3.4 Decompression Library Structure....11
3.5 Dedicated SBT Tasks....12
4 Testing throughout Development....13
5 Benchmarks....14
5.1 On Sizes....14
5.2 Compilation Time Results....16
5.3 Time Break Done....16
5.4 Jar Example....16
6 Future Work....19
6.1 Storing Type Hierarchy, Symbols and Constants....19
6.2 Storing Compressed ASTs in .class File at Compile Time....19
6.3 Scalability and Going Further....19
7 Conclusion....19

1 Introduction

1.1 Motivation

With the development of Scala macros[1] and the use of Scala reflection APIs[2], abstract syntax trees often need to be constructed. For example, reify and more recently quasiquotes[3] already provide suitable abstraction to rebuild trees at compile-time. But what if a code rebuilt by a quasiquote rely on a third-party library, with inaccessible sources? The goal of AST persistence is to provide, among others, a solution to this problem: being able to package ASTs in a compressed, minimalistic way, which will allow, through a reflection library, to get and modify specific parts of trees.

Moreover, storing ASTs provides other interesting outcomes, even outside of the scope of Scala macros. They could for instance be used in debuggers to evaluate specific expressions and add debugging commands inside the trees or allow run time macro expansion.

1.2 The challenge of compressing ASTs

Even though storing ASTs is useful, we need to mind the associated overhead since space consumption might also be important to the user. A natural choice here is to go for compression.

ASTs contain important informations such as types, symbols and names. Types and symbols don't rely on a redundant pattern as they have, respectively, parents and children, or owners. Hence they are more or less incompressible based on a structural approach.

As mentioned in the abstract, we view AST persistence as a combination of two problems: 1) storing the structure of trees and 2) storing the metadata. Here we focus only on the first problem, considering the second one in later sections of the paper.

An interesting thing about considering just the tree structure in isolation is that it features unique styles of redundancy. As will be shown later, such patterns can be categorized in a dictionary which can be used to simplify the trees and allow compression.

This problem was the object of research in the past. For instance Slim Binaries[4] use a tree-based representation of programs in order to allow better cross-platform portability and run time optimization, while drastically reducing the size of the compiled code.

It is not obvious that in Scala, which is a functional language and therefore concise, the amount of redundancy of the code is going to be considerable. Fortunately, even if source code tends to be not very redundant, ASTs use a lot of common patterns, which can easily be compressed. For instance, our experiments have shown that some simple sequences are frequently repeated, such as:

```
TypeApply(Select(Ident), TypeTree)
```

Extracting, compressing and storing ASTs at compile time could therefore be a viable solution, as the size required to store them might be reasonably small compared to bytecode and source code. ASTs could moreover be packaged as artifacts and published along with binary files.

This was our initial hypothesis when we started the project. Follow along to discover where we ended up!

2 Compression Algorithm

2.1 Overview

The compression algorithm presented in this paper is a variant of Lempel-Ziv-Welch[5] and consists in generating a dictionary of subtrees which will then be used to transform the tree into a list of occurrences and joining edges. It is mainly inspired by the paper

Efficient Lossless Compression of Trees and Graphs by Shefeng Chen and John H. Reif from Duke University[6], which proposes a compression algorithm for homogeneous, binary trees only. However the algorithm developed here for Scala ASTs differs greatly, as it considers typed nodes with unfixed and unbounded number of children.

2.2 Pseudocode

The AST compression is done in two main phases: first, the dictionary generation, which produces all potential subtree sequences to be used as parts of the compression dictionary; then, once the dictionary is generated, the tree is reparsed in proper sets of entry occurrences and joining edges.

2.2.1 Dictionary Generation

The first phase goes through the tree and generates a dictionary of matching subtrees while keeping the frequencies of their occurrences.

```
Initialisation (dictionary D, queue S):

      D contains only the root as
      subtree, with frequence 0.

      S contains the root of the tree.

Loop (until S is empty):

      Let N be the head of the queue S;
      remove N from S.

      Starting from N, find the maximal
      matching subtree match(N) in the
      entry of D. Let's call the subtree
      below N (subtree with N as root node)
      subtree(N).

      If subtree(N) = match(N), then all
      the subtree is covered.

            Update the frequency of
            match(N) in D (add 1 to the
            previous value).

      If |subtree(N)| > |match(N)|, then
      only  a subpart of subtree(N) is
      covered by match(N).

            Add to match(N) the first
            node in BFS order L in
            subtree(N) not covered by it.
            Call this new match match'(N).

            Update the frequencies of
            all matching entries found
            in D, and add match'(N) to
            D. Also add L to D if not
            already in it, and update
            its frequency by adding 1 to
            it.

            Remove all nodes covered by
            match'(N) in subtree(N), and
            add the roots of each new
            subtree to S.
```

2.2.2 Encoding

The encoding phase sorts the dictionary in order to use the more common entries.

To have a valid compression the algorithm however first rejects all the entries with size bigger than the square root of n, where n is the size of the whole tree to encode. This is a heuristic proposed by our reference paper which intuitively makes sense. If a tree has more than the square root of the number of nodes, then expanding each of its nodes might create a representation of the tree bigger than the original one, which should not happen.

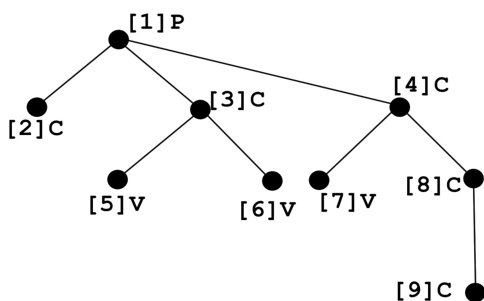
The entries inside the dictionary are then sorted based on their frequencies and sizes.

```
Initialisation (Dictionary D, queue S):

      Remove entries in D following the
      heuristic presented above.

      For each entry E in D with
      frequency f, compute K = |E|·f.

      Sort these entries based on K in
      decreasing order. In case of
      equality, the bigger subtree will be
      first. Add a new frequency count,
      initialized to 0.

      Let S contains the root of the tree.

      Let B be an output buffer for the
      encoded tree, initially empty.

      Let L be an output buffer for the
      edges between each encoded
      subtrees.

Loop (until S is empty):

      Let N be the head of the queue S;
      remove N from S.
```

```
        Find all matching entries in D for
        the subtree starting with root N.
        Keep only the one with the biggest
        K; let's call this one match(N).

        Remove all nodes covered by
        match(N) in subtree(N), and add the
        roots of each new subtree to S.

        Add in L the indexes (calculated in
        BFS order) of the nodes joining the
        roots added to S, in order to store
        the edges between the subtrees stored
        separately.

        Increase the frequency count for
        match(N) in D by one.

        Add the identifier of match(N) to B.

Finalization:

        Remove all unused entries in D.

        Compute Huffman code based on the
        new frequencies computed in the
        previous loop.

              This frequency might be
              different than the one
              computed in the first phase
              of the compression.

        Replace the identifiers in B by the
        Huffman codes, let's call this new
        output buffer B'.

        Output D (subtrees and Huffman
        codes only), B', and L.
```

2.3 Compression Example

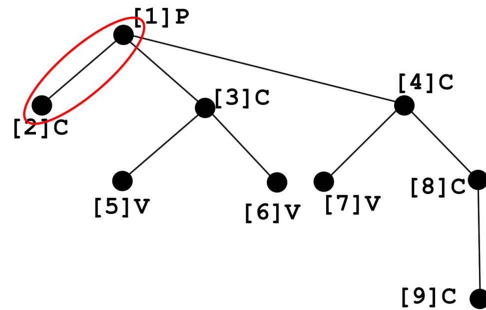
Below is a theoretical example to illustrate the algorithm. Note first that the tree, for simplification reasons, is not a valid Scala AST. The nodes are however typed (hence the letter tag for each node) and the degree of each node is not bounded nor fixed.



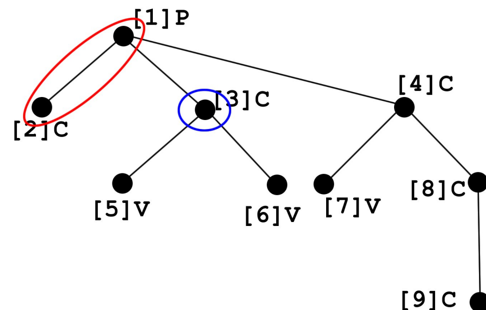
2.3.1 Dictionary Generation

Initially only `[1]P` is in `S` and `D`.

Step 1: the root `[1]P` is popped from `S` (i.e. `N = [1]P`). As `P` was inserted in `D`, we have a matching subtree `match(N)` of `P` only. Since the subtree below `[1]P` is bigger than `match(N)`, which has size one, we insert the next node of the subtree starting at `[1]P`, which is `[2]C` according to the BFS ordering. Therefore we have: `D = {P → 1, PC → 1}` and we add the roots of the subtrees not covered by `match(P)` in `S`: `S = {[3]C, [4]C}`.



Step 2: this time, we pop `N = [3]C`. Since `N` has no matching entry in `D`, we simply add it to `D`. Therefore we have that `D = {P → 1, PC → 1, C → 1}`. Moreover we add the root of the subtree immediately below `[3]C` to `S`: `S = {[4]C, [5]V, [6]V}`.

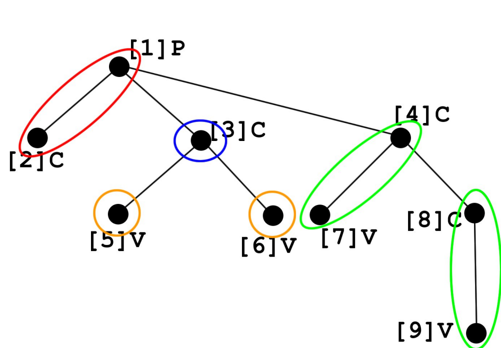


Step 3: `[4]C` is popped from `S` and the algorithm is repeated again until nothing remains in `S`.

The dictionary generated along with the frequencies is therefore as follow:

```
P => f = 1
PC => f = 1
C => f = 2
CV => f = 2
V => f = 2
```

Those frequencies will then be used in the encoding as explained in next section.



2.3.2 Encoding

The algorithm removes the entries in the dictionary that are too big, i.e. more than $\sqrt{n}$ where n is the size of the tree. Here, $\sqrt{n} = \sqrt{9} = 3$. Nothing has therefore to be removed.

The algorithm now computes `K = |E|·f`, for each entry `E` in , `D` and sorts `D` based on `K`:

```
CV, K = 2·2 = 4
PC, K = 1·2 = 2
C, K = 2·1 = 2
V, K = 2·1 = 2
P, K = 1·1 = 1
```

Based on that, the algorithm finds the following sequence of matching subtrees with maximal frequencies:

```
PC / CV / CV / V / CV
```

The joining edges are:

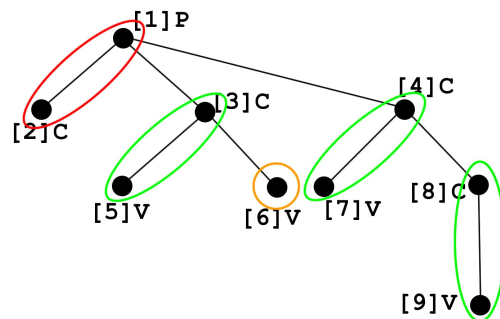
```
L = {1,1,3,8}
```

The Huffman codes based on the frequencies of occurrences are as follow:

```
PC  frequency: 1       code: 01
CV  frequency: 3       code: 1
V   frequency: 1       code: 00
```

The compressed tree is therefore `0111001`, and we output it along with `L = {1,1,3,8}` and the reversed dictionary:

```
1  => CV
01 => PC
00 => V
```



2.4 Most Frequent Patterns

Below are the most frequent patterns for the file `Typers.scala`[7] from the Scala compiler, along with their appearance frequencies.

```
(Ident, Select) → 1664

(Ident, Select, Select) → 892

(TypeTree, ValDef) → 810

(Ident, TypeTree, Select, TypeApply) → 230

(This, Select, TypeTree, Select, TypeApply)
→ 220

(EmptyTree, TypeTree, Apply, ValDef,
Function) → 169

(Ident, Typed, TypeTree, Ident, Ident,
Apply, ValDef, TypeTree, Select, Apply,
Block, TypeApply, If, LabelDef) → 139

(Ident, Select, TypeTree, Select,
TypeApply) → 83
```

3 Implementation

3.1 Overview

The most information about a Scala program is available in the Scala compiler right after typechecking. Before typechecking, there is no information about symbols and types, and after typechecking trees are progressively simplified in order to optimize bytecode generation, which loses information about their original shapes. Thus the best way to persist trees is to add a phase to the Scala compiler that comes right after the typer phase. A suitable way to do this is to implement a plugin for the Scala compiler.

As a consequence, the new phase of the compiler

changes the representation of the Scala trees prior to perform the compression, by storing separately names, types, symbols and other metadata. Those informations are removed from the ASTs, producing a simplified version of them. The ASTs are then compressed using the algorithm presented above (at tree-level), and then compressed again at byte-level using an XZ library[8] built in Java.

Once compressed, the ASTs as well as the other required informations are put into an `asts/` folder right near the compiled classes and respect the packaging hierarchy.

Accompanying the plugin we have implemented a decompression library. It provides various methods to access specific parts of the trees, such as methods or classes declarations.

Moreover, the plugin and the packaging of the ASTs should be as transparent as possible to the user. New packaging and publishing tasks were added to SBT and abstracted behind an SBT plugin, as well as the use of the compiler plugin.

3.2 Internal Data Representation

Unfortunately manipulating trees isn't always easy. There is no way to keep a parent and a child close together in a list while preserving the ordering. The algorithm builds a new intermediate representation by flattening all the children contained in an AST into a single, simple list, which can then be accessed uniformly without having to take the type of the node into consideration.

In order to easily manipulate those new nodes, the algorithm also heavily relies on a BFS representation of the whole tree, with children first. This ordering is well suited since it becomes relatively easy to cut a tree at a specified position without generating orphan children. Moreover, the compression algorithm builds a dictionary of entries which are monotonically increasing and follow a BFS ordering. This representation was therefore even more indicated. Furthermore, our algorithm needs to traverse the tree in a uniform way both for compression and decompression. As a consequence, we had to find a new representation for the ASTs that fits our needs and simplified the implementation of the algorithm.

More specifically, we only need to store a subset of information contained in the original ASTs, namely, the type of a node, its name if any, and its children.

3.2.1 Representing Nodes

Since we only need to remember the type of a node and its children, we created a case class named `Node` with the following signature:

```
case class Node(
      tpe: NodeTag.value,
      children: List[Node])
```

where `NodeTag` is an enumeration of the types that exist in the Scala AST's, and children is simply the list of nodes that have this one for parent. With this representation we can now uniformly access and traverse the nodes of the tree without worrying about its type and, therefore, its attributes.

3.2.2 Representing Nodes in BFS Order

We said before that our algorithm uses a breadth first search traversal of the tree. As a consequence, we created another case class that wraps the `Node`'s representation into what we called a `NodeBFS`:

```
case class NodeBFS(
      node: Node,
      bfsIdx: Int,
      parentBfsIdx: Int)
```

where the `bfsIdx` is the index of the node in BFS order, starting at the root of the tree, `parentBfsIdx` is the index in the same representation of the node's parent (if the node is at the root, then it is -1). We usually use this representation when we flatten the tree into a list that respects the BFS order. By doing this, we

are able to use the functions defined for Lists.

3.2.3 Representing Names

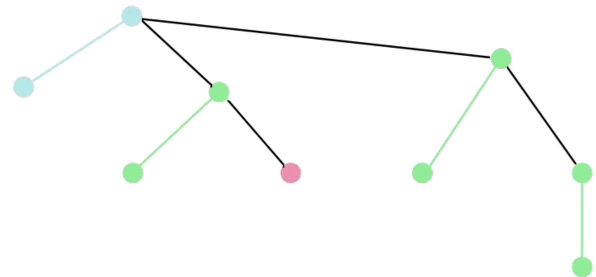
As you can see, names for types and terms are not part of this representation. We removed them and put them into a separated List. This list respects the order of appearance in the tree. We then transform it into a `Map[String, List[Int]]` where the list corresponds to BFS indexes of the nodes where they appear. This representation avoids repeating duplicates and moreover allows a better compression due to its concise representation. This enables an efficient search through the tree represented as a BFS list of nodes when we need to find the definition of some element, since we don't have to go through the whole list (see high level compression on names for an example).

3.2.4 Representing Edges

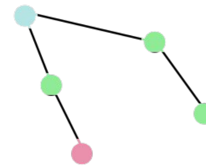
Representing edges was a challenging part. In the algorithm presented above an edge linking a subtree to its parent is stored using the BFS index of the parent of its root node. However such index must be known at decompression, which is impossible since not all the tree is rebuilt. To address this issue, we decided to represent the edges as a list of pairs of integers. The first integer is the parent's subtree index, that is, it is the index of the subtree corresponding to an entry in the dictionary, in BFS order, that contains the node to which the subtree we are considering is connected. The second integer is the BFS index, relative to the subtree, of the parent node. We only need the inter-subtrees edges to be able to reconstruct the original tree (since the other edges are part of the entries in the dictionary). Another way to see this, is that we transform the original tree by merging nodes that form a subtree contained in the dictionary and therefore create a much simpler tree for which we need to store the edges.

Example

Let's take back our tree used as a compression example. Here subtrees corresponding to entries in the dictionary are colored. Edges in black are inter-subtree edges:



Then, we merge all nodes belonging to a single subtree into one node, but we keep the inter-subtree edges:



Each color corresponds to a Huffman code. To be able to get back the original tree, we simply need to know how to relink the nodes that we have here. Once this is done, we can expand each node to the original tree that it corresponds to.

3.2.5 Huffman Dictionary

The dictionary is used to encode the tree and is a specific type `HuffDict`, which is simply a map from a list of `NodeBFS`, representing a subtree, to a list of bytes, which is the Huffman code corresponding to this entry. We chose to generate Huffman codes in order to both save space and get rid of any possible ambiguity in the decompression phase. At this point, each byte in the list is either 0 or 1, but when we compress the dictionary we transform them into bits (hence the need for an unambiguous representation encoding).

3.3 Compiler Plugin Structure

For the compression, the `Plugin.scala` class has an

apply method that is responsible for correctly generating the compressed representation of the AST and writing it to a file.

We can break the compression into two phases: the encoding of the AST and Names from the internal data representation to an array of bytes (“high-level compression”), and the use of the XZ library to compress those bytes (“low-level compression”).

The first phase produces a list of bytes that are to be compressed in the second phase.

In order to correctly read the elements, due to the fact that we do not have a uniform encoding, we needed to find a way to efficiently detect when the encoding of one element ends, and when another begins. Furthermore, as we will see later, our implementation generates bytes equal to -1 which prevents us from using most of the tools that I/O libraries provide to detect the end of the input.

3.3.1 Transforming the AST Representation

The Scala ASTs are first transformed into our own internal data representation by a tail recursive call in a specific class called `TreeDecomposer.scala`. The tree decomposer moreover stores the names in a separated list. Those two kinds of instances are then used by the tree compression as well as the name compression, respectively.

3.3.2 High-level Compression (on ASTs)

The class `AstCompressor.scala` contains the concrete implementation responsible for encoding the original tree.

It performs the compression of the AST in its apply method. This class contains different methods to encode each and every element that we decide to store. We will present them and the technique to encode them in the same order they are processed in the apply method.

The Occurrences

We begin by saving the occurrences, that is, the series of Huffman codes representing the tree flattened in BFS order and encoded with our dictionary. This is done in a very straightforward way, we first generate two bytes corresponding to the size in bytes of the occurrences. Then we simply write the occurrences. Of course, in order to save as much space as possible, all bits representing this encoding are concatenated and grouped into bytes. Since we use a Huffman code, we know that we can unambiguously recover the original list of entries by using the longest prefix match rule.

The Edges

The next element to be saved are the edges.

As we said, this part was quite challenging, not only because we needed to adapt the original algorithm to be able to handle nodes with variable number of children, but also in terms of space required to store this information. As a matter of fact, edges are containing the whole structure of the tree which, obviously, represents a lot of information. The first version of our algorithm was not efficient enough simply due to the fact that edges took too much space. Hopefully, we found a way to reduce the size needed to store them.

To encode them, we decided to break the list of pairs of integers into two lists of integers.

In fact, we noticed that the indexes of the parent tree repeat themselves a lot and are ordered. Therefore, we were able to delete this redundancy by encoding a parent’s entry as the index, followed by the number of time it repeats itself. We also took advantage of the fact that indexes are close to each other. Therefore, instead of encoding all of them as integers (four bytes), we decided to store a short (two bytes) that represents

the difference between this element and the precedent one, therefore reducing greatly the average length of the data needed to encode the edges.

The Dictionary

The next phase writes the dictionary. For that purpose we first write the number of entries in the dictionary as an integer (encoded on four bytes). Then, for each entry, we write the size of the list of bytes representing the Huffman code corresponding to the list of `NodeBFS` defining the entry. We then transform the list of `NodeBFS` into a simpler representation, where each element is encoded as a triplet (Byte, Short, Short), where the byte encodes the type (the `tpe` attribute), the second element is the BFS index of the node and the third the parent's index.

3.3.3 High-level Compression (on Names)

Names are encoded to bytes using the `NameCompressor.scala` class. For each name, we generate a list of the BFS indices where it appears, therefore creating a map from a string to a list of occurrences. The first occurrence corresponds to the definition of the element. We did this hack in order to be able to quickly reconstruct only a subtree of the original tree. When the user specifies a class, object, value or method definition that he wants to reconstruct, we simply feed it as input to the map containing the names as keys, look for the definition bfs index, extract the subtree from the list of `NodeBFS` and recompose only the part that we want. This is explained in more details in the next chapter.

Storing the map of names was however done using a different representation to allow a better compression at low-level. The mapping `[String, List[Int]]` is transformed as follow. First, the strings of the names are sorted and stored with a simple separator of one byte. All of them then receive a specific Id. Those ids are then used to generate a list of consecutive occurrences of names, each position specifying in BFS order the id of the name of the next named node (not the BFS indices directly). For example, if we have the following dictionary:

- `a → 1, 4, 5`
- `b → 2, 3`
- `c → 6, 7`

… with corresponding ids for keys:

- `a → 0, b → 1, c → 2`

… then the encoding would be:

```
"a\nb\nc\n0110022"
```

Using this representation, repeating patterns in the sequence of names can then be more efficiently compressed by the low-level compression.

3.3.4 Low-level Compression

Now that we have all the bytes corresponding to each element, all we need to do is to feed them to the XZ library in order to apply the LZMA[9] compression algorithm. This is done by `XZWriter.scala`. Using this library, we had to face an important problem. The library provides, for decompression, an `available` method that estimates the number of bytes that we can still read from the input source. This method's implementation is based on the assumption that the byte -1 is reserved and used to mark the end of the input. But we sometimes generate -1 as a byte encrypting one of our elements. Therefore, in order to be able to correctly decompress the file, we had to perform the following trick: we first write the total number of bytes that corresponds to the whole compression, and then write the compression to the file using the XZ library. Using this method, we know exactly how many bytes we are supposed to read and we are able to stop the decompression at the exact correct moment.

3.4 Decompression Library Structure

3.4.1 Low-level Decompression

The `XZReader.scala` class is responsible for decompressing the bytes corresponding to our encoding. It begins by reading a long (four bytes) corresponding to the total number of bytes that are supposed to be decoded by the XZ library. We then decompress this number of bytes.

3.4.2 High-level Decompression (ASTs)

The `AstDecompressor.scala` class is the symmetric of `AstCompressor.scala` that we presented before and therefore we will not explain in details how it works. All elements are regenerated from the list of bytes we got from the XZReader in the precedent step in the obvious way.

3.4.3 High-level Decompression (Names)

The `NameDecomposer.scala` class is responsible for getting back the names and their occurrences.

Due to their representation, names are not stored along with their BFS indices in the tree, but are only ordered in BFS. Therefore those indices need to be reconstructed by traversing the tree during decompression. The result is a map from String to a list of integers corresponding to the BFS indexes of the nodes in which the names appear.

3.4.4 Recomposing the Tree

The `TreeRecomposer.scala` class is responsible for reconstructing the Scala AST. In other words, it translates our representation, which uses `Node` and `NodeBFS`, into the original structure used in the Scala compiler to represent syntax trees.

For that, it traverses the tree node by node and rebuilds the corresponding AST element using the universe object that it received in its constructor. This is done in a straightforward way, using a match on the `tpe` attribute of the node.

This implementation had some challenges. First, since we extracted the names from the trees, we needed to find an efficient way to put them back. Our solution was greatly influenced by the fact that we expected a toolbox (see next section) to be used to get back only parts of the original tree. Thus, only some parts of the tree needed to be reconstructed.

3.4.5 The "Toolbox"

The Toolbox (`ToolBox.scala`) is designed to be the interface between the user and the decompression library. The interface proposes a method to get classes, objects methods or value definitions corresponding to some symbol specified as input:

```
def getSource(symbol: Symbol): Tree
```

This one could moreover be easily wrapped in an implicit class to be called by a one-liner such as:

```
symbol.source
```

Our goal is to be as efficient as possible and avoid unnecessary computations. This greatly influenced the overall implementation of our compiler plugin as well as our decompression library as our internal representation had to enable us to reconstruct specifically selected parts of the original AST.

To answer the user's request, we first fetch the file corresponding to the path obtained from the symbol. This file has to be in a jar contained in the classpath. We extract the file, decompress it and then search for the part that we need to answer the request correctly. We take care to save the triplet formed by the tree represented as a Node, the BFS representation of it (which is expensive to compute) and the map for names occurrences.

If a request then comes for the same file, we do not have to parse anything again and can directly look into

the map.

One limitation of our implementation is that the smallest unit that we can read is a file. In fact, for any request we need to read at least one entire file, even if we will not use every element that we get from it.

This problem can be addressed with some modifications in the future. In fact, we could imagine storing the names first in the file, then the other needed information with the occurrences last. Since we have the size for each of them, reading the map for names would give us the BFS index of the part that we need to read from the occurrences, and we could therefore jump to the correct place, that is, we ask XZ to discard bytes that we are not interested in (we do not try to recompose anything from them). But this is quite complicated and XZ works sequentially. So we cannot ask it to process byte number 4 before byte number 3 for example, which forces us to read everything that precedes what we are interesting in, in the file. We did not try to implement this solution for the moment.

Once the file is decompressed, we have the elements corresponding to the tree, as described in the precedent part.

We then lookup for the name corresponding to the user's request in the map containing the names occurrences. We take the BFS index corresponding to the definition of this element and check that the node in the BFS tree has the type corresponding to the user's request (class, method, or value definition). If it does not, we look at the next index in the occurrences. We repeat the process until we find the correct `NodeBFS`. Then, we extract the subtree from the list of `NodeBFS` corresponding to the element's definition. This can be done efficiently using our `NodeBFS` representation, since it allows tree manipulation by removing elements from the BFS list.

Finally we feed this to the `TreeRecomposer` which will produce the AST that we can return to answer the user's request.

For the moment, we did not implemented the code to handle the cases of overloading or multiple definitions of same type (i.e. same Scala AST type, with the same name). But we will easily add some function that will take a more detailed path, for example by full names, and would correctly identify the subtree to reconstruct, hence providing the user with a way to select the definition that he is interested in.

Example

```
class C{}
object X {
    class C {}
}
```

Where the enclosed C in X could be accessed by :

```
symbolOf[X.C].source
```

3.5 Dedicated SBT Tasks

3.5.1 Overview

Creating, packaging and publishing the compressed AST should be as much transparent to the user as possible. A way to ensure this property is to create an SBT plugin which will automatically add the compilation settings as well as creating the required commands for publishing the new artifacts.

More specifically, our SBT plugin provides the following features:

- It adds by default the compiler plugin and ensures consistency, even if compilation fails, by keeping into a backup the previously compressed ASTs and restoring them if needed. This is done by overriding the main compile task as well as defining a task executed prior to any compilation.
- It adds a `packageAst` task which creates a

new jar with a specific AST classifier ready to be published.

- The `package` task is overriden in order to execute `packageAst` along with the `packageBin` task as well.
- The packaged ASTs are also added as an artifact to be published.

The SBT plugin is implemented as a `AstcPlugin.scala` file, extending the default SBT plugin class. We decided not to use the new `AutoPlugin` from SBT 0.13.5[10], to avoid errors in overriding settings of older SBT versions.

3.5.2 Example

Assuming we have a simple project on which we would like to use our SBT plugin. In the `plugins.sbt` file, we can simply specify :

```
addSbtPlugin("org.scalareflect" %
"persistence-sbt" % "0.1.0-SNAPSHOT")
```

This will automatically add the compiler plugin as well as the features required to manipulate ASTs files. Below is a simplified output using SBT:

```
> packageAst
Updating {file:/tests/}tests... Resolving
jline#jline;2.11 ...
Done updating.
Compiling 1 Scala source to ...
Packaging .../tests_1-asts.jar ...
Done packaging.
[success] Total time: ...
> publishLocal
Packaging
      .../tests_2.11-1-sources.jar ...
Done packaging.
Packaging .../tests_1-asts.jar ...
Main Scala API documentation to ...
Done packaging.
Wrote .../tests_2.11-1.pom
:: delivering :: ...
delivering ivy file to ...
Packaging .../tests_2.11-1.jar ...
Done packaging.
model contains 2 documentable templates
Main Scala API documentation successful.
Packaging
      .../tests_2.11-1-javadoc.jar ...
Done packaging.
published tests_2.11 to
      .../tests_2.11.jar
published tests_2.11 to
      .../tests_2.11-javadoc.jar
published tests_2.11 to
      .../tests_2.11-asts.jar
published tests_2.11 to
      .../tests_2.11-sources.jar
published tests_2.11 to
      .../tests_2.11.pom
published ivy to .../ivy.xml
[success] Total time: ...
```

Following those commands, the ASTs are already available in the local repository. Of course, the added settings also allow to publish in non-local repositories.

To fetch published ASTs for dependencies, we can then specify needing them in the build, as follow:

```
libraryDependencies += "your.org" % "tests"
% "0.10" classifier "asts"
```

This could also be done transitively by specifying which classifiers to fetch for all the dependencies:

```
transitiveClassifiers := Seq("asts")
```

Those settings are standard options in SBT[11].

The decompression library will then automatically fetch the ASTs, since they are available throughout the classpath once the classifiers fetched.

We first thought of adding those settings automatically using our SBT plugin, but unfortunately there is no way to know exactly for which dependencies the project would require the ASTs without knowing in advanced its definition. Using the lines specified above, the user can specify the classifiers he needs easily.

4 Testing throughout Development

Since we are experimenting with new algorithms for which we only had a theoretical support, and no reference implementation, we needed a way to check the correctness of our implementation step by step. Again, the overall architecture / organization we decided to adopt for this project helped us. As you

have seen, each step of the compression is implemented separately and has enough independence to be tested on its own.

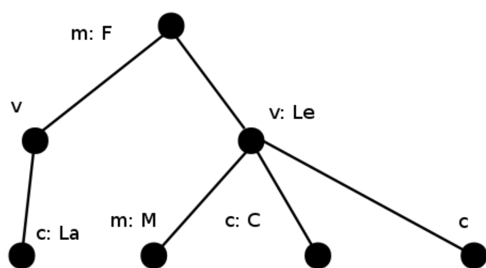
One major difficulty we had to address was to generate test cases on demand, of variable complexities without requiring the plugin to be executed on a real Scala file.

We decided to use *ScalaTest's funSuite*[12] in order to ensure that our code had the required properties. Furthermore we created two parsers that generate trees from simple strings in order to be able to quickly and easily generate test cases. The semantic to write trees is as follows:

```
def tpe: Parser[NodeTag.Value] = (
  ident ^^ {case e => e}
)

def NodeParse: Parser[NodeName] = (
  tpe ~("!"~> ident <~"!").? ^^ {
    case e1 ~ Some(e2) => NodeName(e1, e2, Nil)
    case e1~None => NodeName(e1, emptyName, Nil)
  }
)

def TreeParse: Parser[NodeName] = (
  NodeParse ~ ("(" ~> rep(TreeParse) <~")").? ^^ {
    case e1 ~ None => e1
    case e1 ~Some(e2) =>
      e1.addChildren(e2)
      e1
  })
```

… where `tpe` is the type of the node, encoded as one character and the identifier between "!" is the name of the node.

So for example, one tree corresponding to the following schema:



... would be represented as:

```
"m !F! (v (c !La! v !Le! (m !M! c !C! c)))"
```

This proved very useful for debugging.

5 Benchmarks

The compression algorithm basically has the same compression properties as a classic Lempel-Ziv-Welch algorithm. The more redundant the tree is, the better the compression. We didn't know at first how redundant the Scala ASTs would be in average and therefore didn't know how our algorithm would perform.

In order to test if the algorithm was working well, we create two benchmarks, respectively on compression ratio and compilation time. We decided to compile for our test the whole Scala standard library (`scala-library.jar`), which proposed approximately 500 different source files of various sizes. We also added an extreme case by compiling the `Typers.scala` file from the Scala compiler (`scala-compiler.jar`), which contains more than five thousands lines of code.

5.1 On Sizes

5.1.1 Comparison Source

In order to compare our compression ratio with an industrial algorithm we create another, small, compiler plugin to extract the raw printout of the trees. In order to have a comparison as correct as possible, our benchmark removes all metadata from trees except names for the tests including them. Below is an example of simplified ASTs only (no names):

```
PackageDef(Ident(<empty>),
List(ModuleDef(Modifiers(), Test,
Template(List(TypeTree(),
TypeTree().setOriginal(Select(Ident(scala),
scala.App))), noSelfType,
List(DefDef(Modifiers(),
termNames.CONSTRUCTOR, List(),
List(List()), TypeTree(),
Block(List(Apply(Select(Super(This(TypeName
("Test")), typeNames.EMPTY),
termNames.CONSTRUCTOR), List())),
Literal(Constant(())))),
ValDef(Modifiers(PRIVATE | LOCAL),
TermName("a "), TypeTree(),
```

```
Literal(Constant(4))),
DefDef(Modifiers(METHOD | STABLE |
ACCESSOR), TermName("a"), List(), List(),
TypeTree(), Select(This(TypeName("Test")),
TermName("a "))), ValDef(Modifiers(PRIVATE
| LOCAL), TermName("b "), TypeTree(),
Literal(Constant(33))), …
```

... is encoded as :

```
a(G, S(R(, x, i(S(Q(), Q()T(F(G(x),x))), ,
S(e(, , S(), S(S()), Q(), j(S(C(F(R(E((x)),
), ), S())), I((())))), d(, (x), Q(),
I((4))), e(, (x), S(), S(), Q(), F(E((x)),
(x))), d(, (x), Q(), I((33))), …
```

... and then compressed using LZMA (Lempel–Ziv–Markov chain algorithm) and compared to our algorithm. Of course, for fairness of comparison, we compress only one file at a time with LZMA.

5.1.2 Results without Names

We achieved to have a better compression in average than LZMA, with some extreme cases where our algorithm performed twice as well. When our algorithm obtained worse results than LZMA, both compression ratio were close.

```
Global statistics

Sources: 2419914, Raw: 3544620, xz: 346008,
astc: 285232

In general, our compression is smaller than
a classic xz of .82435

Tests where xz was better: 61 over 516
tests

In comparison with the sources, our
compression is 8.48402 times smaller.
```

For detailed results, see:

github.com/scalareflect/persistence/blob/master/benchmark/results/SizeBenchmarkNoNames.txt

5.1.3 Results without Names and no showRaw Preprocessing

When we do not help LZMA by preprocessing the output of `showRaw`, we get the following result, which is really good:

```
Global statistics

Sources: 2419914, Raw: 9577264, xz: 721000,
astc: 285232

In general, our compression is smaller than
a classic xz of .39560

Tests where xz was better: 0 over 516 tests

In comparison with the sources, our
compression is 8.48402 times smaller.
```

For detailed results, see:

github.com/scalareflect/persistence/blob/master/benchmark/results/SizeBenchmarkNoNamesNoModifShowRaw.txt

5.1.4 Results with Names

In order to test our implementation with compressing names as well as ASTs, we modified our simplification of `showRaw` as presented above to preserve names.

In average, we also obtained better results than the industrial LZMA. This time, the results of two techniques were closer due to the fact that our implementation targets the compression of trees, therefore forcing names to be stored separately.

```
Global statistics

Sources: 2419914, Raw: 4595521, xz: 548996,
astc: 517684

In general, our compression is smaller than
a classic xz of .94296

Tests where xz was better: 176 over 516
tests

In comparison with the sources, our
compression is 4.67450 times smaller.
```

For detailed results, see:

github.com/scalareflect/persistence/blob/master/benchmark/results/SizeBenchmarkWithNames.txt

5.1.5 Results with Names and no showRaw Preprocessing

Again, we tried to compare our algorithm without helping LZMA with the output of showRaw. This time,

the results were as follow:

```
Global statistics

Sources: 2419914, Raw: 9577264, xz: 721000,
astc: 517684

In general, our compression is smaller than
a classic xz of .71800

Tests where xz was better: 2 over 516 tests

In comparison with the sources, our
compression is 4.67450 times smaller.
```

For detailed results, see:

github.com/scalareflect/persistence/blob/master/benchmark/results/SizeBenchmarkWithNamesNoModifShowRaw.txt

5.2 Compilation Time Results

The algorithm involves a lot of manipulations on trees under the form of lists in BFS order, and a couple of construction / deconstruction from those lists to actual tree representation.

The lookup into the dictionary of subtrees during the encoding phase in the worst case is in $O(n \cdot m)$, where n is the average subtree size and m the number of dictionary entries, since we have to compare each node inside the BFS trees step by step.

By changing a bit our implementation we were able to drastically reduce the time needed for the compression. Our algorithm still increases the time required by the compilation of 15% in average:

```
Total time with astc: 2893.549

Total time normally: 2541.068

In average, the time is increased of
(ratio): .13871

Results on an Intel I5 @ 2.67 Gz
```

For detailed results, see :

github.com/scalareflect/persistence/blob/master/benchmark/results/TimeBenchmarkNoNames.txt

The reader should be aware of the fact that, even if having good time performance was important, we did not put any specific effort into optimizing it. That means that we did not run any profiler tool on the code, and therefore, it is highly likely that some optimization can be performed in order to improve those results. Furthermore, we do not provide any comparison of time with the LZMA algorithm, since our plugin has compilation overhead and runs in the JVM, which is not the case of the industrial LZMA implementation we used, provided by 7zip.

5.3 Time Break Done

The main overhead of the algorithm comes from the transformation of Scala ASTs to our internal representation:

```
#Plugin part ~~~~~~~~~~ Time (seconds)

Start of the plugin      0
Parsed into Nodes        1.02
End of ComputeFreqs      0.21
End of SplitTree         0.22
End of Huffman gen       0.0
End of encodeOccs        0.52
End of outputOccs        0.0
End of outputEdges       0.32
End of outputDic         0.02
End of plugin            0.00

Compilation times for Typers.scala (5500
lines of code) on an Intel I5 @ 2.67 Gz
```

We believe that the big part of the conversion overhead comes from the inefficient organization of the big pattern match that comprises the conversion function, because such pattern match is in $O(n)$ and also relies on run time reflection. In project Palladium[13], abstract syntax trees have integer tags, which can be efficiently matched upon in $O(1)$.

5.4 Jar Example

We packaged the sources, the binaries as well as the ASTs for the whole Scala Library in order to have a rough approximation of the size ASTs could take compared to other packages.

Simplified output of `ls -l`:

```
drwxr-xr-x  5 4096 api
drwxr-xr-x 16 4096 asts
drwxr-xr-x  3 4096 classes
-rw-r--r--  1 767818
tests_2.11-0.1-SNAPSHOT-asts.jar
-rw-r--r--  1 5524069
tests_2.11-0.1-SNAPSHOT.jar
-rw-r--r--  1 15858557
tests_2.11-0.1-SNAPSHOT-javadoc.jar
-rw-r--r--  1 891105
tests_2.11-0.1-SNAPSHOT-sources.jar
```

ASTs take a little less space than the sources, but binaries are definitely bigger.

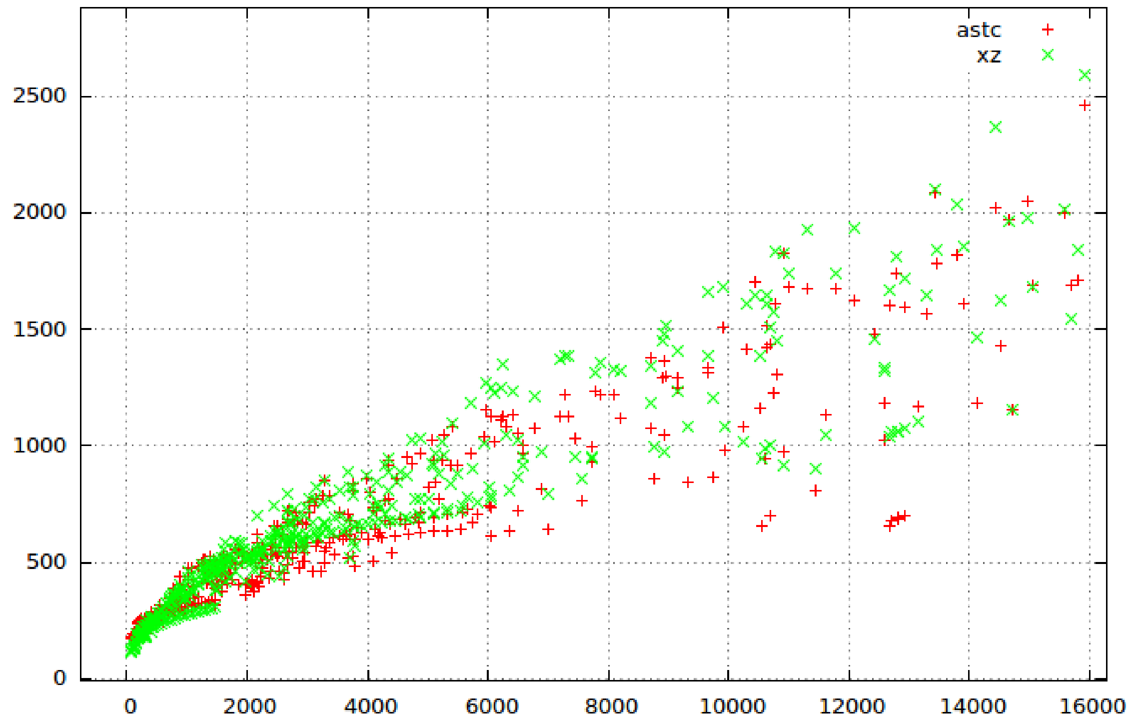


Illustration 1: With names and preprocessing of `showRaw`. x: size of `showRaw` output. y: sizes of the compressed files. All sizes are in bytes.

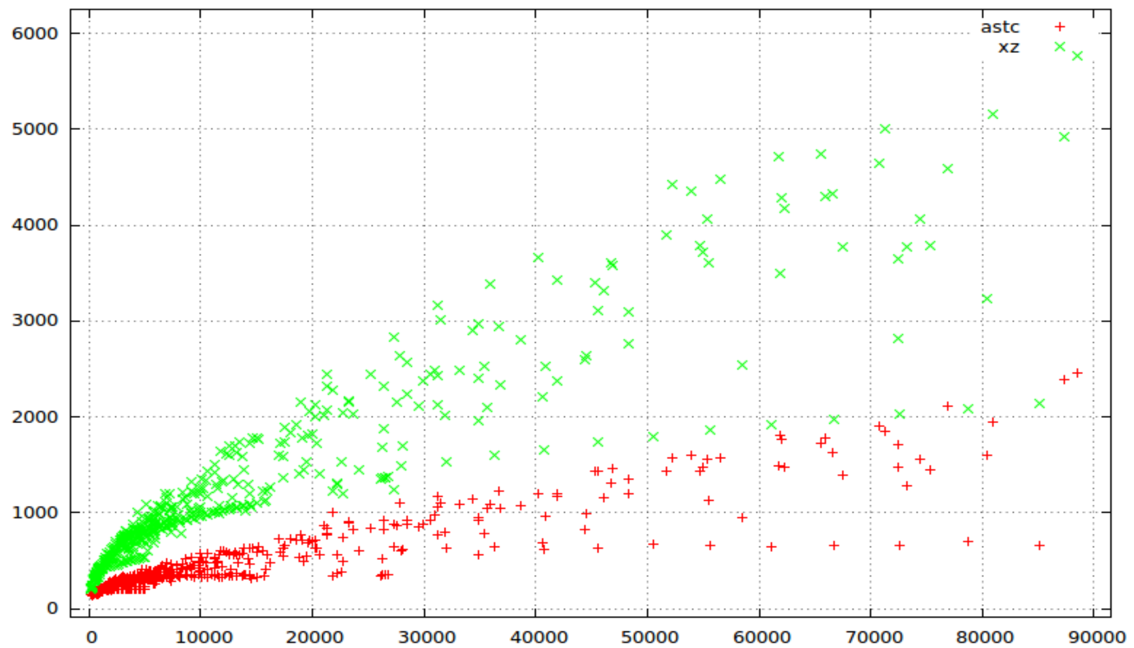


Illustration 2: Without names and no preprocessing of `showRaw`. x: size of `showRaw` output. y: sizes of the compressed files. All sizes are in bytes.

6 Future Work

6.1 Storing Type Hierarchy, Symbols and Constants

The goal of the AST persistence project was to find a way to store ASTs in a compressed way. This goal is achieved in the sense that the results of compression are better than a classic LZMA.

However this is only a prototype: to be properly usable, the plugin would require to store type, constant and symbols informations.

Storing constant for instance is trivial: it is basically the same as names and could rapidly be added.

Such informations could also be used for IDE integration and interpretation, which are among the other subparts of the Palladium project. Once the exact fields required for such a tool are known, the plugin will be easily extended.

6.2 Storing Compressed ASTs in .class File at Compile Time

For simplicity and to avoid loading in memory unnecessary data at run time, our plugin stores the ASTs in a separated compilation folder. Since those files are small it would be possible to store them directly into bytecode, avoiding the generation of more files at compile time. During packaging the compressed ASTs could then be extracted from the bytecode and packaged, like it is now the case with the `packageAst` SBT task, as an artifact with its own classifier.

6.3 Scalability and Going Further

In this implementation, we used the XZ library to perform a compression on our internal representation of the ASTs. The implementation is such that replacing this library by any other one would be really easy and, therefore, if a library with better ratios appears, we should be able to scale our performance by integrating it. We could imagine a system in which the user chooses among a list of such libraries which one he wants to use, according to his needs in terms of space / run time.

7 Conclusion

During the first phase of development we read numerous papers[14] on tree compression, leading us to a better understanding of the challenges it raises. Developing the compression algorithm was a really interesting challenge.

Its implementation is however complicated as we had to do some complex tree manipulations. The compression library was partially optimized to avoid large overhead on compilation time, but unfortunately its main time consumption comes from the translation to our internal representation, which is a cornerstone of the whole implementation.

The AST persistence project was developed as a part of the Palladium project. The main use of the AST persistence compiler plugin were therefore developed at the same time and the exact informations required to be stored were not known during the development phase. As such, our solution targets Scala trees but its port to Palladium seems straightforward, especially given the only non-tree fields in Palladium trees are primitives and strings (names, symbols and types are represented as trees).

For this project we feel like we achieved our goals, since we found a way to efficiently store ASTs along with their names. We also think that adding other metadata such as constants would be easy.

Moreover, the implementation of the plugin respects the transparency requirement that we wanted for the user, giving him a very simple way to add the plugin as well as other tasks related to them through SBT. The toolbox also provides a simple, neat interface to get back ASTs from symbols.

References

[1]Scala macros: docs.scala-lang.org/overviews/macros/overview.html

[2]Scala reflection APIs: docs.scala-lang.org/overviews/reflection/overview.html

[3]Quasiquotes: infoscience.epfl.ch/record/185242

[4]Slim Binaries: wiki.tcl.tk/4400

[5]Lempel-Ziv-Welch: en.wikipedia.org/wiki/Lempel-Ziv-Welch

[6]*Efficient Lossless Compression of Trees and Graphs*: www.cs.duke.edu/~reif/paper/chen/graph/graph.pdf

[7]Typers: github.com/scala/scala/blob/2.12.x/src/compiler/scala/tools/nsc/typechecker/Typers.scala

[8]XZ library: tukaani.org/xz/java.html

[9]LZMA: www.7-zip.org

[10]SBT 0.13.5: github.com/sbt/sbt/releases/tag/v0.13.5

[11]Classifiers options in SBT: www.scala-sbt.org/0.13.5/docs/Detailed-Topics/Library-Management.html

[12]Scalatest funSuite : doc.scalatest.org/2.1.7/#org.scalatest.FunSuite

[13]Palladium project: scalareflect.org

[14]See: goo.gl/FrjsdR