University of Brasília

Computer Science Department

Automatic Lifting of Functions for Incremental Analysis

Breno Augusto Fatureto de Bortolli

Monograph submitted in partial fullfilment of
the requirements to the Computer Engineering Program at University of Brasília

Advisor
Prof. Dr. Vander Ramos Alves

Co-advisor
Prof. Dr. Leopoldo Motta Teixeira

Brasília
2022

University of Brasília
Computer Science Department

Automatic Lifting of Functions for Incremental Analysis

Breno Augusto Fatureto de Bortolli

Monograph submitted in partial fullfilment of
the requirements to the Computer Engineering Program at University of Brasília

Prof. Dr. Vander Ramos Alves (Advisor)
CIC/UnB

Prof. Dr. Leopoldo Motta Teixeira (Co-advisor)
CIn/UFPE

Prof. Dr. Genaina Nunes Rodrigues
CIC/UnB

Prof. Dr. Ralf Lämmel
Universität Koblenz-Landau

Prof. João José Costa Gondim
Coordinator of Computer Engineering Program at University of Brasília

Brasília, May 12, 2022

"We can't have full knowledge all at once. We must start by believing; then afterwards we may be led on to master the evidence for ourselves."

—St. Thomas Aquinas

Dedicated to

This thesis is dedicated to my aunt Márcia,
without whom I would not be where I am.

Acknowledgements

Thanks to my advisors Vander Alves and Leopoldo Teixeira for giving me consistently good feedback for this thesis and beforehand for my undergraduate research. This period from 2019 to 2022 highly motivated me to keep learning more about deep topics.

Thanks to Frédéric Chopin for writing the music that I listened to consistently over the production of this work, and throughout long days of writing and programming in general.

Thanks to the Free Software Foundation and to all the GNU Emacs developers for developing such a high-value piece of software. The entirety of this work, from coding to writing, was made on Emacs.

Thanks to my family for providing the environment and opportunity for me to discover and nurture my vocation.

Thanks to my girlfriend for reminding me to take care of myself and for always providing kind words of encouragement.

Abstract

Evolution is a fact of software development. Modern software workflows employ software analyses extensively on large codebases that are under constant evolution. As such, analyses need to develop some method that deals with incremental changes in the input, often in ways that reuse previously computed results. Despite that, software analyses often deal with that in ad-hoc ways, requiring high development and engineering costs to deal with evolution. This work develops an approach of systematic reuse based on the natural unfolding of recursive procedures on tree-shaped data structures. A preliminary empirical assessment suggests that the method achieves fine-grained reuse for a class of software analyses We evaluated an implementaion of our method on a couple of scenarios in which the input undergoes evolution, and were able to obtain significant speed-up after successive evolutions when compared to the plain analysis.

Keywords: Static analysis, Evolution-aware analysis, Metaprogramming

Resumo

Evolução é um fato do desenvolvimento de software. Fluxos modernos de desenvolvimento de software usam análises extensivamente em grandes bases de código que estão sob constante evolução. Como consequência, análises precisam desenvolver métodos de lidar com mudanças incrementais na entrada, geralmente de maneiras que reutilizam resultados computados previamente. Apesar disso, análises de software geralmente lidam com isso de maneira *ad-hoc*. Este trabalho tem como objetivo desenvolver uma abordagem de reuso sistemática baseada no desdobramento natural de processos recursivos em estruturas com formato de árvore. Uma avaliação empírica preliminar sugere que o método obtémreuso de granularidade fina para uma classe de análises de software. Avaliamos a implementação do nosso método em dois cenários em que a entrada passa por evolução, e obtemos aceleração significativa da análise em relação à análise simples após evoluções sucessivas.

Palavras-chave: Análise estática, Análise ciente de evolução, Metaprogramação

Contents

1 Introduction **1**

2 Background **4**
2.1 Programming in Racket and Tree-Walk Analyses 4

3 Problem Statement **7**
3.1 Running example 8

4 Method **12**
4.1 Method Definition 12
4.2 Design 14
4.3 Running Example Revisited 19
4.4 Implementation 21

5 Preliminary Evaluation **28**
5.1 `tree-count-even` 29
5.2 `expr-depth` 30

6 Conclusion **35**
6.1 Summary 35
6.2 Limitations 35
6.3 Related Work 35
6.4 Future Work 37

References **38**

List of Figures

3.1 Input tree . . . 8
3.2 Input data after change . . . 10

4.1 Plain analysis and evolution aware analysis . . . 13
4.2 Base analysis procedure. OL refers to *object language*. . . . 15
4.3 Evolution-aware analysis procedure. ML refers to *metalanguage*. . . . 15
4.4 Overview of the Metainterpreter . . . 18
4.5 Input tree . . . 19
4.6 Cache of the `countEven` function . . . 19
4.7 Input data after change . . . 19
4.8 Cache with evicted entries . . . 19
4.9 Cache after update . . . 21

5.1 tree-count-even nodes inserted vs. nodes accessed, method vs. baseline . . . 30
5.2 tree-count-even Running time in milliseconds per iteration, method vs. baseline . . . 31
5.3 expr-depth evolution round vs. number of node accesses, method vs. baseline 33
5.4 expr-depth Running time in milliseconds per iteration, method vs. baseline . 34

Chapter 1

Introduction

As software grows larger and more complex, developers rely more on static code analysis. These analyses are able to derive an ensemble of useful metrics and properties, such as type correctness, degree of code complexity, and unreachable control-flow paths. These analyses are integral to the software development workflow, and allow users to develop programs that have increased correctness and quality guarantees [1].

A large set of static code analyses typically walk through the abstract syntax tree of the program in order to derive useful metrics or properties. In some sense, the task of software analyses is similar to that of a tree-walk interpreter, in the sense that the results of a parent node is derived from a computation involving data of the current node along with recursive calls to the node's children. The recursive shape of the input data is thus mirrored in the recursive, accumulative structure of the analysis. In contrast with interpreters, however, static code analyses are often simpler in complexity, primarily to not run into problems regarding termination [2][3].

Static code analyses, however, often either do not take evolution into account, or implement coarse-grained methods in order to track software changes, such as using checksums or file write timestamps in order to determine whether there have been changes at the level of source code files. Furthermore, these systems often deal with dependencies of the input in regard to the output in an ad-hoc way.

The GNU make program, for instance, verifies file modification timestamps in order to determine whether to re-execute a computation. While the program is able to determine the order in which to carry out operations in regard to dependencies, it requires the user to manually specify the files on which each rule depends.[1]

The low resolution of dependencies and the inability to deal with evolution incurs an expensive computational cost for running the analysis, specially as systems grow larger with more frequent changes. This is a common situation in software development work-

[1] `https://www.gnu.org/software/make/manual/html_node/How-Make-Works.html`

flow. In this scenario, code undergoes some sort of analysis. Developers then make small changes to the code base. After these changes are committed to a central repository [1], the analysis has to be performed again from scratch or taking into consideration low-resolution units of change, such as on the level of individual files.

We can make a distinction between approaches that work on the level of artifacts, such as files, called *tool-based approaches*, and ones that work with the abstractions provided by a host programming language, called *language-based approaches* [18]. We pose that operating on a language-level provides unexploited potential for dealing with evolution, in the grounds that such a level allows us to better investigate the relationships between incremental evolution and the impact on the corresponding computation.

The paradigm of static code analysis is replicated whenever one has to traverse a tree-shaped structure with a recursive function. Examples of such analyses include traversing any model described in a structure such as XML or JSON. As with static code analysis, these analyses compute a result on the overall input based on combinations of partial results of smaller units. Furthermore, these units often have a recursive structure, structure that which is mimicked in the unrolling of the analysis function itself. Despite seminal work in incremental computatoin [4], research in this direction is still scarce.

We thus identify an opportunity to deal with evolution-aware software analyses in a more general language-based manner. To do this, we examine the relation between the recursive analysis functions and the input data structures they operate on, as well as the dependency relations originating from tree-shaped data. In this way, we devise a method that is able to recompute analysis functions in a manner that takes into consideration fine-grained evolution of the input data.

This work explores the relationship between the unfolding of computational processes and the manner in which evolution manifests in tree-shaped data structures. Our method exploits the evaluation mechanism of the language in which the analysis is written in, referred to as the object language, in order to compute applications of an analysis in an evolved input in a manner that reuses previous results. Furthermore, we modify the evaluation semantics of the object language in a way that we store partial results derived in a recomputation for later use. In this manner, we can perform successive applications of an analysis after iterated rounds of evolution on the input. The component that inspects the code of the analysis is called the *metainterpreter*. This component analyses the evolution applied to the input, as well as the code of the analysis function, in order to (1) generate an expression for the result that reutilizes previous computations and (2) evaluate the expression in a bottom up manner, such that we keep track of the partial results generated during the process.

The remainder of the work is organized as follows. Chapter 2 gives background on

tree-walk analysis, as well as a brief exposition of programming in Racket and its metaprogramming abilities. Chapter 3 describes the problem of incremental analysis in detail, and introduces points in which our technique will become relevant. Chapter 4 describes the method as well as its Racket implementation. Chapter 5 carries out a preliminary empirical analysis in order to assess the potential benefits of our method. In Chapter 6 we summarize our work, examine its limitations, discuss related work, and consider future directions.

Chapter 2

Background

In this chapter, we provide a high-level overview of the type of analysis explored in this work, namely tree-walk analyses. We provide an overview of the Racket programming language, and also briefly describe the manner in which metaprogramming is done in this language.

2.1 Programming in Racket and Tree-Walk Analyses

The implementation of the technique presented in this work is done in the Racket programming language. This is a Lisp based on the Scheme dialect. Its simple syntax enables us to easily write programs that operate on programs, thus making it a natural fit for our method. It also provides useful features such as structs and an extensive library of data structures and utilities. Therefore it also gives us the power needed to implement the method without having to concern ourselves with implementing the core data structures and abstractions needed.[1]

We can think of expressions in Racket are parenthesized lists or atoms.[2] These expressions are called *s-expressions*. Examples of atoms are variable identifiers and symbols. Function applications are denoted by lists where the first element is the function and the remaining elements are the arguments. For example, `(+ 1 2)` denotes the addition of 1 and 2, or the application of function `+` to the arguments `1` and `2`.

Note that the syntax of Lisp allows us to easily deal with elements that correspond to code. We do not need to parse code into a detached abstract syntax tree; the tree

[1]The reader who wishes to learn more about Racket may refer to the Racket Language Guide at `https://docs.racket-lang.org/guide/`

[2]Racket also affords some syntactic improvements, such as the use of square brackets, in order to reduce the number of parentheses. For the intents of this work, we take into consideration the simpler, albeit less ergonomic, syntactic structure which can be seen in the Scheme dialect of Lisp.

structure of the code is explicit in the code itself. As a result, we can easily manipulate pieces of data that represent code and later on evaluate them.

We can obtain the data corresponding to a piece of code by *quoting* it. Executing the expression `(+ 1 2)` in a Racket evaluator will give is the result, namely `3`. But if we quote it as follows `'(+ 1 2)` (note the apostrophe), we obtain a list containing the elements `+`, `1`, and `2`. We obtain the first element of a list with the `car` operation, and the remaining elements with the `cdr` operation. Therefore, to obtain the operation from the list we do `(car '(+ 1 2))`, and to obtain the list of operands we do `(cdr '(+ 1 2))`.[3]

Suppose we want to write a function that receives as input an expression and returns the amount of times that the addition operation is executed. We essentially need to write a function that traverses nested lists, accumulating the number of times that the `+` operator is seen. The code for this function is shown in Listing 2.1.

```
1 (define (count-add expr)
2   (cond ((null? expr) expr)
3         ((list? expr)
4          (+ (count-add (car expr)) (count-add (cdr expr))))
5         ((eq? expr '+) 1)
6         (else 0)))
7
```

Listing 2.1: Function that counts the number of additions in an s-expression

Other than analyzing code, we can also execute transformed code with ease. The `eval` function receives an expression (in data form) as input and returns the result of evaluating that expression. For instance, `(eval '(+ 1 2))` evaluates the expression `'(+ 1 2)`, returning `3`. Take, for instance, the situation where we want to print a message whenever the addition function is invoked, without modifying the code of a program. We can define a function that logs a message before returning the sum of the arguments. We then write a function that transforms all the addition calls into this version with logging. After we obtain the transformed program, we simply execute the obtained expression using `eval`. The code for this example is in Listing 2.2.[4]

[3]The name of these operations refers back to the `car` and `cdr` instructions of the IBM 704, for which the first Lisp implementation was created [5]. They can be taken to mean, respectively, *contents of the address register* and *contents of the decrement register* [6].

[4]For more information on implementing full interpreters in Scheme, the reader may refer to Structure and Interpretation of Computer Programs [7].

```
(define (log-add x y)
    (println (format "Adding ~a and ~a" x y))
    (+ x y))

(define (transform-add expr)
  (cond ((null? expr) expr)
        ((list? expr)
         (cons (transform-add (car expr))
               (transform-add (cdr expr))))
        ((eq? expr '+)
         log-add)
        (else expr)))

(define program
  '(+ 1 (+ 2 3)))

(define transformed-program
  (transform-add program))

(eval transformed-program)
; "Adding 2 and 3"
; "Adding 1 and 5"
; 6

```

Listing 2.2: Function that counts the number of additions in an s-expression

One final mechanism that is useful for manipulating programs is *quasiquoting*. Like quoting, this allows us to generate pieces of data that correspond to code. Additionally, it allows us to interpolate expressions that are evaluated at object creation time. While quoting allows us to return expressions *literally*, quasiquoting allows us to insert pieces that are evaluated. Whereas we quote expressions with an apostrophe (`'`), we quasiquote expressions with a backtick (`` ` ``) and prefix evaluated expressions with a comma. As such, `` `(+ ,(* 2 3) ,(* 4 5)) `` evaluates to `'(+ 6 20)`.[5]

[5] The Racket guide contains a detailed section on quasiquoting at https://docs.racket-lang.org/reference/quasiquote.html

Chapter 3

Problem Statement

Evolution is an essential part of software analysis. Artifacts are not analyzed only once after they are complete; on the contrary, they undergo analysis at multiple points as they evolve through the software development lifecycle. As such, analyses need to implement evolution-awareness mechanisms into them. Many tools often take into consideration file modification timestamps in order to determine whether an artifact has undergone a change before rebuilding. With the assistance of a dependency graph, for instance, tools rebuild only the artifacts impacted by a change, and reuse previously generated results.

There are two problems with this. One is that the units of change are often taken to be too coarse grained. This causes more work to be duplicated than needed. Take, for instance, the case where only one statement is changed in a file with thousands of lines. A compiler has to deal with that single file from scratch, and also rebuild artifacts that depend on that file, despite the fact that only a fraction of the actual abstract syntax tree may have changed.

The other problem is that different tools may have different views of what constitutes a change and what are the units of reuse. When dealing with compilers, the unit of change is often taken to be changes to a source code file. In an interactive environment for a interpreted or JIT runtime, the units of change may be an individual function or class definition. This implies in different mechanisms for handling evolution.

As a result of this situation, we have high costs in the computational dimension from unnecessary recomputations, as well as high engineering costs needed to develop custom solutions to handle evolution. We propose a fine-grained mechanism in which the unit of change is a modification to the AST, and recomputation takes into consideration the AST nodes affected by a change.

The advantage of this is that the pattern of modifying nodes in a tree-shaped structured is mimicked in the unfolding of recursive computational processes on these struc-

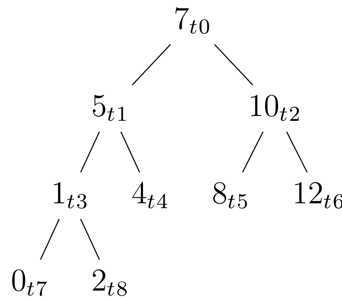

Figure 3.1: Input tree

tures. As such, there is a unexploited correspondence between input data structures and function evaluation that may afford us to provide a uniform method of reuse.

Throughout this work, we use the terms *input data* and *model*, as well as *cache* and *context*, interchangeably.

3.1 Running example

```
data Tree = Node Int Tree Tree
          | Null

countEven tree =
  case tree of
    Node x l r ->
      (if even x then 1 else 0)
      + countEven l
      + countEven r
    Null -> 0

```

Listing 3.1: Running example in pseudo-Haskell

To motivate the research problem addressed in this work, we rely on the running example of counting nodes of a binary tree that stores integers, as described in Listing 3.1.

We note that computation of the function on a node depends on a combination of a local computation with the recursive calls on each of the node's children. Another key aspect is that we visit each node once. The state of the computation is thus carried around purely by the expansion of the recursive calls, i.e., the call stack.

The evolution on the model can be expressed in terms of evolution primitives [8]. An evolution primitive is a change that adds, removes, or changes a node in the model. In the binary-tree example, an evolution primitive may be inserting a node below an existing node, changing an existing node, or removing a subtree.

Suppose, for example, we wish to compute the `countEven` function on the tree depicted in Figure 3.1. We can think of the computation in terms of an expression that expands until all recursive calls are expanded. Let us denote the `countEven` function by f and the root node of the tree by t_0. We start with a base expression.

$$f(t_0)$$

We perform one round of expansion of the expression up to the recursive calls. At this point, we compute a local function that checks whether the value of the node is even, returning 1 if this is the case and 0 otherwise. Since the root node contains 7, this computation results in the atom 0. We also perform one round of expansion up to the recursive calls on the node's left and right children, denoted by t_1 and t_2 respectively.

$$0 + f(t_1) + f(t_2)$$

We can then repeat this process recursively, until we have no more function calls to the base function. At this point, we have an expression whose evaluation reduces to the result of the function application.

$$\begin{aligned}
&0 + f(t_1) + f(t_2) \\
&0 + (0 + f(t_3) + f(t_4)) + (1 + f(t_5) + f(t_6)) \\
&0 + (0 + (0 + f(t_7) + f(t_8)) + 1) + (1 + 1 + 1) \\
&0 + (0 + (0 + 1 + 1) + 1) + (1 + 1 + 1) \\
&\rightarrow 6
\end{aligned}$$

If we go back to the expanded expression evaluation and inspect the step-by-step bottom-up evaluation, we can see that each step of the reduction corresponds to the partial result of the function on some subtree.

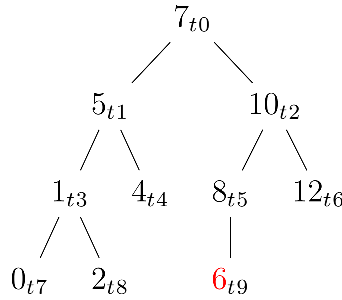

Figure 3.2: Input data after change

$$0 + (0 + (0 + \overbrace{1}^{t_7} + \overbrace{1}^{t_8}) + \overbrace{1}^{t_4}) + (1 + \overbrace{1}^{t_5} + \overbrace{1}^{t_6})$$

$$0 + (0 + (0 + 1 + 1) + 1) + \overbrace{3}^{t_2}$$

$$0 + (0 + \overbrace{2}^{t_3} + 1) + 3$$

$$0 + \overbrace{3}^{t_1} + 3$$

$$\overbrace{6}^{t_0}$$

This whole process shows that there is a correspondence between the shape of the expression and shape of the input data structure, and a correspondence between the bottom-up evaluation of the expression and the partial results of the function on each piece of the model. This suggests the possibility of exploiting these correspondences in order to achieve efficient recomputation.

Let us consider the case where we insert a new node into the tree. The expression expansion is the same as before up to the step where we expand node t_5. At this point, we expand the expression taking into account the new node, but notice that most of the expression remains unchanged. Having performed the process with the initial data structure, we observe that we could prune much of the expression with the partial results computed beforehand.

$$
\begin{aligned}
&0 + f(t_1) + f(t_2)\\
&0 + (0 + f(t_3) + f(t_4)) + (1 + f(t_5) + f(t_6))\\
&0 + (0 + (0 + f(t_7) + f(t_8)) + 1) + (1 + (1 + 0 + f(t_9)) + 1)\\
&0 + (0 + (0 + 1 + 1) + 1) + (1 + (1 + 0 + 1) + 1)\\
&\rightarrow 7
\end{aligned}
$$

A model is taken to be a tree-shaped data structure that represents an artifact. This can be, for instance, an abstract syntax tree or an XML description of the artifact. In the running example, the model is the binary tree. We focus on analysis functions on a model. We require the analysis function to have a specific compositional structure.

Chapter 4

Method

To address the research problem described in Chapter 3, in this chapter we present a method that transforms a plain analysis into one that is able to reuse previous results in order to compute results for evolved models after an initial computation. The remainder of this chapter is organized as follows: Section 4.1 defines the method; Section 4.2 presents its design; Section 4.3 illustrates the method with an example, and Section 4.4 highlights key aspects of its implementation.

4.1 Method Definition

When we analyze a model for the first time, we store the values of the partial computations in a cache. Afterwards, the model is evolved in some way, as described by an evolution primitive. When running the analysis for the second time, instead of recomputing the results from scratch, we generate an expression that computes the result of the analysis on the evolved model. We use the existing cache to prune out portions of the expression that have already been computed. Afterwards, we obtain the result by evaluating the remaining expression, and enrich the cache with the results of the subexpressions generated in the process. These workflows are depicted in Figure 4.1.

Our method depends on some assumptions about the model and the analysis function. First, the method requires the model to be described in terms of nodes, and those nodes need to be well-founded, i.e., there is a well-founded relation on the set of nodes. We can think of this as stating that the model has a hierarchical description. Second, we assume that the analysis function is compositional, in the sense that the result of the analysis on a node is composed on a combination of a value derived from the current node along with the recursive results of the function on the child nodes.

We hold that these assumptions are reasonable for a large class of analyses and input data structures. These assumptions comprise stateless tree-traversal functions, such as

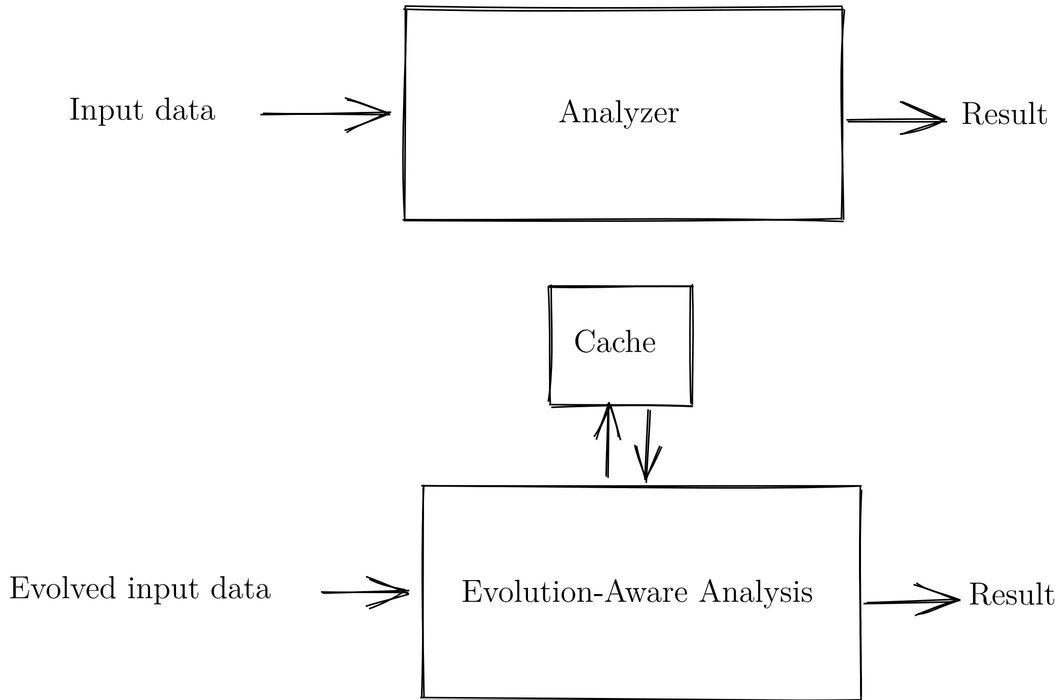

Figure 4.1: Plain analysis and evolution aware analysis

elementary syntactic analyses on programs. Incremental changes to the input data can be expressed in terms of evolution primitives [8].

Models often exhibit a recursive, tree-like structure. As a result, analyses have a compositional structure, based on a base computation combined with the results of recursive calls to subpieces of the model. Exploiting the compositional structure of tree-shaped data structures alongside with the recursive expansion of analyses thus allows us to partition the execution into parts that may be reused later. Therefore, upon changing the model, we may be able to *replay* the analysis in an alternative mode, running expensive computational steps only for the parts that have been changed, while reusing partial results from unchanged portions of the input.

Our technique reuses previous work by recomputing the analysis only for the pieces of the model affected by a change. We assume the existence of a cache that stores results of partial computations on the model. We then apply an evolution to the model, marking the affected parts of the model in the process. Afterwards, we expand the expression of the analysis as much as needed, reusing results of the unaffected parts.

We describe our framework in terms of recursive functions on tree data structures. Given an initial model m, an evolved model m', and a function f, we are able to compute $f(m')$ in a way that reuses previous results, given that $f(m)$ has been computed in the past.

Our technique operates on programs. As such, it is useful to make a distinction

between the metalanguage and object language. The object language is the language in which the analysis under consideration is written. The metalanguage is the language in which we perform the operations of our method, including transformations to code in the object language. We assume that the metalanguage has the capability to inspect, manipulate, and execute code in the object language. We can also think of our method as implementing an analysis domain-specific embedded language (DSEL) for which the host language provides base operations [9].

We now describe the steps of the method. Given that the code of the analysis is written in some object language, we make use of a metalanguage to generate code that computes the transformed function.

Our method comprises the following steps:

1. *Apply evolution to the input.* Given an evolution primitive, apply it to the input data, marking the affected nodes along the process.

2. *Function expansion.* We perform a fixed-point operator unfolding of the function on the evolved model. We use the preexisting cache to prune the expansion and achieve reuse. We also keep track of the partial expressions that arise from traversing the affected nodes of the input data structure.

3. *Expression evaluation.* We evaluate the expression obtained in the previous step in a bottom-up fashion to obtain the result of the function application. At each step of the evaluation, using the information obtained in step 2, we associate the partial value obtained with the corresponding affected node in the input data structure. This information is stored in an auxiliary context.

4. *Cache update.* Using the auxiliary context created in step 3, we add the results computed for the affected nodes back to the initial cache. This allows us to achieve efficient recomputation after successive evolutions.

4.2 Design

We call the metalanguage component of our method the *metainterpreter*. This component can be thought of as a mechanism to modify the original evaluation semantics of the object language taking under consideration evolutions on the input. The metainterpreter communicates with the facilities of the object language in order to accomplish the goal of computing the analysis function with evolution-awareness.

The typical analysis procedure is illustrated in Figure 4.2. An analysis is written in the object language as a piece of data that, alongside the input data, is fed to the interpreter of the object language, producing the analysis result.

In contrast, the evolution-aware analysis process described in this work feeds the object language analysis and input data as input to a metainterpreter written in the metalanguage. The metainterpreter has access to a cache, performs expression expansion, and calls upon the object language interpreter to evaluate object language expressions. As the output, we have the result and, as a side effect, an updated cache.

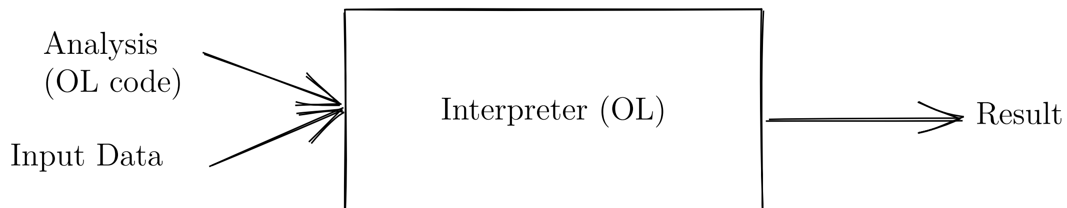

Figure 4.2: Base analysis procedure. OL refers to *object language*.

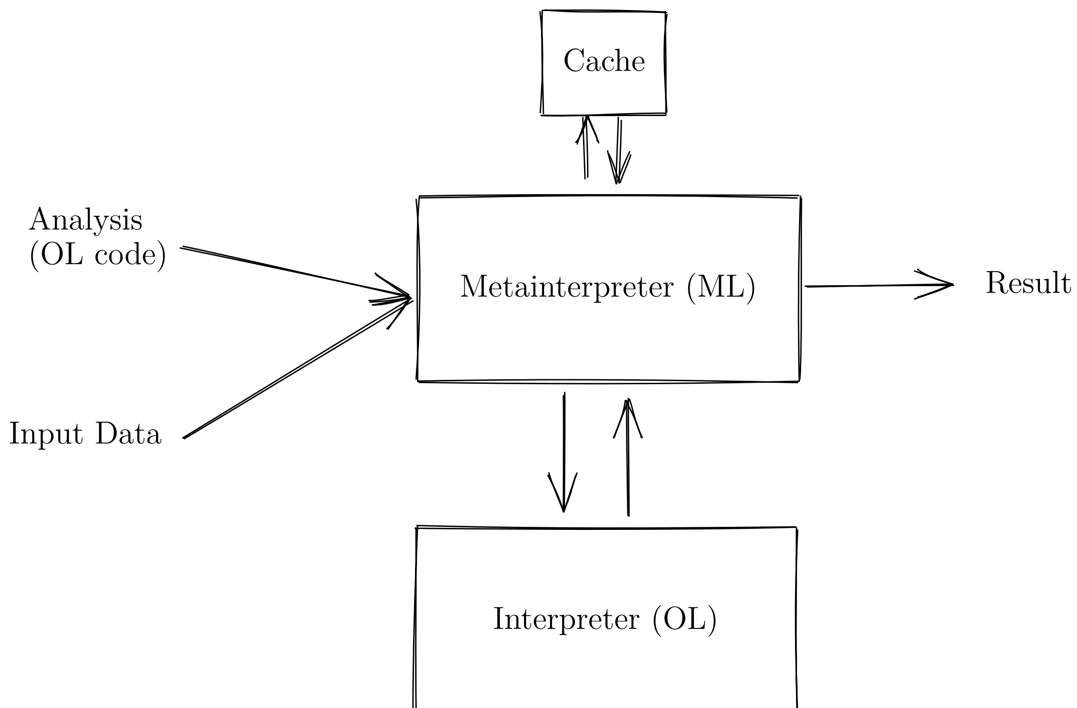

Figure 4.3: Evolution-aware analysis procedure. ML refers to *metalanguage*.

The core of the expression expansion component of the metainterpreter can be described in terms of two mutually recursive functions that traverse the abstract syntax tree of the object language: `unfold` and `reduce`. The former does the

expansion of the expression taking into consideration the preexisting cache of partial results, referred to as context, while the latter is responsible for expanding function applications that are not computed at the time of program transformation. The pseudocode for these two functions is shown in Algorithms 1 and 2.

Algorithm 1 unfold

Input: expression *expr*

Output: result expression *resultExpr*

Result: *bctx* contains mappings from generated expressions to corresponding input nodes

```
case App f arg ← expr do
    if f is transformed function then
        if arg is in context then
            return ctx[arg]
        else
            expr' := rewrite f arg
            resultExpr := unfold ctx bctx expr'
            bctx[resultExpr] <- arg
            return resultExpr
        end
    else
        return reduce ctx expr
    end
end
return expr
```

Algorithm 2 reduce

Input: expression *expr*

Output: result expression

```
case App f arg ← expr do
    arg' := unfold arg
    return App f arg'
end
return expr
```

Algorithm 3 $eval^{\uparrow}$

Input: result expression *expr*
Output: evaluated result
Result: $fctx$ contains mappings from input nodes to corresponding analysis results

```
if expr not in bctx then
    return eval expr                  ▷ expr does not correspond to any node in the input data
else
    inputNode := bctx[expr]           ▷ expr corresponds to some node in the input
    if inputNode in fctx then
        return fctx[inputNode]        ▷ already computed expr; result is in fctx
    else
        if isCompound expr then
            expr := map eval↑ expr
            return eval expr′
        else
            return eval expr
        end
    end
end
```

Algorithm 1 starts by examining if the current expression is the application to the function under transformation. If that is the case, we then check if the argument is in the context. If it is, we simply return the cached result, otherwise we expand the expression and apply unfold once again.

The `rewrite` function shown in Algorithm 2 expands the function expression up to the recursive calls. If the expression denotes the application of some other function, we return a call to the function with each argument unfolded. For any other language construct, we simply return the expression unchanged.

The other component of the metainterpreter is called the *lifted evaluator*, denoted $eval^{\uparrow}$, whose pseudocode is shown in Algorithm 3. This component is responsible for implementing the transformed bottom-up evaluation semantics that we need in order to obtain the updated partial results to be filled in the cache later. The main task of the lifted evaluator is keeping track of the results of the expressions as they are evaluated so we can fill the cache after the process is done. The lifted evaluator is written in the metalanguage, and communicates with the cache and with the interpreter of the object language. An overview of the components of the metainterpreter is shown in Figure 4.4.

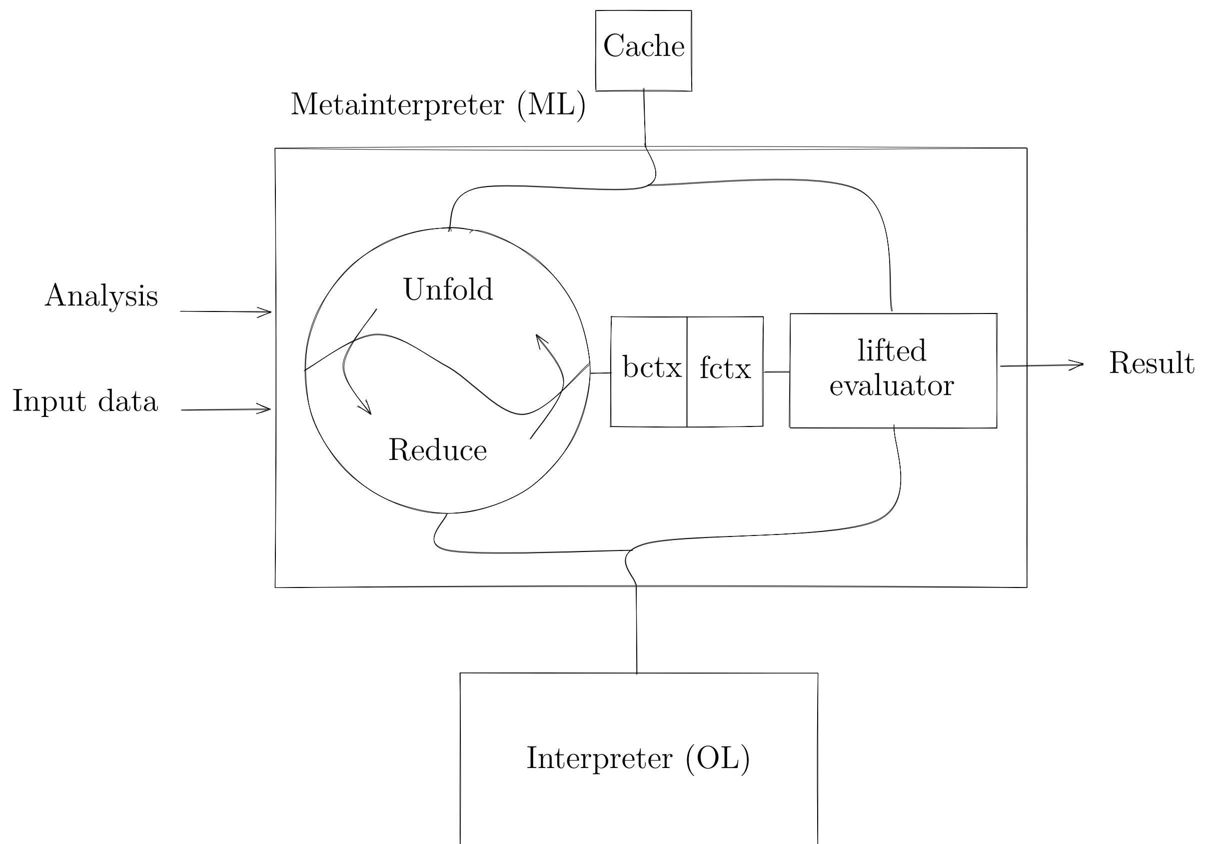

Figure 4.4: Overview of the Metainterpreter

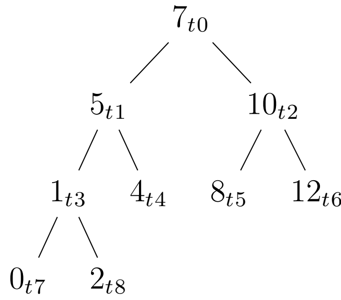

Figure 4.5: Input tree

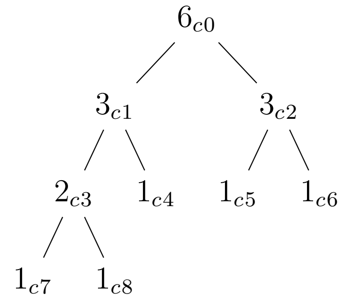

Figure 4.6: Cache of the `countEven` function

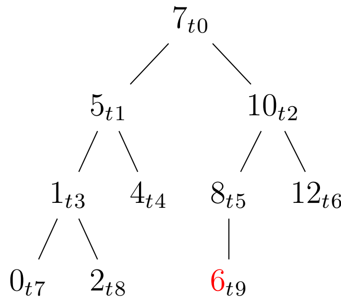

Figure 4.7: Input data after change

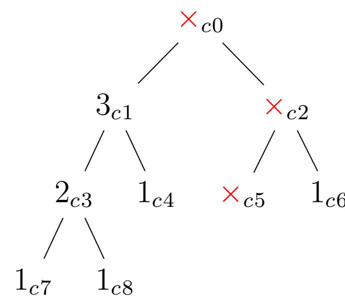

Figure 4.8: Cache with evicted entries

4.3 Running Example Revisited

To illustrate the method define in Section 4.2, we will revisit the `countEven` function from the running example (c.f. Section 3.1). We are given an input tree and a cache containing the results of partial computations on that tree, as depicted in Figures 4.5 and 4.6. We use the correspondence assumption to denote the cache as having a structure isomorphic to that of the tree. This makes evident the correspondence between the expansion of the function and the input structure.

Now suppose we insert a node containing the value 6 below the 8 node, as shown in Figure 4.7. We carry out step 1, updating the cache by erasing the values that correspond to locations affected by the change. This can be seen as marking each node from the path of the parent to the first affected node. Meanwhile, the remaining nodes are left unchanged. This process is depicted in Figure 4.8.

Note that in this step, we use the correspondence assumption to map the nodes that have been affected by the change. This allows us to rewind the computation exactly up to the point where we need and no more.

Now we are able to perform the expansion of the function expression as prescribed by step 2. Taking into account the values of the nodes in the input model, we can then expand the recursive expression step-by-step.

We start with the function call of function f on the evolved model m'.

$$f(m')$$

Next, we perform one round of expansion, denoting the left and right children of the root node of the tree by t_1 and t_2, respectively.

$$0 + f(t_1) + f(t_2)$$

Note that, since the result of the left subtree is not affected by the change, we can reuse the previously computed result (as shown in Figure 4.8). We can thus expand the expression once more with the previously computed result.

$$0 + 3 + f(t_2)$$

We perform this successively, reusing results from unaffected subtrees along the way.

$$\begin{aligned}
&0 + 3 + f(t_2)\\
&0 + 3 + (1 + f(t_5) + f(t_6))\\
&0 + 3 + (1 + f(t_5) + 1)\\
&0 + 3 + (1 + (1 + f(t_9)) + 1)\\
&0 + 3 + (1 + (1 + 1) + 1)
\end{aligned}$$

Finally, given that node t_9 is the newly added leaf node, we reach a base case. The result of the computation is now the evaluation of the obtained expression, as per step 3.

$$\begin{aligned}
&0 + 3 + (1 + (1 + 1) + 1)\\
&= 7
\end{aligned}$$

We also need to associate the evaluation of the subexpressions to the nodes affected by the change, as per step 3. We can visualize this process by decomposing the result expression. Note that each expression points to the subexpression of some other affected node, denoted as $e(t_i)$. We can thus evaluate all these expressions in a way that takes into consideration their dependencies.

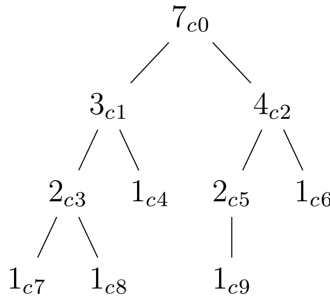

Figure 4.9: Cache after update

$$
\begin{aligned}
&t_0 \to 0 + 3 + e(t_2) && \to 7 \\
&t_2 \to 1 + e(t_5) + 1 && \to 4 \\
&t_5 \to 1 + e(t_9) && \to 2 \\
&t_9 \to 1 && \to 1
\end{aligned}
$$

To finalize, we carry out step 4 and update the cache. The updated cache is shown in Figure 4.9.

We have only visited the nodes affected by the change. In an expensive computation, this could mean that we save significant time by not having to recompute the function for a large amount of nodes.

4.4 Implementation

We implemented the technique described in this work using Racket.[1] In this implementation, both the object and metalanguages are both Racket. The language provides extensive metaprogramming capabilities, therefore being a natural fit for this method. The dynamic character of the language easily allows us to manipulate and evaluate syntax objects on runtime, while also providing useful libraries of standard data structures. One decisive aspect is the homoiconicity between code and data. Since the syntax is essentially the bare abstract syntax tree, we are easily able transform analysis functions in order to leverage our method.[2]

[1] `https://racket-lang.org/`

[2] The implementation is available as a repository on GitHub `https://github.com/brenoafb/evolution-aware-analysis`

One important aspect is that Racket allows us to modify data structures in place. This allows us to apply evolution primitives in place. We can also use the pointer address of composite objects as their identity. This allows fast lookup of large trees in hash maps, which is crucial for the speed of our technique.

In order to implement mappings, such as for the `ctx`, `bctx`, and `fctx` elements of the design, we utilize the built-in `hasheq` data structure.[3] To achieve efficient indexing, the implementation relies on pointers to identify data structures in memory. As such, the implementation relies heavility on mutation. Another benefit of that is that we are able to work with data structures in place, thus allowing us to easily apply evolution primitives and mark affected structures in one operation.

One way to achieve the expansion of the function expression up to the recursion is by using *quasiquoting*. This language feature allows us to specify a piece of data that corresponds to some code, while interpolating expressions that are evaluated when the object is constructed.

Take the `countEven` function from the running example. In Racket, we can implement it in the manner described in Listing 4.1, in the `tree-count-even` function. The implementation requires us to specify a *meta-transformed* version of the function, which instead of returning a value, returns the syntax for the expression of the function. This is accomplished in the code for the `tree-count-even-m` function. Note that this function uses quasiquotes to generate an expression that corresponds to the expression of the function up to the recursion point, shown in lines 18-19 and 21-22).

[3] `https://docs.racket-lang.org/reference/hashtables.html`

```
(struct tree
  ([value #:mutable]
   [left #:mutable]
   [right #:mutable]))

(define (tree-count-even tree)
  (cond ((null? tree) 0)
        ((even? (tree-value-log tree))
         (+ 1 (tree-count-even (tree-left tree))
              (tree-count-even (tree-right tree))))
        (else
         (+ 0 (tree-count-even (tree-left tree))
              (tree-count-even (tree-right tree))))))

(define (tree-count-even-m tree)
  (cond ((null? tree) 0)
        ((even? (tree-value-log tree))
         `(+ 1 (tree-count-even ,(tree-left tree))
               (tree-count-even ,(tree-right tree))))
        (else
         `(+ 0 (tree-count-even ,(tree-left tree))
               (tree-count-even ,(tree-right tree))))))

```

Listing 4.1: Running example implementation in Racket

We implement the key `reduce` and `unfold` functions in a very similar manner to the one described in Section 4.2. The Racket code is in Listing 4.2. The `unfold` function explicitly keeps track of partial expressions generated along the process. These expressions are stored in the *back context* `bctx`, which associates a node in the input data structure to an expression whose result corresponds to the respective partial result. In this manner, we can separate out the expansion of the main expression with the accumulation of partial expressions. This association is done in a hash map where the key is the expression and the value is the corresponding node.

The `unfold!` function (Listing 4.2) implements the core of the metainterpreter. We start by checking whether the expression passed is a function application (line 7). If it is, we need to verify whether it is the application of the function under consideration. If this is the case, we extract the argument of the function application (line 9). If the argument is in the context, we have already computed its result, and we simply return that result (lines 10-11). Otherwise, we recursively call `unfold!` and store this in the `result` variable. This is done so we can later on store the expression obtained in the back context `bctx` (lines 12-14). Note that in the recursive call to `unfold!`, we apply

the meta-transformed function to the argument of the function call. This carries out the process of expanding the function expression up to the recursive calls, which in the formal description is done by the `rewrite` function.

If the expression passed to `unfold!` is the application of some other function, we call `reduce!` on the expression in order to carry out the expansion (line 18). `reduce!` recursively calls `unfold!` in order to execute the full expansion of the function application expression.

We need to implement an evaluation mechanism different to that of the object-language in order to be able to compute the result expression in a bottom-up manner while accumulating the partial results. We do this in the `eval-lift` function, shown in Listing 4.3, which takes the expression-to-node mapping `bctx`, a node-to-result mapping `fctx` to be filled in the process, and the topmost expression. The back context contains mappings from expressions generated during the unfolding process to the corresponding input node. As a result, if the received expression is not in `bctx`, it is an expression that does not correspond to any node, therefore we only return its evaluation without caching its result (lines 2-3). Otherwise, we obtain the input node that corresponds to the expression received (line 4) and check whether we have already computed this expression by verifying whether it is in the forward context `fctx` (lines 4-5). If this is the case, we simply return the precomputed result (line 7). Otherwise, we need to evaluate this expression. If it is a compound expression, we evaluate each of these components with `eval-lift` and then evaluate the resulting expression, storing its result in the `fctx` in the process (lines 9-15). If it is an atomic expression, we evaluate it's result, store it in the `fctx`, and return it.

In order to allow for successive evolutions, we need to update the cache with the results of the partial expressions generated. We do this in the `update-ctx` function, shown in Listing 4.4, which receives the context `ctx` and the result context `fctx`. The function simply merges the two contexts together. When generating the expression for a further evolved input, we can use this updated context in order to reuse partial results produced in the current iteration.

The typical evolution-aware computation flow is described in Listing 4.5. The evolution-aware version of the `tree-count-even` is shown in the `evolution-aware-tree-count-even` function (lines 1-18). This function starts by initializing the back and forward contexts (lines 3-4) which will be used in the expression expansion, carried out by `unfold!` (lines 6-12), and bottom-up evaluation with reuse steps, carried out by `eval-lift` (lines 13-16). Note that in the expression expansion process, we pass as input to `unfold!` an expression that corresponds to the plain function call of `tree-count-even` on the argument. (line 12).

After the evolution-aware version of the analysis is defined, initial analysis is carried

out as usual on an empty context the first time the analysis is executed (lines 20-28). Then, we apply the evolution primitive to the input data structure (line 31). To do this in a manner that marks the affected nodes, we implement modification primitives on the input data structure that mark the nodes affected by the change. We accomplish this by implementing an access function that receives a callback function which is executed on each node accessed by the recursive modifier. Since evolution often involves a top-down recursive traversal of the structure, this does not add significant computational cost. For this example, the callback is shown in lines 23-24. Finally, we execute the same function call to the evolution-aware analysis (line 34). The difference is that this time, the cache will be populated, so we are able to perform the computation with reuse of the results computed in the previous run. This concludes the procedure.

```
1  (define (reduce! ctx bctx f meta-f expr) (cons (car expr) (map (lambda
2  (x) (unfold! ctx bctx f meta-f x)) (cdr expr))))
3
4  (define (unfold! ctx bctx f meta-f expr)
5    (if (func-app? expr)
6        (if (eq? (car expr) f)
7            (let ((arg (cadr expr)))
8                 (if (in-ctx? ctx arg)
9                     (ctx-get ctx arg)
10                    (let ((result
11                           (unfold! ctx bctx f meta-f
12                                    (apply meta-f (list arg)))))
13                      (begin
14                        (hash-set! bctx result arg)
15                        result)))))
16             (reduce! ctx bctx f meta-f expr))
17       expr)
18
```

Listing 4.2: reduce and unfold in Racket implementation

```
(define (eval-lift bctx fctx expr)
  (if (not (hash-has-key? bctx expr))
      (eval expr ns)
      (let ((src (hash-ref bctx expr)))
        (cond ((hash-has-key? fctx src)
               ; we already computed this expr
               (hash-ref fctx src))
              ; haven't computed expr yet
              ((list? expr)
               (let*
                   ((new-expr (map (lambda (x) (eval-lift bctx fctx x))
    expr))
                    (result (eval new-expr ns)))
                   (begin
                     (hash-set! fctx src result)
                     result)))
              (else
               (let ((result (eval expr ns)))
                 (begin
                   (hash-set! fctx src result)
                   result))))))))

```

Listing 4.3: Custom evaluation mechanism implementation

```
(define (update-ctx ctx fctx)
  (hash-union ctx fctx))

```

Listing 4.4: Function used to load newly evaluated results into cache

```
(define (evolution-aware-tree-count-even tree ctx)
  ; Initialize back and forward contexts
  (define bctx (make-hasheq))
  (define fctx (make-hasheq))
  ; Apply evolution-aware analysis
  (define
    expr
    (unfold! ctx
             bctx
             'tree-count-even
             tree-count-even-m
             `(tree-count-even ,tree)))
  ; Evaluate expression bottom-up and obtain the final result
  (define result (eval-lift bctx fctx expr))
  ; Update context
  (update-ctx ctx fctx)
  ; Return result
  result)

(define my-tree (make-new-tree))
(define ctx (make-hasheq))

(define (mark-node-callback tree)
  (hash-remove! ctx tree))

; Execute initial computation.
; Since the context is empty, we will compute from scratch.
(evolution-aware-tree-count-even my-tree ctx)

; Apply evolution
(tree-insert! my-tree some-value mark-node-callback)

; This time, the computation will run with reuse.
(evolution-aware-tree-count-even my-tree ctx)

```

Listing 4.5: Example of typical evolution-aware analysis usage

Chapter 5

Preliminary Evaluation

In order to provide a preliminary evaluation of this work, we use the implementation described in Section 4.4 to evaluate the method in a set of scenarios. In these scenarios, we start with a base input and, at each round, apply an evolution that extends the input in some way. We then evaluate an analysis function on the input.[1]

We implement two analysis functions:

tree-count-even Count the number of nodes containing even numbers in a binary tree. At each round, insert a random number in the tree in a balanced fashion. Run the analysis after each evolution.

expr-depth Compute the total depth of an s-expression in a Lisp program. At each round, mutate the program randomly. Such mutation may be wrapping an existing sub-expression in a structure, or inserting a new expression in the program. Run the analysis after each evolution.

In the `tree-count-even` scenario, we initialize a tree with a given size and apply a random node insertion repeatedly. We assume that this scenario, in contrast to node removal or value update, is representative of the more costly situation in which an analysis needs to be executed multiple times, namely that where the input data structure grows at each round. In the `expr-depth` scenario, we start with a base program, which can be seen as a list of s-expressions, and extend the program by either adding new s-expressions at some level or enclosing existing s-expressions in function applications. An overview of the scenarios is shown in Table 5.

We hold two goals for this experiment:

- Assess *empirical correctness*. The results given by our method should always match those given by the baseline analysis.

[1] The code of the experiments, along with the implementation of the method, is available as a repository on GitHub `https://github.com/brenoafb/evolution-aware-analysis`

- Assess *time benefit*. The execution time for the analysis powered by our method should be lower than the baseline analysis.

In order to accomplish the first goal, for each scenario, we execute the baseline analysis function and the evolution-aware version enabled by our method and compare the results. The criterion is that the results match exactly at each step.

The second goal is accomplished by measuring the execution time and input accesses for each iteration in both scenarios. The first metric determines whether our method obtains measurable speed-up in comparison to the baseline. The second metric is used to determine the degree in which we accomplish reuse. A lower number of accesses of the input data shows us that we accomplish a higher degree of reuse.

The experiments were run on a NixOS 21.11 system with an Intel i7-8650U with 16GB of RAM, on Racket version 8.4.

Scenario	Input Data	Analysis	Evolution scenario
tree-count-even	Integer binary tree	Count even numbers in the tree	Insert nodes in balanced fashion
expr-depth	Lisp programs	Maximum s-expression depth	Randomly mutate program

5.1 `tree-count-even`

For this scenario, we start with a balanced binary tree of 1000 nodes and add 1000 new nodes randomly in a balanced manner, one at a time. We run the analysis after adding each node.

The correctness assessment consists of whether our method obtains the same results as the baseline analysis at each step. We simply evolve the input at each step, execute the baseline analysis, execute the evolution-aware analysis, and compare the results. For each step, the evolution-aware analysis yields the same results as the baseline analysis.

In order to carry out the performance assessment, we keep track of the number of nodes accessed in the input data structure at each round, as well as the execution time of each iteration. We expect a lower number of accesses to correspond to a lower number of computations executing without reuse.

In Figure 5.1 we see that the baseline has to access every node of the tree at each round of analysis, as expected. Our method, however, is able to execute the analysis by accessing only a small fraction of the tree's nodes. This indicates that we have been able to achieve a great amount of reuse throughout the function evaluation.

For the running time, we measure the time it takes to run one iteration, starting at node insertion and ending after the final result and partial results are evaluated. We measure

the execution time with Racket's built-in `(current-inexact-monotonic-milliseconds)` at the two points and compute the difference. Each measurement is collected 10 times. In Figure 5.2, we see the average of the 10 points plotted. The graph shows that time taken for the baseline analysis grows at each iteration, which reflects the growing size of the input data structure. Our method, however, displays a much smaller rate of growth in execution time, along with lower execution time overall. We note that the initial execution with our method requires a much higher execution time, since the cache is initially empty.

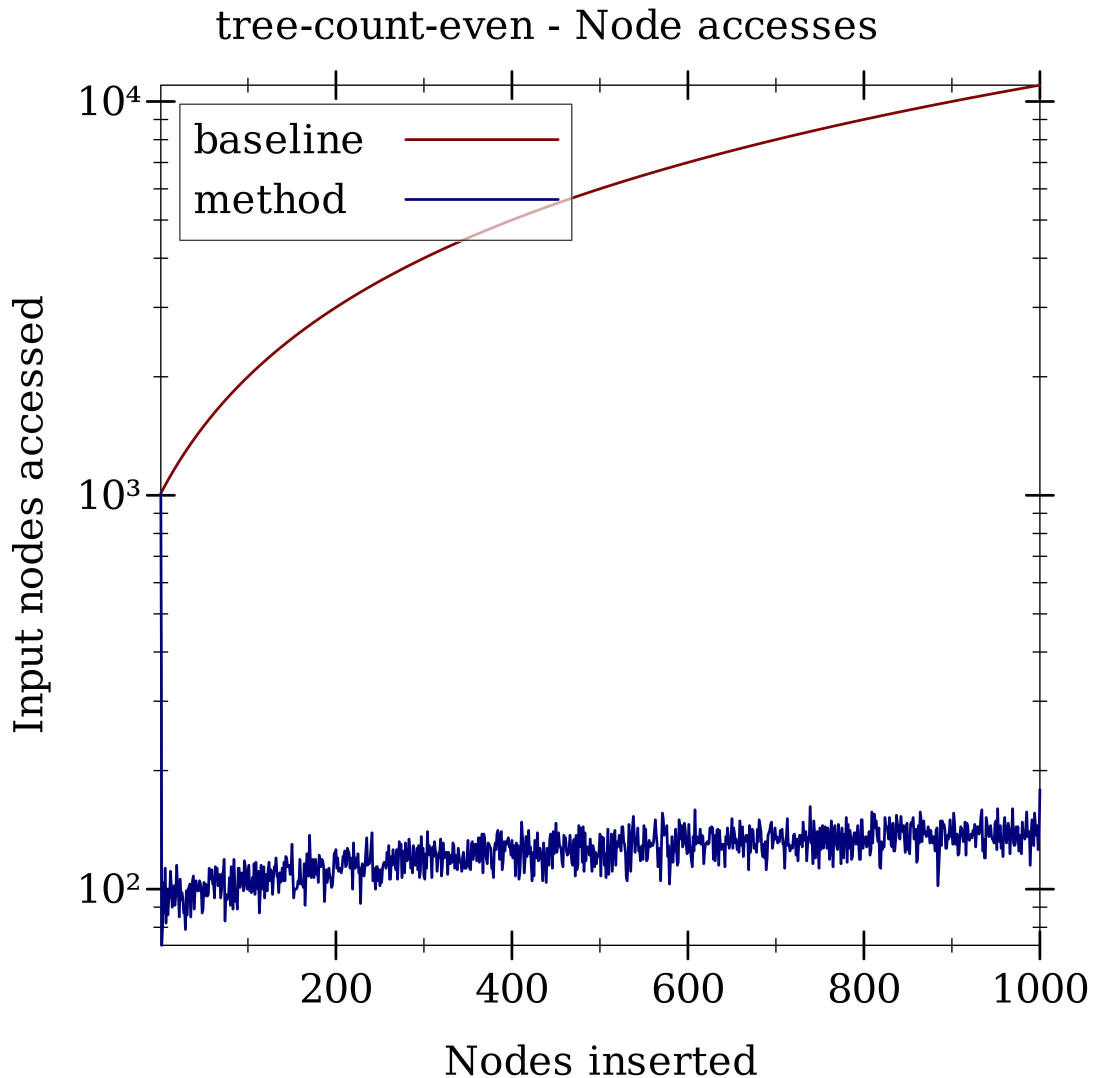

Figure 5.1: tree-count-even nodes inserted vs. nodes accessed, method vs. baseline

5.2 `expr-depth`

In this scenario, we start with a base 100KLOC Racket program and mutate the program 1000 times in a random manner. We perform analysis after each mutation. The analysis

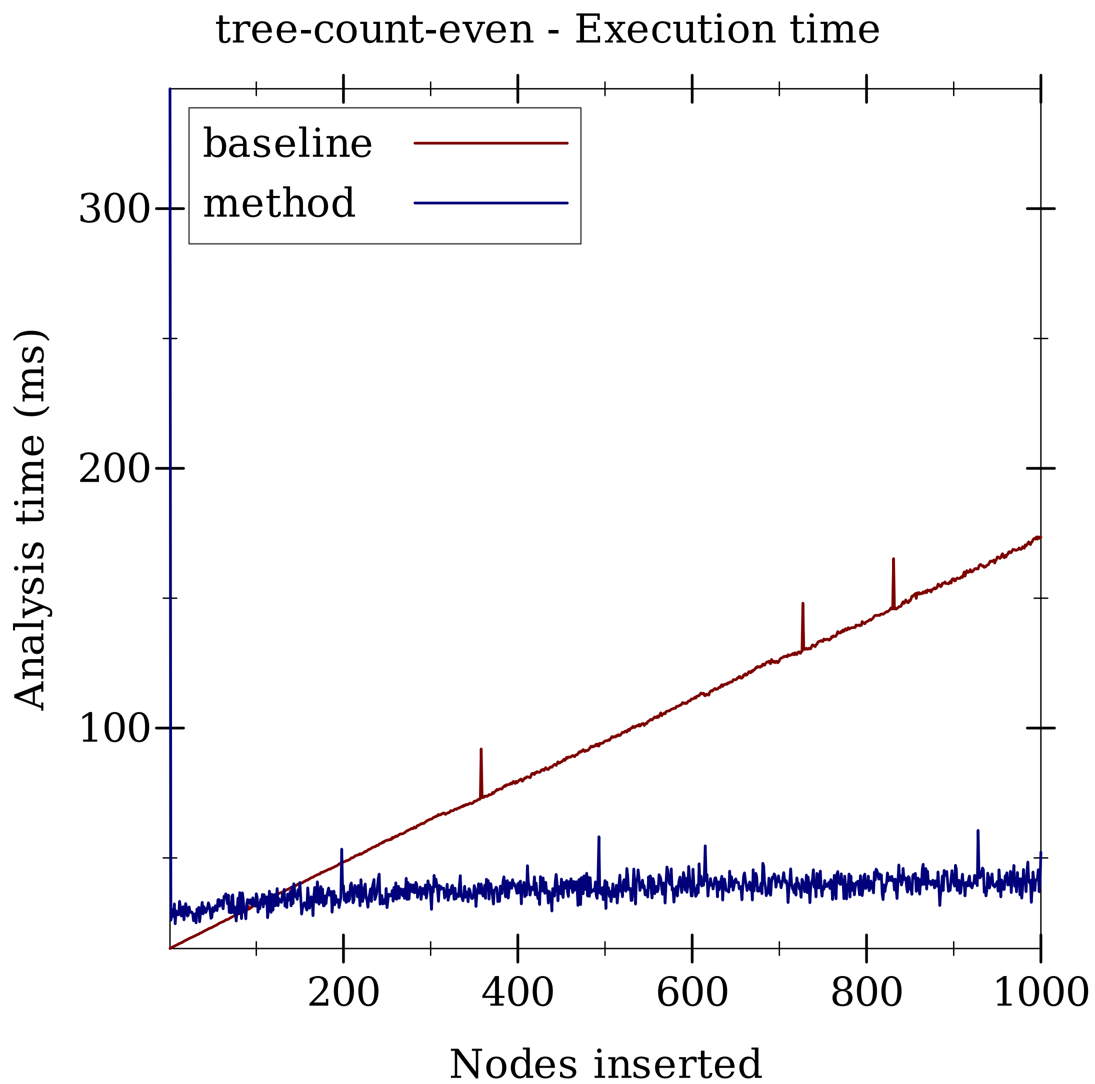

Figure 5.2: tree-count-even Running time in milliseconds per iteration, method vs. baseline

consists of summing the total depth of s-expressions in a program. For instance, an atom is taken to have zero depth. The depth of a list is one plus the depth of its elements. This is a simple compositional analysis that fits our framework. A mutation either inserts a new expression in a list or wraps some random expression in a program in a function application, such that the new expression has one additional nesting level than the original. Note that, in order to perform a mutation, we walk the AST from the root and at each node either continue with 80% chance, wrap the current node in an s-expression with 10% chance, or insert a new randomly-generated expression after the current one with 10% chance.

As with the `tree-count-even` scenario, we evaluate empirical correctness by running our method alongside the base analysis and comparing the results. We start with the base Racket program and evolve it 1000 times, comparing the results at each iteration. The implementation successfully passed this test.

The metrics in this scenario are tracked differently than in the `tree-count-even` one. Instead of tracking the number of individual nodes accessed in the input, we instead keep track of the number of times we access nodes of an expression in the input data at each round. This is done since a node in an s-expression may be accessed multiple times, or may have different components accessed if it is a compound expression. Similarly to the `tree-count-even` scenario examined earlier, however, we expect a lower number of accesses to correspond to a lower number of computations executing without reuse.

In Figure 5.3 we see that, in the base analysis, the amount of accesses to expression nodes remains approximately constant throughout the execution. Our method, however, starts out with the same amount of accesses as the baseline, however this sharply decreases on successive executions. This provides evidence that our method is able to accomplish the same analysis by accessing only a fraction of the nodes as the baseline analysis.

Figure 5.4 shows the running time of each iteration. The running time for the initial iteration with our analysis was too great to be shown alongside the other elements. We observed that the initial iteration in this scenario took 31.962 seconds on average, meaning that there is significant cost in running our method with an unpopulated cache. Despite that, once the cache was populated, the running time on subsequent iterations was significantly faster on our method than on the baseline analysis.

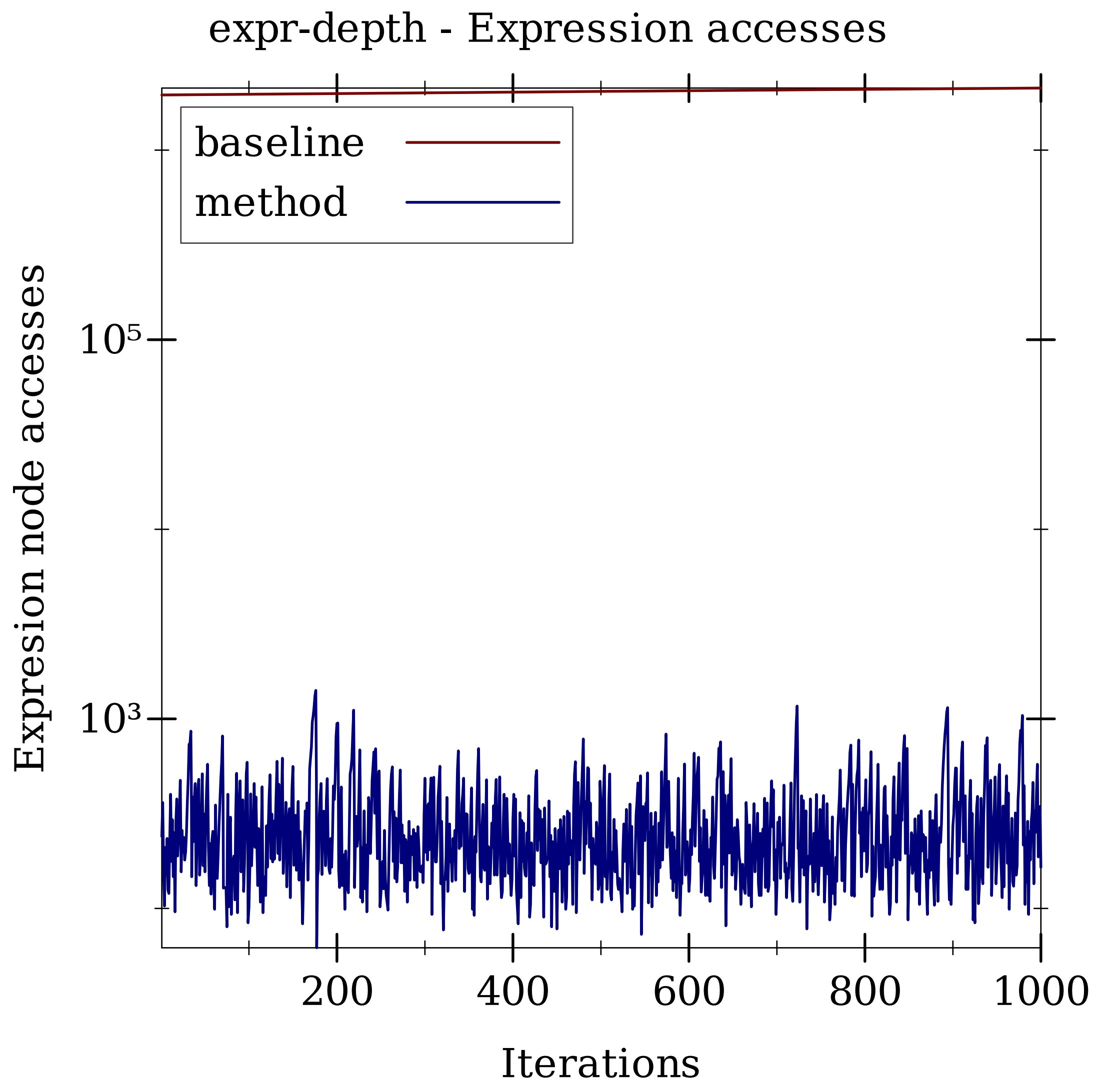

Figure 5.3: expr-depth evolution round vs. number of node accesses, method vs. baseline

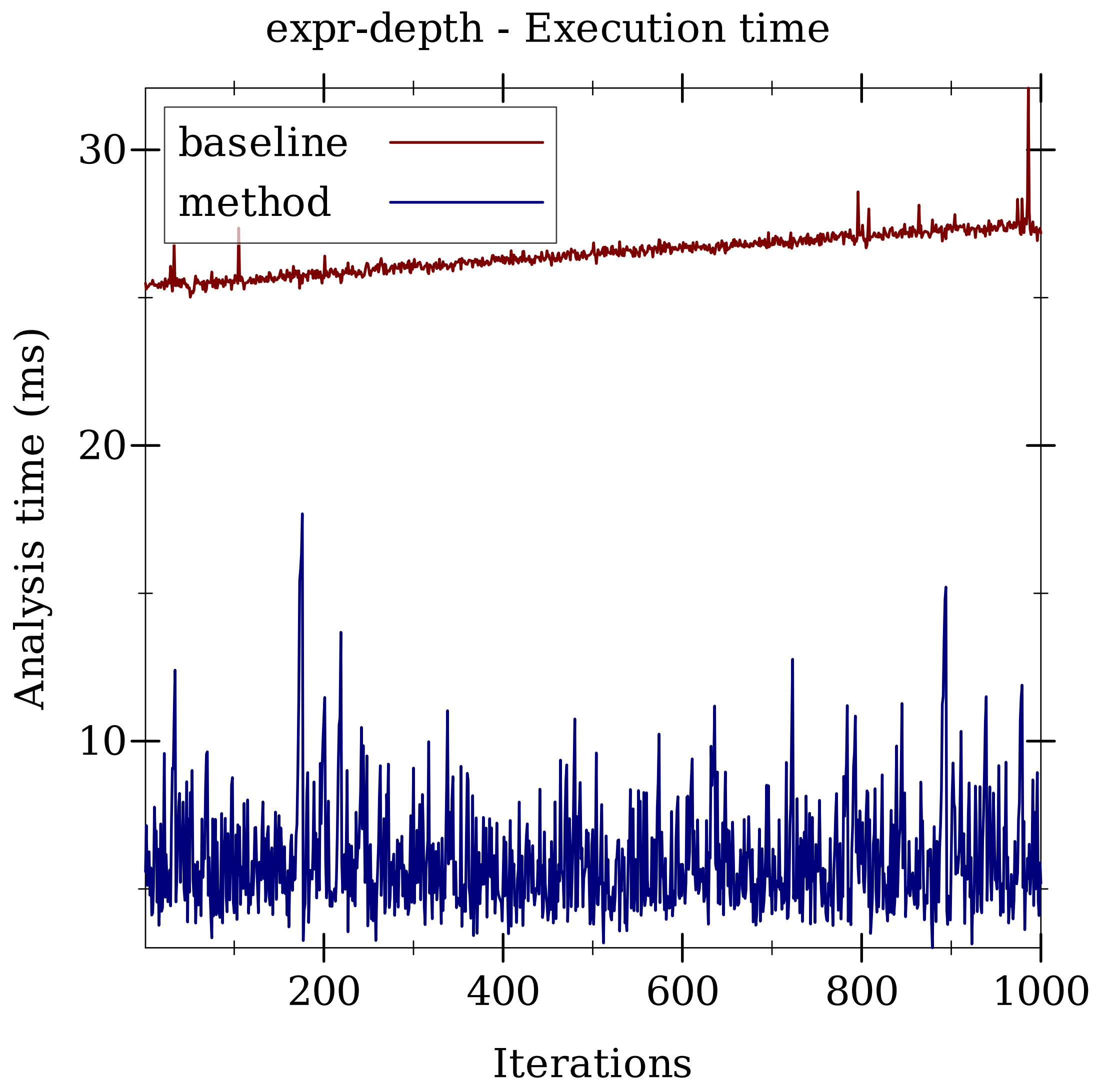

Figure 5.4: expr-depth Running time in milliseconds per iteration, method vs. baseline

Chapter 6

Conclusion

6.1 Summary

We have described a metaprogramming method to provide computation of analysis on tree-shaped data structures with reuse. The method consists of a metainterpreter written in a metalanguage operating on top of a base interpreter of an object language in which an analysis is written. Our method requires no substantial modification to the code of the original analysis, and is able to achieve great amounts of reuse between successive executions.

6.2 Limitations

The exploratory nature of this work predisposes it to some shortcomings. The initial execution of the evolution-aware analysis, in which the cache is empty, is exceedingly slow. This process could be alleviated by running the initial analysis with a memoized version of the procedure. There exist automatic memoization techniques that can make this process viable [10]. However, the reliance on automatic memoization may make the technique redundant. Therefore, we need to further evaluate the potential benefits of the symbolic approach taken in this work to exploit these to a higher extent.

Another shortcoming is due to the limitation imposed on the manner on which the data structure is traversed. The limitation to stateless tree-walk analyses is a key concern to address for large-scale real-world analyses.

6.3 Related Work

There has been significant work in dealing with evolution-aware analysis across many disciplines, with varying degrees of automaticity and generality. In the vein of transforming

existing analyses to deal with evolution, Arzt et al. [11] describes a technique for performing data-flow analysis in inputs that change incrementally, named REVISER. This technique operates on analysis described with the IFDS/IDE frameworks, in which a user describes a data-flow analysis as a set of templates. In similar spirit to our method, REVISER operates on an analysis that has been executed before, updating the results where necessary due to a change. When applying evolutions, instead of using evolution primitives, REVISER relies on a *structural diff* in order to pinpoint the nodes affected by the change. Sundaresh and Hudak [4] describe a general approach based on partial evaluation to deal with incremental functions. As with our approach, theirs take into consideration changes in the input in order to recompute only parts that have been affected. While our approach is restricted to recursive functions on tree-shaped input data structures, theirs is able to deal with a wider variety of analyses. In the context of functional programming, Leather et al. [12] describe a method for incrementalizing folds on algebraic data types. A fold, or catamorphism, is a function that traverses a data structure while accumulating values. We can see the type of functions treated in this work as folds. In contrast to this work, however, theirs examines the passing of information from parent to children nodes (downwards incrementalization), from children to parent nodes (upwards incrementalization), and both ways simultaenously (circular incrementalization).

There are also techniques for automatically transforming existing analyses in a manner that deals with some orthogonal aspect, such as evolution or variability. Shahin and Chechik [13] describe a technique for automatically transforming plain analyses into variability-aware ones. A variability-aware analysis takes into consideration multiple configurations of a program prescribed by a set of configuration variables or flags, for instance C preprocessor directives. As when dealing with evolution, adapting analyses to deal with variability is time-consuming and error-prone. While our method is able to transform a plain analysis to deal with evolution, theirs is able to transform an existing analysis to deal with variability.

Other approaches for dealing with software evolution include Rothenberg et al. [14], which describes a automata-based approach to perform incremental verification, and Trostanetski et al. [15], which describes an approach to detect *semantic differences* in imperative programs. This approach works on programs that arise from a patching process, such as when a program undergoes evolution.

Concerns in the usability of development environments have also driven the development of evolution-aware techniques for providing better support for techniques such as syntax highlighting. The Tree Sitter parser generator tool[1] is able to deal with incremental changes in a program's source code to provide incremental updating of the abstract syntax

[1] https://tree-sitter.github.io/tree-sitter/

tree. This way, text editors are able to verify the syntax of a program as the user types without slowing down, for instance. Tree sitter is based on the research of Wagner [16], which describes a set of algorithms and data structures used in the construction of an incremental software development environment.

6.4 Future Work

The metaprogramming technique described in this work requires little annotations in the analysis that is transformed. There may be further work required in determining to which extent these annotations can be performed automatically, and in which degree we may change the amount of computations of the original function performed by the metainterpreter. This may include work alongside the limitations in expressivity of the analyses handled by our method, as well as studies on the termination properties of such computations.

The field of Software Product Line Engineering (SPLE) studies software in which variability occurs [17][18][19]. Whereas software evolution can be seen as *variability in time*, the field of SPLE has focused mostly with *variability in space*. As such, there is a concern with dealing with variability in both dimensions [20][21]. We plan to extend the framework developed in this work could be used to deal with variability in space as well as time. One possible strategy is to integrate it into a recent automatic lift for variability [13].

Finally, Racket was chosen as the implementation language in part due to its simple syntax and interactive environment, allowing us to prototype quickly. We believe, however, that there is significant potential in exploring a purely functional solution in a language such as Haskell [22]. Haskell is widely used for developing DSELs [9]. Besides that, there are elegant functional abstractions for exploring the main elements of this work, such as pattern matching for analysis and transformation of data structures, generic programming with the Scrap Your Boilerplate design pattern for applying transformations and traversing data structures [23], and zippers to provide the concept of location within a data structure [24][25].

References

[1] Harman, Mark and Peter O'Hearn: *From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis.* In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23, 2018. 1, 2

[2] Ranta, A.: *Implementing Programming Languages.* College Publications, 2012. 1

[3] Møller, Anders and Michael I. Schwartzbach: *Static program analysis*, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`. 1

[4] Sundaresh, R. S. and Paul Hudak: *A theory of incremental computation and its application.* In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, page 1–13, New York, NY, USA, 1991. Association for Computing Machinery, ISBN 0897914198. `https://doi.org/10.1145/99583.99587`. 2, 36

[5] McCarthy, John: *History of LISP.* In *History of programming languages*, pages 173–185. Association for Computing Machinery, New York, NY, USA, June 1978, ISBN 978-0-12-745040-7. `https://doi.org/10.1145/800025.1198360`, visited on 2022-05-01. 5

[6] Mitchell, John C.: *Concepts in Programming Languages.* Cambridge University Press, 2002. 5

[7] Abelson, Harold and Gerald Jay Sussman: *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, July 1984, ISBN 978-0-262-01077-1. 5

[8] Marques, Maíra, Jocelyn Simmonds, Pedro O. Rossel, and María Cecilia Bastarrica: *Software product line evolution: A systematic literature review.* Information and Software Technology, 105:190–208, 2019, ISSN 0950-5849. `https://www.sciencedirect.com/science/article/pii/S0950584918301848`. 8, 13

[9] Hudak, Paul: *Modular domain specific languages and tools.* In *Proceedings of the Fifth International Conference on Software Reuse, ICSR 1998, Victoria, BC, Canada, June 2-5, 1998*, pages 134–142, 1998. `https://doi.org/10.1109/ICSR.1998.685738`. 14, 37

[10] Wimmer, Simon, Shuwei Hu, and Tobias Nipkow: *Verified memoization and dynamic programming.* In Avigad, J. and A. Mahboubi (editors): *Interactive Theorem Proving (ITP 2018)*, volume 10895, pages 579–596, 2018. 35

[11] Arzt, Steven and Eric Bodden: *Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes.* In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 288–298, New York, NY, USA, May 2014. Association for Computing Machinery, ISBN 978-1-4503-2756-5. `https://doi.org/10.1145/2568225.2568243`, visited on 2022-05-01. 36

[12] Leather, Sean, Andres Löh, and Johan Jeuring: *Pull-ups, push-downs, and passing it around - exercises in functional incrementalization.* In Morazán, Marco T. and Sven-Bodo Scholz (editors): *Implementation and Application of Functional Languages - 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2009. `https://doi.org/10.1007/978-3-642-16478-1_10`. 36

[13] Shahin, Ramy and Marsha Chechik: *Automatic and Efficient Variability-Aware Lifting of Functional Programs.* arXiv:2010.00697 [cs], October 2020. `http://arxiv.org/abs/2010.00697`, visited on 2022-05-01, arXiv: 2010.00697. 36, 37

[14] Rothenberg, Bat Chen, Daniel Dietsch, and Matthias Heizmann: *Incremental verification using trace abstraction.* In Podelski, Andreas (editor): *Static Analysis*, pages 364–382, Cham, 2018. Springer International Publishing, ISBN 978-3-319-99725-4. 36

[15] Trostanetski, Anna, Orna Grumberg, and Daniel Kroening: *Modular demand-driven analysis of semantic difference for program versions.* In Ranzato, Francesco (editor): *Static Analysis*, pages 405–427, Cham, 2017. Springer International Publishing, ISBN 978-3-319-66706-5. 36

[16] Wagner, Tim A.: *Practical Algorithms for Incremental Software Development Environments.* PhD thesis, EECS Department, University of California, Berkeley, Mar 1998. `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/5885.html`. 37

[17] Clements, Paul and Linda Northrop: *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2001. 37

[18] Apel, Sven, Don Batory, Christian Kästner, and Gunter Saake: *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, 2013. 2, 37

[19] Linden, Frank J. van der, Klaus Schmid, and Eelco Rommes: *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer-Verlag, 2007. 37

[20] Thüm, Thomas, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer: *Towards efficient analysis of variation in time and space.* In *Proceedings of the 23rd International Systems and Software Product Line Conference, Volume A*, SPLC, 2019. 37

[21] Krueger, Charles W.: *Variation management for software production lines.* In *Software Product Lines, Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22, 2002, Proceedings*, pages 37–48, 2002. `https://doi.org/10.1007/3-540-45652-X_3`. 37

[22] Marlow, Simon *et al.*: *Haskell 2010 language report.* Available online http://www.haskell. org/(May 2011), 2010. 37

[23] Lämmel, Ralf and Simon Peyton Jones: *Scrap your boilerplate: A practical design pattern for generic programming.* SIGPLAN Not., 38(3):26–37, jan 2003, ISSN 0362-1340. `https://doi.org/10.1145/640136.604179`. 37

[24] Huet, Gérard P.: *The zipper.* J. Funct. Program., 7(5):549–554, 1997. `http://journals.cambridge.org/action/displayAbstract?aid=44121`. 37

[25] Adams, Michael D.: *Scrap your zippers: A generic zipper for heterogeneous types.* In *WGP '10: Proceedings of the 2010 ACM SIGPLAN workshop on Generic programming*, pages 13–24, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0251-7. 37