STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.01

COMPILER FRONT END FUSION: UNDO DESUGARING IN LANGUAGE PROCESSING TOOLS

ARTÚR POÓR, TAMÁS KOZSIK, MELINDA TÓTH, AND ISTVÁN BOZÓ

Abstract. Compiler front ends often perform *desugaring* on the source code while constructing the abstract syntax tree (AST). A programming language processing tool (such as a refactoring tool) working with the desugared AST perceives the code at this abstract level, and loses information on the rich syntax used in the actual source code. This paper discusses the concept of *front end fusion*, a technique which may help language processing tools to retain the syntactic sugar information on the source code in the presence of desugaring compiler front ends. We propose a hybrid front end created from two separate front ends: one provided by the compiler, which offers type information, and another one, which provides the details of the concrete syntax used in the source code. Specifically, we show how to construct a hybrid front end in a language processing tool for the Scala programming language.

1. Introduction

Programming language processing tools provide invaluable help during software development and maintenance. They can statically analyse source code for debugging, code upgrade or grokking purposes, and they can perform source code transformations and refactoring as well. These tools typically need to be able to parse and pretty print source code, and may also require semantic information, e.g. the type of expressions and the result of name resolution.

There are two major approaches to implement language processing tools. Firstly, the tool may have a custom lexer, parser, type checker, static semantic analyser, and pretty printer built in, and tailored for, the tool (*standalone*

Received by the editors: April 17, 2018.

1991 *Mathematics Subject Classification.* 68N15, 68N20.

1998 *CR Categories and Descriptors.* D.3.4 [**Programming languages**]: Processors – *Compilers.*

Key words and phrases. parser, abstract syntax tree, compiler front end, syntactic sugar, desugaring, refactoring.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

approach). Secondly, the *external* approach relies on some existing, widely-used compiler infrastructure for the given programming language. However, as we shall see, both approaches have disadvantages.

Older compilers provide no convenient ways to access the output of the compiler front end. For example, earlier versions of GCC (before v4.5) offer intermediate files for observing available information between different compilation phases – which is a rather inconvenient input to a language processing tool. This is where the standalone approach may be needed: the tool needs to re-implement (a part of) the compiler front end for the language. Obviously, this is quite expensive from the tool developer's point of view. Moreover, this makes the tool more vulnerable against the evolution of the programming language. Modern compilers such as Clang [10, p. 32], GHC [3] or Scalac [14] provide APIs to access (annotated) abstract syntax trees (AST) at different stages of the compilation process. Annotated ASTs may convey not only syntactic information, but semantic information (e.g. types) as well. This turns out to be a useful input for a language processing tool – a clear benefit of the external approach.

Rich languages offer a great amount of syntactic sugar, so that programmers can write terse, expressive, and easy-to-read code. The syntactic sugar, however, is typically eliminated from the AST. The compiler replaces certain fancy programming language constructs with semantically equivalent simpler constructs (often referred to as *core language* constructs). This *desugaring* process results in loss of information, which can be a disadvantage of the external approach: the language processing tool will be unable to reproduce the original, syntactically rich source code. Although syntactic sugar does not affect the meaning of a program (with respect to core language constructs), it does have a significant impact on readability and maintainability – i.e. code quality. Therefore recovery of syntactic sugar in a language processing tool is an essential issue. For instance, we would like to observe the original, rich syntax, when the tool communicates analysis results back to the programmer, or pretty prints the code.

As the main contribution, this paper proposes the concept of *front end fusion*: a technique to preserve syntactic sugar for a programming language processing tool, if the external compiler infrastructure used by the tool applies desugaring during the construction of annotated abstract syntax trees. We propose a *hybrid front end*, a language processing tool front end, which is the result of front end fusion: it is hybrid because it combines external and standalone front ends. The presented approach performs a fusion of a custom standalone parser and an external compiler infrastructure when creating the hybrid front end. The main advantage of the presented methodology is to

rely on an external compiler front end, use the static semantic information calculated by the compiler, and replace its parsed information with a "non-desugared" syntax tree.

In the presentation below we show how to assemble a hybrid front end for Scala. The concrete problem to solve is to obtain an AST representing the rich, sugared syntax of a Scala source code, and annotate its nodes with type information provided by the desugaring Scala compiler.

The rest of the paper is structured as follows: in Section 2 we present desugaring in Scala, and provide a few examples. Section 4 describes some difficulties in front end fusion, and Section 5 provides the fusing algorithm. Section 6 presents a discussion about the presented methodology. Finally, in Sections 7 and 8 we present related work and conclude the paper.

2. Desugaring

Scala is a particularly good language to study desugaring, since it heavily relies on syntactic sugars. For example, in this language one-argument methods can be invoked without dot and a pair of parentheses as well. This makes both `args contains "--help"` and `args.contains("--help")` valid. Furthermore, the anonymous (or lambda-) function that increases an integer by one may be written as `_ + 1`, which will be expanded into `x => x + 1`. The **`for`**-loop is also a syntactic sugar, and not part of the core language. The loop that prints powers of two to the standard output is the following:

```
for (e <- List(0, 1, 2, 3, 4)) println(Math.pow(2, e))
```

This may as well be written using the `foreach` method:

```
List(0, 1, 2, 4).foreach{ e => println(Math.pow(2, e)) }
```

Lastly, the expression which overwrites an element of an array is as follows:

```
val xs : Array[String] = Array("zero", "one", "")
xs(2) = "two"
```

The second line may also be written as

```
xs.update(2, "two")
```

In all of these examples the compiler rewrites the former to the latter during parsing. An important consequence of these and the many other syntactic sugars is that Scala is especially well-suited for embedding languages (e.g. creating embedded domain-specific languages, EDSLs). However, syntactic sugars are rather ubiquitous, and can be found in other languages as well. In Java, anonymous functions are syntactic sugars for instances of classes with suitable "functional interface". Anonymous functions have the benefit that they are easier to construct and pass around, especially when working with

streams. Another nice example of syntactic sugar is the do-notation of Haskell, which makes it possible to write imperative style code in a purely functional language.

Syntactic sugar can be defined in terms of *rewriting rules*. A rewriting rule specifies the equivalent language constructs of the core language, thus it gives semantics to a syntactic sugar. The application of the rewrite rules may take place at different phases of the compilation process. For example, semicolon inference (e.g. in Scala and Eiffel) may be performed by the lexer. Operator syntax in Scala is rewritten to method calls during parsing. Finally, lambda-functions are rewritten to occurrences of `PartialFunction` or `Function` objects after typing, in a separate phase. In the end, however, the compiler front-end can output a desugared annotated abstract syntax tree containing the constructs of the core language.

Desugaring is not an injective function, different source code may result in the same desugared AST. On the one hand, the desugared AST is convenient to work with in the compiler, which is only interested in whether the code is semantically correct, and in the meaning of the code. On the other hand, the desugared AST may be too abstract to work with in a static analyser, in a refactoring tool, or in a pretty-printer, where the faithful reproduction of the original source code is expected.

Another source of information loss about the syntax used in the source code is demonstrated by the following example. Consider a simple Scala class, which implements a counter. It has a hidden mutable variable `count`, an `increment` procedure to increase `count` by one, and a `get` function to retrieve the current value.

```
class Counter {
 private var count : Int  = 0
 def increment()   : Unit = count = count + 1
 def get()         : Int  = count
}
```

The compilation technique used in the compiler turns the hidden mutable variable into even more hidden ("object-private"), generates a getter (`count`) and a setter (`count_=`) method, and rewrites every access to the `count` field to an invocation of the getter, and every update to an invocation of the setter. Shall we consider this as removal of syntactic sugar? Or is this `Counter` example a counter-example to desugaring? In any case, when pretty-printing the AST constructed by the compiler, the class looks quite different compared to the original source code.

```
class Counter {
  private[this] var count : Int = 0
```

```
  private def count : Int = count
  private def count_=(newVal : Int) : Unit = count = newVal
  def increment() : Unit = count_=(count.+(1))  // + is a
 method in the Int class
  def get() : Int = count
}
```

On a side note, this code may seem broken because of a name conflict between the field and its getter method. If we investigate the AST directly, we discover that the name of the field is not "`count`", but "`count `". The extra space character in the name of the field is not handled properly by the standard pretty-printer, and this causes the confusion. The right way to pretty-print the AST would be to use a so-called "literal identifier", as follows.

```
private[this] var count : Int = 0
private def count : Int = count
```

All in all, this example also makes it clear that the abstract representation of the code in the compiler-generated AST may lose too much syntactical information about the source code.

3. Hybrid front end

In the presence of a desugaring compiler and an independent parser producing accurate, syntactically sugared ASTs, a hybrid front end can be assembled. The hybrid front end produces an AST which is built from the AST of the custom parser, which avoids desugaring, and preserves all the syntactical information available in the source code. Then, this AST is combined with the desugared AST constructed by the desugaring compiler front end, which contains collected and inferred static semantic information. In this approach only a parser (and a lexer) may need to be developed, and the "hard part", the semantic analyses including name resolution and typing can be carried out by an existing tool, the compiler. This combination of the *standalone* and *external* approaches should be a good trade-off for many cases.

In this paper we investigate how to build such a hybrid front end for the Scala language. Scala is selected as case study for its richness in syntactic sugars, its desugaring compiler. Fortunately, there is no need to develop an accurate parser for Scala: Scalameta, an open-source meta-programming library [13], suits our needs. The proposed hybrid front end relies on Scalameta to parse source codes, and on the Scala compiler to resolve names and infer types. In other words, the parser of the Scala compiler is "replaced" by the parser of Scalameta, as illustrated on Figure 1.

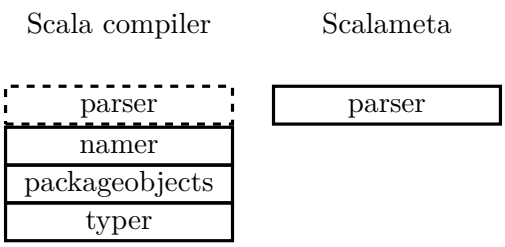

FIGURE 1. Phases of the hybrid front end for Scala.

4. Difficulties in front end fusion

Our main goal is to propose a hybrid front end for Scala producing syntactically rich AST annotated with proper type information. The front-end constructs the AST using Scalameta, and attaches type information computed by the Scala compiler. This annotated AST is an excellent input for various language processing tools.

In order to find the type of expressions represented by the nodes of the Scalameta AST in the typed, desugared AST produced by the compiler, a matching between the two ASTs must be established. The two ASTs have a very similar structure, apart, of course, from the nodes representing a syntactic sugar in the Scalameta AST and their desugared counterparts in the compiler AST. However, the two tools use different names for the same syntactic categories. Literals are represented with nodes of type `Literal` in the Scala compiler, while Scalameta uses type-specific specializations of the `Lit` type. Therefore, in the case of the selected two tools, a matching between the two ASTs based on node types is cumbersome to define. The position information attached to AST nodes proved to be a better basis for the matching. The details of this typing technique will be discussed in Section 5.1. Before that, we investigate two issues which can hamper our fusion approach.

4.1. **Position consistency.**

Position based matching works when both terminal and non-terminal nodes have information about their positions in the source file. Position ranges of non-terminal nodes are synthesized from the positions of their children.

We can say that a node from one of the ASTs and a node from the other AST are in *same-position relationship*, if the position ranges defined by their tokens are equal. The same-position relationship between the two tools is position consistent if it is a one-to-one relationship. In this case, the fact that two nodes from the two ASTs are in same-position relationship guarantees that they are the roots of subtrees representing the same code fragment.

Unfortunately, Scalameta and the Scala compiler are not position consistent. Some of the desugaring transformations can result in position inconsistencies. An example will be presented in Section 5 (Figure 3).

4.2. **Preservation of types in desugaring.** When we copy type information from the typed AST to the sugared AST, we identify matching AST nodes using the same-position relationship. If a node c in the compiler AST is in this relationship with a node s in the sugared AST, we copy the type information from c to s. This approach is correct, if c and s represent Scala expressions of the same type.

In the case of desugaring, however, nodes in the same position in the two ASTs may refer to Scala expressions of different types. Consider, for example, the `increment` method of `Counter` in Section 2, where the assignment to the `count` variable is desugared to the invocation of the setter method `count_=`. Here, the `count` variable on the left-hand side of the assignment operator is in same position relationship with the setter method. Note that the type of `count` is `Int`, and the type of `count_=` is the function type `(Int) : Unit`. Hence it is an error to copy the type information from the compiler AST to the sugared one with respect to these two AST nodes.

Section 2 offers examples of type-preserving desugarings as well. When desugaring `args contains "--help"` to `args.contains("--help")`, the types for all pairs of nodes in same position relationship are identical. The same holds for desugaring the anonymous function `_ + 1` to `x => x + 1`.

Note that nodes inserted by the compiler during desugaring do not cause a problem if they are inserted to "unused" positions.

The construction of the hybrid front end would be easy if position consistency and type preservation held. In that case nodes in same-position relationship would represent the same expression, thus they would have the same type. Unfortunately, these properties do not hold for the chosen front ends: the Scalameta library and the Scala compiler. This may lead to annotating with ambiguous and even incorrect types, as we shall see in Section 5.1.

5. Fusion of two compiler front ends

Now we need to investigate how to use Scalameta and the Scala compiler together. The hybrid front end traverses the sugared and the desugared ASTs from top to bottom. The output is an annotated AST, which includes all terminal and non-terminal nodes of the sugared AST, as well as the semantic information of the desugared AST, as presented in the rest of this section. We conclude with challenges posed by Scala compiler desugarings.

5.1. **Typing a sugared AST.** Our type copying algorithm annotates the sugared AST with semantic information from the desugared AST. The algorithm is given in pseudo-code below. The procedure TYPE takes two nodes, a sugared one and a desugared one, as parameters. The nodes need not be in same-position relationship. The procedure searches for same-position counterparts in the desugared subtree. The procedure ANNOTATE appends the type of a match to a list of possible types for $node_s$. Then the typing algorithm for children of $node_s$ is done, this time the match is set as root of the desugared AST. Note that this choice restricts the search for same-position counterpart to the subtree of the match.

If there is no match found for $node_s$ then there still may be matches for children of $node_s$, so the typing continues.

```
TYPE(node_s, node_d)
  let node_s be the root of sugared and node_d the root of the
    desugared abstract syntax subtrees
  matches = SAME-POSITION(node_s, node_d)
  for match in matches
    ANNOTATE(node_s, TYPE(match))
    for child in CHILDREN(node_s)
      TYPE(child, match)
  if EMPTY(matches)
    for child in CHILDREN(node_s)
      TYPE(child, node_d)
```

The function SAME-POSITION returns a set of desugared nodes that are in same-position relationship with $node_s$. The function performs a recursive depth-first traversal of the desugared subtree. The operator "includes" checks whether a position range of a node is between the start and end of position range of another node. The function SAME-POSITION uses "includes" to skip unrelated parts. In case of position consistency, the function SAME-POSITION always returns a singleton set.

```
SAME-POSITION(node_s, node_d)
  let matches be an empty set of desugared nodes
  if POSITION(node_s) == POSITION(node_d)
    ADD(node_d, matches)
  for child in CHILDREN(node_d)
    if POSITION(child) includes POSITION(node_s)
      UNION(SAME-POSITION(node_s, child), matches)
  return matches
```

We demonstrate the typing algorithm using the desugaring examples from Section 2. We show how to type the anonymous function `_ + 1` and the array element overwrite `xs(2) = "two"`.

For the anonymous function, we are required to use the following class definition because the compiler accepts only complete compilation units.

```
class C {
  val inc : Int => Int = _ + 1
}
```

The sugared and desugared ASTs of the expression `_ + 1` is illustrated on Figure 2. The nodes `ApplyInfix` and `Function` are the roots of the subtrees in the two ASTs. They are in same-position relationship. For each of the children of `ApplyInfix`, the algorithm searches for same-position counterparts in the subtree of `Function`. It annotates `Placeholder` correctly with the `Int` type. However, `ApplyInfix` has ambiguous type because it has two same-position counterparts (a result of the violation of position consistency): the algorithm annotates with `Int` and `Int => Int`. Also, the algorithm does not annotate `Name("+")` since the node is not in same-position relationship with any nodes.

For the array element overwrite, we use the following program:

```
object O {
  def main(args : Array[String]) {
    val xs : Array[String] = Array("zero", "one", "")
    xs(2) = "two"
  }
}
```

The sugared and desugared ASTs of the assignment `xs(2) = "two"` is shown on Figure 3. Every node in the sugared can be annotated since each node is in one or more same-position relationships. The types of `Update`, `Int(2)` and `String("two")` are `Unit`, `Int`, `String`, respectively. However, `Name("xs")` receives two distinct types: the correct type `Array[String` and the type of the method `update`, which is `(Int, String) : Unit`. Again, this is a result of violation of position consistency.

6. Evaluation

The problem to be solved is implementation of a suitable front end for a variety of external language processing tools. Most common features that these tools offer are *static analysis* and *program code transformation*. Many tools statically analyse the code at hand, and even perform transformation

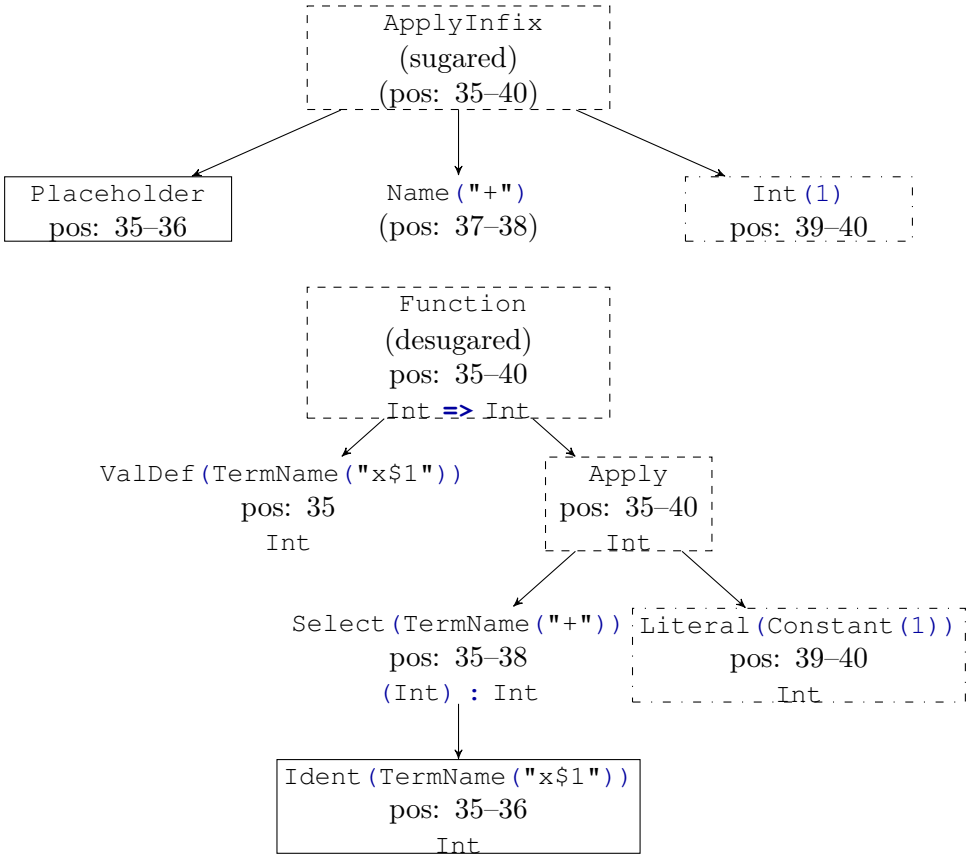

FIGURE 2. Sugared and desugared ASTs of `_ + 1`. Positions are given in offsets.

based on information from static analysis, combining the two features. We elaborate on the effect of hybrid front ends on these features in what follows.

6.1. **Benefits in program code transformation.** External tools which perform source code transformations would benefit from a front end that generates a more accurate source code representation. A typical workflow consists of parsing, locating the code to be transformed in the AST, transformation and pretty printing the AST.

A hybrid front end may improve locating the code in the AST in specific cases. Depending on the compiler front end infrastructure and the order and organisation of the compilation phases, the AST may become subject to optimizations and compile time meta-programming. By the time the external tool

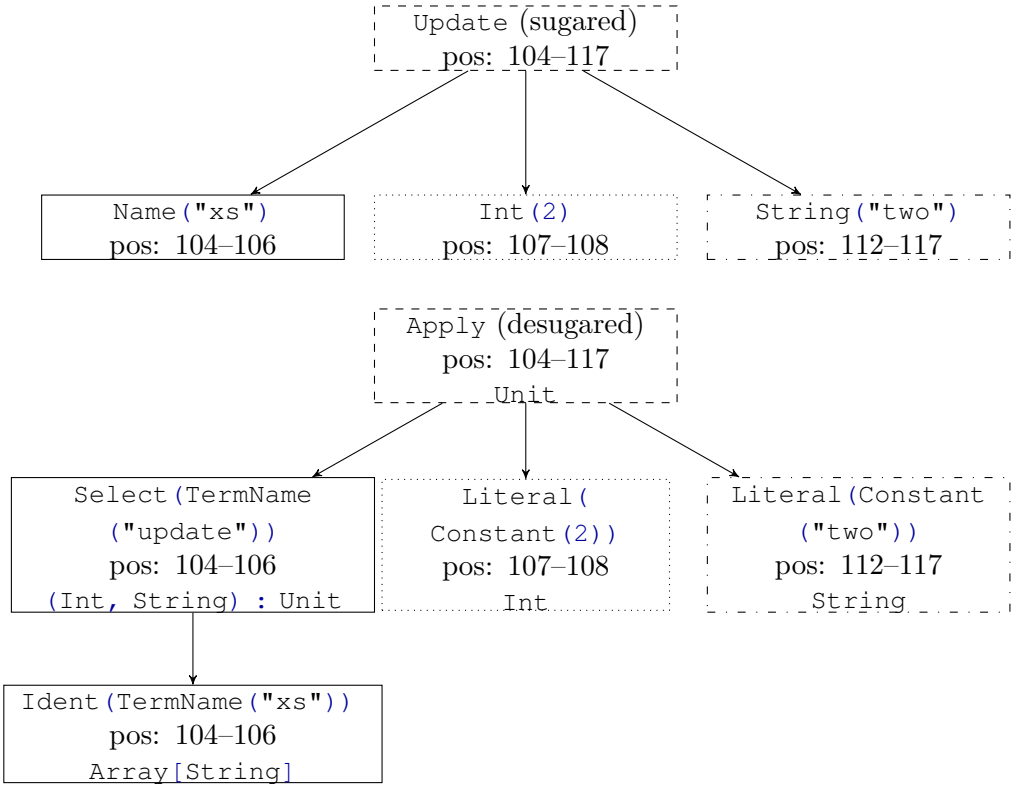

FIGURE 3. Sugared and desugared ASTs of `xs(2) = "two"`. Positions are given in offsets.

receives the AST, constant expressions may be folded, and meta-programming constructs are expanded into generated code. It may happen that the programmer specifies a (part of a) meta-programming construct as the subject of a code transformation, and the compiler front end replaces it with its expansion in the AST, thus the search in the AST fails.

Benefit in pretty printing is clear. A hybrid front end retains lexical and syntactical information on the code. The retained information, which includes syntactic sugars, comments and whitespaces, helps the pretty printer to generate code that pleases the programmer. Without this information, as a side effect, **`for`**-loops may become `foreach` functions, invaluable documentation comments may be lost, and tabs may be replaced with spaces or vice versa, throwing away careful indentation.

6.2. **Effect on static analysis.** The difference between ASTs in representation of the same statement, such as the assignment `counter = counter + 1`,

may cause ambiguity in interprocedural semantic analyses. This statement can be regarded as an assignment, where the value from the right hand side flows to the left hand side, or as an invocation of the setter method `counter_=` in the class `Counter`.

We can resolve this ambiguity in the following way. When the user does not define a setter for `counter`, the compiler will generate a trivial setter. In that case, the analysis can treat the statement as an assignment. Otherwise, the assignment regarded as an invocation of the setter method.

7. Related work

Several programming languages offer syntactic sugars to make software development more convenient and efficient. At first, we present a resugaring technique in Scala, then we investigate other languages as well.

7.1. **Resugaring in Scala.** An alternative approach to build a language processing tool is to further enhance the annotated AST provided by an external tool by undoing the desugaring and adding the sugared syntax tree to the representation. In [12] we introduced a *resugaring* algorithm for Scala by linking two ASTs, a sugared and a desugared, together. The output is a joint AST which includes terminal and non-terminal nodes of both ASTs and the links between them. Figure 4 illustrates the relevant fragment of the joint AST of the `Counter` class, with a sugared AST constructed with Scalameta, and a desugared AST provided by the *scalac* compiler.

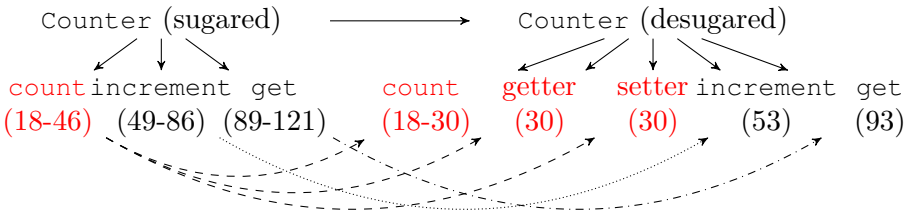

Figure 4. Resugared AST of the `Counter` class. Positions are given in offsets.

The links between the nodes of the two ASTs are established by an algorithm that traverses the two ASTs simultaneously in level-order. The algorithm is presented on Figure 5.

7.2. **Scalameta.** Our choice of parser library, Scalameta [13], comes with capability to annotate ASTs with types. The Scalameta compiler plugin collects information from compiler into semantic database. During parsing, Scalameta

```
RESUGAR(tree_s, tree_d)
  let tree_s be the root of sugared and tree_d the root of the
    desugared AST
  edges = RESUGAR-CHILDREN(tree_s, tree_d)
  if POSITION(tree_s) overlaps POSITION(tree_d)
    ADD(edges, EDGE(tree_s, tree_d))
  return edges

RESUGAR-CHILDREN(tree_s, tree_d)
  let edges be an empty set of links between the nodes
  let mapping be an empty mapping from positions to nodes
  for i = 1 to NUM-CHILDREN(tree_d)
    ADD(mapping,
        POSITION(CHILDREN(tree_d, i)),
        CHILDREN(tree_d, i))
  for i = 1 to NUM-CHILDREN(tree_s)
    if CHILDREN(tree_s, i) has a matching node in mapping
      let match be the desugared node with overlapping
        position in mapping
      UNION(edges,
            RESUGAR-CHILDREN(CHILDREN(tree_s, i), match))
      return edges
```

FIGURE 5. Resugaring algorithm

consults the semantic database and exposes types in its Semantic API. Similarly to our fused front end, Scalameta also uses position information.

So far, Semantic API is limited to symbol types and name resolution. Annotating complex expressions is a work in progress.

7.3. **Syntactic sugars in other languages.** The records in Erlang are taken as syntactic sugar, thus are translated to tuple expressions by the compiler. A record of n fields is substituted with a tuple of $n + 1$ elements, where the very first element is the name of the record and the following elements are the values of the fields (listed in the defined field order).

There are two major refactoring tools for Erlang, RefactorErl [5] and Wrangler. These tools use different approaches in source code processing. RefactorErl follows a standalone approach: it uses its own analyser framework to make every bit of information available. Even the layout, comment, preprocessor constructs and record information are stored in the Semantic Program

Graph (SPG), thus the source code restoration with its context is straightforward. Opposed to RefactorErl, Wrangler [9] is closer to the external approach, since it uses the standard *syntax tools* [1] library that comes with Erlang OTP. Atop of the standard parser, however, the tool annotates ASTs with additional information with macro, record and layout information.

The Glasgow Haskell Compiler [11] uses several well-separated phases in compilation process. Type checking is performed right before desugaring phase, thus extracting representation after type checking phase includes information on syntactic sugar constructs. Haskell-tools [4] refactoring framework uses this representation for further analysis and transformation – this is an external approach.

7.4. **Literate Programming.** Donald Knuth's literate programming [8] allows us to generate documentation and code from a single `WEB` file. The *weaving* process generates TEX document, which can be rendered in human-readable format. The *tangling* process generates code (say, C), which can be compiled and run.

It may be possible to reconstruct the original `WEB` file from TEX document and code. This problem is similar to ours: assuming that front ends for TEX and C can be fused, a hybrid front end could produce a `WEB` file from TEX documentation and C code of the same program. The C code carries information to restore section structure of the `WEB` file. The section names and documentation in each section are part of the TEX file. Annotation comments (e.g. `/*8:*/` and `/*:8*/`) and TEX macros (e.g. `\X8:`) provide a way to establish connection between C code and TEX file. In contrast, we used position information in our hybrid front end for Scala.

7.5. **Preprocessor constructs.** Preprocessor constructs, such as macros, can also be considered as a special form of desugaring. In the original source code a macro application is presented, but usually well before the static semantic analysis a preprocessor substitutes the macro application with the corresponding macro body, and the compiler builds the annotated AST from the expanded macro body. This raises a similar problem as the desugaring in a language processing tool. For example, the source code needs to be pretty printed after a refactoring transformation with the original macro applications kept.

For Erlang, the tool RefactorErl provides a custom parser to store both the original code and the preprocessed one [7, 6]. This makes the pretty printing after refactoring straightforward, and the static analysis more accurate on the expanded AST.

The C programming language also provides a powerful macro system. The tool CRefactory [2] introduces a standalone approach to solve the same issue by preserving the preprocessor directives during parsing.

8. Conclusion

In this paper, we elaborated on how to implement a programming language processing tool in order to minimize the effort. We showed that building upon modern compiler infrastructure helps, but it comes at the price of losing information, due to desugaring. We presented an approach, *front end fusion*, to circumvent this. We proposed an algorithm to construct an annotated abstract syntax tree by fusing the ASTs of the Scala compiler and the Scalameta library.

The presented algorithm is based on the simultaneous traversing of the ASTs to be fused while considering the position consistency of the desugared nodes. We also discussed the need of type-preserving desugaring in terms of the fusion, and presented the solution for the Scala-specific deviations.

We have implemented and evaluated our methodology by creating a language processing tool for Scala with the aim of providing a refactoring framework for parallelisation. Probably, the presented approach may be used for other programming languages as well.

9. Acknowledgement

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

We would like to thank the anonymous reviewers for calling our attention to literate programming.

References

[1] Ericsson AB. Erlang Syntax Tools User's Guide. http://erlang.org/doc/apps/syntax_tools/users_guide.html, 2018.

[2] Alejandra Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005. AAI3199001.

[3] Adam Gundry. A Typechecker Plugin for Units of Measure. *SIGPLAN Not.*, 50:11–22, August 2015.

[4] Haskell-tools Refact. A GHC based toolset for Haskell programming. http://haskelltools.org, 2018.

[5] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Melinda Tóth, István Bozó, and Roland Király. Modeling semantic knowledge in erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT, Sp. Issue, Studia Universitatis Babeş-Bolyai, Series Informatica*, volume 54, pages 7–16, 2009.

[6] Róbert Kitlei, I. Bozó, Tamás Kozsik, Máté Tejfel, and Melinda Tóth. Analysis of preprocessor constructs in erlang. In *Proceedings of the 9th ACM SIGPLAN Erlang Workshop*, pages 45–55, Baltimore, USA, September 2010.

[7] Róbert Kitlei, László Lövei, Tamás Nagy, Zoltán Horváth, and Tamás Kozsik. Layout preserving parser for refactoring in Erlang. *Acta Electrotechnica et Informatica*, 9(3):54–63, 2009.

[8] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[9] Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In *Partial Evaluation and Program Manipulation*, San Francisco, California, USA, January 2008. Assoc of Computing Machinery.

[10] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 1st edition, 2014.

[11] Simon Marlow and Simon Peyton-Jones. The glasgow haskell compiler, 2012. in The Architecture of Open Source Applications (Volume II: Structure, Scale, and a Few More Fearless Hacks), http://aosabook.org/en/ghc.html.

[12] Artúr Poór and Tamás Kozsik. Resugaring: Undo desugaring in language processing tools. Thessaloniki, Greece, 2017. To appear in the Proceedings of the Symphosium of Computer Languages and Tools.

[13] Scalameta. Metaprogramming library for Scala. http://scalameta.org, 2018.

[14] Dean Wampler and Alex Payne. *Programming Scala – Scalability = Functional Programming + Objects*. O'Reilly Media, 2nd edition, December 2014.

Eötvös Loránd University, Budapest, Hungary
Email address: {poor_a, kto, toth_m, bozo_i}@inf.elte.hu