Worst Case Resource Analysis in Program Synthesis

Tomer Ghelber
Worst Case Resource Analysis in Program Synthesis

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Tomer Ghelber

Submitted to the Senate of the Technion — Israel Institute of Technology Iyar
5780 Haifa May 2020
This research was carried out under the supervision of Prof. Shachar Itzhaky, in the Faculty of Computer Science.

Acknowledgements

I would like to thank my advisor, my family and my friends.

The Technion’s funding of this research is hereby acknowledged.
## Contents

List of Figures

<table>
<thead>
<tr>
<th>Abstract</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abbreviations and Notations</td>
<td>3</td>
</tr>
</tbody>
</table>

1 Introduction | 5 |

2 TransCal Rewrite System | 11 |

2.1 Program Expression Graphs (PEGs) | 11 |

2.2 TransCal’s E-PEG Data Structure | 12 |

2.3 Rewrite Rules in TransCal | 14 |

2.4 TransCal’s Actions | 15 |

3 Complexity Analysis Through Rewrite | 19 |

3.1 Simple rules of WCRA | 19 |

3.2 Analyzing function’s WCRA | 21 |

4 Comparing WCRAs | 27 |

4.1 Comparison Rules | 27 |

4.2 Reconstructing the best program | 29 |

5 Empirical Evaluation | 31 |

5.1 Case Study | 31 |

6 Conclusion | 37 |

A Rewrite rules used in this paper | 39 |

A.1 Program Expression Rewrite rules | 39 |

A.2 Space complexity rewrite rules | 40 |

A.3 Time complexity rewrite rules | 40 |

Hebrew Abstract | i |
List of Figures

1.1 Performance graph example .............................................. 6

2.1 E-PEG example which represents the expression \texttt{filter \ p \ (map \ f \ (x::xs))}
and also the (equivalent) expression \texttt{filter \ p \ ((f \ x)::(map \ f \ xs))} . 12

2.2 E-PEG example which represents \(a \ast 0\) before multiplication rule application 13

2.3 E-PEG example which represents \(a \ast 0 = 0\) after multiplication rule
application ................................................................. 13

2.4 Left: Rewrite rule’s premise. Right: Rewrite rule’s conclusion. .... 15

2.5 Rewriting steps in the development of the example program \texttt{nodup} . . 18

3.1 Inferring the time complexity of the body of \texttt{map} using WCRA rules .. 23

3.2 Inferring the time complexity of the body of \texttt{filter} using WCRA rules 24

5.1 Inferring the time complexity of the body of \texttt{map} using WCRA rules .. 32

5.2 Inferring the time complexity of the body of \texttt{filter-map} using WCRA
rules. ................................................................. 33

5.3 Inferring the time complexity of the body of \texttt{nodup} using WCRA rules . 35
Abstract

Every day there are millions, and maybe more, lines of code written in the world. It would be a good practice, to be able to verify and optimize each program but unfortunately, it cannot be done automatically, due to inherent undecidability issues.

What can be done is to create some rules that will allow for optimization approximation. The first step is to have a compiler that can generate provably equivalent implementations from a given program. The second step is to use static analysis techniques to estimate time and space requirements.

In order that the static analysis check will be effective, it is required to estimate big-O notation time and space complexity, as well as the constants involved. For this purpose, we use the notion of counting high-level instructions that are executed by the program. A default procedure for minimizing this measure, is to create a set of program variations and check each one of them with a static analysis tools for deciding the best one.

Our goal is to make this procedure more efficient, thus allowing us to automate it. We construct a system of transformation rules for creating program variations and then allow it “freedom of choice” in selecting the best variation of the program. In this process the system can be assisted by a set of tools, including the static analysis mentioned before. A naive approach would not work in this case, since even for a small set of transformations, generating the entire space of equivalent programs is intractable.

This thesis suggests a more practical way for getting the best time efficient program, in many cases, based on a set of rules that will generate the required information for selecting the best variations of a program. We introduce a semi-automated system based on a pure functional language called TransCal. We describe the system and underlying structures used by it, and present actions that a user can execute. The data structure used by TransCal is an E-PEG, a hyper-graph that holds all the programs found in the search, in a compact representation that is sublinear in the number of programs.

Using different kinds of rules for transformation and for analysis, we separate the different concerns and avoid unnecessary coupling that would complicate the procedure. For that purpose, we will create targeted rules for performing Worst-Case Runtime Analysis (WCRA) on top of existing rewrite rules, without modifying the underlying transformations, and thus preserving the semantic equivalence guaranteed by them. Thanks to that methodology, the original system is modular and extensible, and new
rules can be added to support different data structures and algorithm libraries.
Abbreviations and Notations

- **Transcal** : our rewrite system
- **E-PEG** : Equiv Program Expression Graph
- **WCRA** : Worst Case Resource Analysis
- **ctime** : Complexity time logic function
- **cspace** : Complexity space logic function
Chapter 1

Introduction

Computer programs are hand written, and as such they are prone to errors and requires a lot of human, expensive, time to tune up until they are optimized. Furthermore, each programmer has its own style of writing and there are many ways to get to the same results, some of them more effective than others. Due to that, the quality may differ and there is a need for a rule based automatic process to unify the styles and improve the quality. There is a lot of research done in this field of program optimization, most of it around improving the compiler output to produce the best effective code.

Three competing technologies (Figure 1.1) can be used to obtain worst-case execution times: experimental (or testing-based) approaches, probabilistic measurement and static analysis. Experimental approaches determine worst-case execution costs by (repeated and careful) measurement of real executions, using either software or hardware monitoring. While they may give good estimates of actual execution costs, they cannot usually guarantee upper bounds on execution cost. Probabilistic approaches build on experimental measurements by measuring costs for repeated executions over a suite of test cases. Under the assumption that the test suite provides representative data, it is then possible to construct statistical profiles that can be used to determine worst-case execution time to some stated probability. Absolute guarantees cannot, however, be provided. Finally, static analysis approaches construct detailed and precise models of processor instruction timings to be able to predict worst-case timings. This typically involves constructing accurate models of the processor state, including cache and pipeline information. The primary advantage of measurement or probabilistic approaches is that they may be applied to arbitrary computer architectures, without detailed knowledge of the underlying design, and using relatively unsophisticated timing techniques. In contrast, static analyses require detailed architectural knowledge and painstaking effort to construct. Moreover, some architectural features, such as Pseudo-LRU replacement policies for caches present specific difficulties. However static analysis approaches provide the only guaranteed bounds of worst-case execution time and are therefore to be preferred for use in safety-critical or mission-critical systems.

However, optimizing the instructions are not enough. In a program there are different
paths for control flow and data which may trigger different costs of the same instructions. Therefore, when optimizing instructions approach is taken, it is best to optimize the binary code after initial compilation and not the source code. An example of such approach is an object code optimizer, sometimes also known as a post pass optimizer or, for small sections of code, peephole optimizer. The optimizer takes the output from a source language compile step - the object code or binary file - and tries to replace identifiable sections of the code with replacement code that is more algorithmically efficient (usually improved speed).

- “IBM Automatic Binary Optimizer for z/OS” (ABO) was introduced in 2015 as a cutting-edge technology designed to optimize the performance of COBOL applications on IBM Z mainframes without the need for recompiling source. It uses advanced optimization technology shipped in the latest Enterprise COBOL. ABO optimizes compiled binaries without affecting program logic. As a result, the application runs faster but behavior remains unchanged so testing effort could be reduced. Clients normally don’t recompile 100 percent of their code when they upgrade to new compiler or IBM Z hardware levels, so code that’s not recompiled wouldn’t be able to take advantage of features in new IBM Z hardware. Now with ABO, clients have one more option to reduce CPU utilization and operating costs of their business-critical COBOL applications.

- The earliest “COBOL Optimizer” was developed by Capex Corporation in the mid-1970s for COBOL. This type of optimizer depended, in this case, upon knowledge of ‘weaknesses’ in the standard IBM COBOL compiler, and actually replaced (or patched) sections of the object code with more efficient code. The replacement code might replace a linear table lookup with a binary search for example or sometimes simply replace a relatively slow instruction with a known faster one that was otherwise functionally equivalent within its context. This
technique is now known as strength reduction. For example, on the IBM/360 hardware the CLI instruction was, depending on the particular model, between twice and 5 times as fast as a CLC instruction for single byte comparisons.

The main advantage of re-optimizing existing programs was that the stock of already compiled customer programs (object code) could be improved almost instantly with minimal effort, reducing CPU resources at a fixed cost (the price of the proprietary software). A disadvantage was that new releases of COBOL, for example, would require (charged) maintenance to the optimizer to cater for possibly changed internal COBOL algorithms. However, since new releases of COBOL compilers frequently coincided with hardware upgrades, the faster hardware would usually more than compensate for the application programs reverting to their pre-optimized versions (until a supporting optimizer was released).

There is a correlation between how common or critical the code is, such as those found in safety-critical domains or libraries, to the amount of time spent on optimization. In this situation a common way for optimization is using a Data-flow optimization. Data-flow optimizations, based on data-flow analysis, primarily depend on how certain properties of data are propagated by control edges in the control flow graph. Some of these include:

- **Common subexpression elimination**: In the expression \((a + b) - (a + b)/4\), "common subexpression" refers to the duplicated \((a + b)\). Compilers implementing this technique realize that \((a + b)\) will not change, and so only calculate its value once.

- **Constant folding and propagation**: replacing expressions consisting of constants (e.g., \(3 + 5\)) with their final value (8) at compile time, rather than doing the calculation in run-time. Used in most modern languages.

- **Induction variable recognition and elimination**: For example, in the code:

  ```c
  int c, i;
  c = 10;
  for (i = 0; i < 10; i++) {
    c = c + 5;
  }
  ```

  the variable \(i\) can be eliminated and the new code would be:

  ```c
  int c, i;
  c = 10;
  for (i = 0; i < 10; i++) {
    c = 10 + 5 * (i + 1);
  }
  ```
- Alias classification and pointer analysis: in the presence of pointers, it is difficult to make any optimizations at all, since potentially any variable can have been changed when a memory location is assigned to. By specifying which pointers can alias which variables, unrelated pointers can be ignored.

- Dead store elimination: removal of assignments to variables that are not subsequently read, either because the lifetime of the variable ends or because of a subsequent assignment that will overwrite the first value.

Static analysis tools refer to a wide array of tools that examine source code, executables, or even documentation, to find problems before they happen; without running the code. These tools vary greatly in scope and purpose, ranging from compiler-level checks for logical errors, to code styling enforcement, to cloud-based suites of tools that cover everything from documentation formatting to code complexity analysis. Put simply, you could think of static analysis tools as anything that helps you maintain a healthy code base without having to run that code.

Static analysis methods compute safe upper bounds of the execution time from a mathematical model of the target architecture. Dynamic analysis methods, on the other hand, derive the execution time from measurements performed on real hardware. Hybrid methods, like our approach, combine execution time information extracted from measurements with statically computable information like control flow graphs to improve safety, precision and/or coverage of the result. Probabilistic methods, finally, try to compute statistical models from measurements to compute upper bounds of the execution time.

By using a static analysis tool one can check both the code’s time efficiency and space requirements. Although a static analysis tool does not check for complexity, it does go into the constant’s resolution. For measuring the time efficiency, a static analysis tool counts the number of high-level instructions needed for the code using an instruction counter [BFHH06] [FH04]. There is no easy way for determining the most effective code. A naive and simple process to get the optimized code is by creating all the different programs which map all inputs to the same outputs and hence are semantically equivalent [LS07]. Then each program is checked with static analysis tools which uses compilers for rewriting the code according to rules a better optimized program. This process repeats itself until no better optimized program can be rewritten. Definitions of the semantically equivalence rewrites, the rewrite rules, enable to alter a given code to a different one yet keep its correctness. By utilization of rewrite rules multiple correct variations can be produced automatically from one program. The programs’ space required by the rewrite systems is huge – it may reach to a double exponent the depth of the original program. The generation process semantically equivalence programs does not produce all possible. For getting the best time efficient program, the generated programs can then be compared for best execution results using a static analysis tools for choosing the best one. The large number of feasible programs to be examined makes
this process unpractical. Furthermore, there is no guarantee that the best performance program is reached even after all possible variations are tested.

There is a need for a better way to find the best equivalent code, in terms of instruction counter efficiency. First, by using rewrite rules one can take a sub-expression of the equivalent code and examine only the reachable programs with finite number of rewrites. For enabling the access to the instruction counter, this thesis proposes the addition of instruction counter rules to the same set of rewrite rules that are used to rewrite terms. In this case rules are activated and incrementally finds each program’s instruction counter, very similar to dynamic programming methods. This process results in reaching the best time-consuming program since the existing system’s structure is used. A learning process is established as the previous results are saved and serve as an input to the next analysis. This saving ability helps in not activating the same rule twice on the code even if the code is found by different paths. The process is semiautomatic as it needs in certain point user intervention. However, the user is given some guidance for making the right choices.
Chapter 2

TransCal Rewrite System

In this chapter, we introduce a semi-automated system based on a pure functional language called TransCal. We describe the system and underlying structures used by it. We then present actions the user can execute.

2.1 Program Expression Graphs (PEGs)

In a traditional compilation system, optimizations are applied sequentially, with each optimization taking as input the program produced by the previous one. This traditional approach to compilation has several well-known drawbacks. One of these drawbacks is that the order in which optimizations are run affects the quality of the generated code, a problem commonly known as the phase ordering problem. Another drawback is that profitability heuristics, which decide whether to apply a given optimization, tend to make their decisions one optimization at a time, and so it is difficult for these heuristics to account for the effect of future transformations.

The approach used for structuring optimizers is based on the idea of having optimizations propagate equality information to a common IR that simultaneously represents multiple optimized versions of the input program. The main challenge in designing this IR is that it must make equality reasoning effective and efficient.

The representation selected for computations is called Program Expression Graphs (PEGs). PEGs are referentially transparent, which intuitively means that the value of an expression depends only on the value of its constituent expressions, without any side-effects. As has been observed previously in many contexts, referential transparency makes equality reasoning simple and effective. However, unlike previous SSA-based [VdBHKO02] [TP95] representations, PEGs are also complete, which means that there is no need to maintain any additional representation such as a Control Flow Graph (CFG) [TSTL08]. Completeness makes it easy to use equality for performing transformations: if two PEG nodes are equal, then we can pick either one to create a program that computes the same result, without worrying about the implications on any underlying representation.
The main challenge is that each additional equality can potentially double the number of represented programs, thus making the number of represented programs exponential in the worst case. To address this challenge, the PEG nodes merged into equivalence classes (as shown at Figure 2.1). The resulting equivalence graph is called an E-PEG [TSTL09], and it is this E-PEG representation that we use in our approach. Using equivalence classes allows E-PEGs to efficiently represent exponentially many ways of expressing the input program, and it also allows the equality saturation engine to efficiently consider previously discovered equalities.

2.2 TransCal’s E-PEG Data Structure

The data structure used by TransCal is an E-PEG: a hyper-graph that holds the set of all programs found by starting from a given program and applying semantics-preserving rewrites. $E$-$PEG = (V, E)$, where $V$ is a set whose elements are called vertices, and $E$ is a set of groups of three - a list of 0 or more vertices $SOURCES$, a label $L$ and a vertex, whose elements are called hyperedges. The vertices in $V$ are labeled arbitrarily (in the examples of this section, a vertex is identified by a number), each representing a set of equivalent expressions. Hyperedges in $E$ are directed, ordered, and their label $L$ are symbols from a vocabulary of function names, variable names, or literal constants. When an edge’s $SOURCES$ is not empty, it means that its label $L$ is a function and the vertices in $SOURCES$ are the arguments passed to it. When an edge’s $SOURCES$ is empty, it represent an atom (a program variable or a literal) and the label $L$ is the name of the atom. Parallel edges with the same sources, targets, and label are redundant and only one instance is kept.

The addition E-PEG adds over PEG, the equivalence part, is the property that some expressions represented by the same vertex are interchangeable in every context.
For example, in Figure 2.3, the vertex 1 represents both \(a \cdot 0\) and 0, which are obviously equivalent due to the fundamental rules of arithmetic.

**Compaction.** Edges with the same sources and label but different targets signify that their targets are in fact equivalent expressions, since they represent pure functions. Our E-PEG has a changed definition for equivalent edges to contain only the the SOURCES and the L. With this change we can facilitate compaction and merge more edges together to eliminating any resulting duplicate edge. Whenever edges are added to the graph, TransCal searches the existing edges for an equivalent edge, that is, having the same sources and label as one of the newly inserted ones. When found, TransCal eagerly merges their targets.

**Example 2.2.1.** One hyperedge is \([\text{ID2, ID3, ...}] \rightarrow \text{ID0}\) and another one with the same source but different target is \([\text{ID2, ID3, ...}] \rightarrow \text{ID1}\). ID1 and ID0 expressions are equivalent and will be merged to keep the hypergraph compacted. Merging means replacing ID1 with ID0 whenever it occurs as a source or a target of an existing edges.
Those could have a cascading effect as more edges may now become compatible, inducing further merges.

**Versioning.** TransCal applies transformations iteratively, so that the E-PEG keeps changing all the time. To avoid expensive re-computation of rewrites that have already been applied, the E-PEG data structure is *versioned*, meaning that it comprises of “layers” of edges corresponding to iterations in which said edges were created. Thus, when performing pattern matching on the E-PEG, a process that is explained later in Section 2.3, it is sufficient to consider only matches containing at least one edge that was introduced later than the last time the same pattern was matched. Any matches comprising of “old” edges have already been treated. This allows for efficient incremental application of term rewrite rules when E-PEGs grow large.

E-PEGs are very compact to begin with, and our compaction helps then stay that way, reducing maintenance overhead and search cost. The versioning helps us dig into the right layer for new information and reduce the number of places we search for patterns. Those patterns will be used as the basis of the rewrite rules, which make the smallest and most critical unit of computation and should therefore be optimized to be as fast as possible.

### 2.3 Rewrite Rules in TransCal

In every rewrite system, the rewrite rules are the most basic and import part. A rewrite rule is built from two parts — a premise, which is a term to look for, and a conclusion, which is a term being generated as a result. In general, the premise term will contain holes, that can match any sub-term, so that a single rule can match (infinitely) many terms. The same holes can be used in the conclusion to create a new expression based on each match, insert it into the pool of programs represented by the E-PEG. For example, rules for simplifying multiplication with 0 and 1 would be: \( \text{hole}_1 \ast 0 \rightarrow 0; \text{hole}_1 \ast 1 \rightarrow \text{hole}_1 \). In the former, \( \text{hole}_1 \) is ignored as it is inconsequential to the result. In the latter, the result is \( \text{hole}_1 \) itself, meaning that the root (representing \( \text{hole}_1 \ast 1 \)) should be merged with its left predecessor (representing \( \text{hole}_1 \)).

In our system, the rewrite rules would look for a pattern in the E-PEG, and insert the new edges and nodes to the graph. We translate the wanted program to a graph’s pattern in a similar way we translate every program for the E-PEG, except we have holes and explicit values. We use the translated pattern to search the E-PEG. The matched holes would be used in the conclusion to create new edges and nodes to insert the E-PEG. The example in Figure 2.4 would create two edges in the premise (left side) – one for \( \ast \) and with two entries and one for ‘0’ with zero entries which it’s target is the second entry in the first edge. The conclusion (right side) uses the same entries to match the premise.

When a pattern matches the premise, the hole 1 of in the nodes would be used to
Another important ability is not to find with the premise something the rewrite rule already found. Thanks to the layers in our E-PEG, our rewrite rules can search only for new information. If nothing new was inserted that is relevant to any one of the current rewrite rule’s premise, nothing new will be created, and the search will be done faster. Then the rewrite rule can remember what was the last layer it searched.

2.4 TransCal’s Actions

The system uses its efficient underline structures to find the satisfying program. To find it, the system can activate different number of actions that are pre-programs and entered by the developers of the system. Those actions are the main API and revealed to the users, and the most important part of the system’s search. These actions provide different capabilities which assist in program rewriting until a satisfying program is achieved. The fundamental actions that are used to add a new rewrite rule or add a new program are Let and Def. The actions are used for structure interacting, defining new rewrite rules. Below we describe the actions:

Let. Creates a rewrite rule, e.g. \(0 \ast \_ \rightarrow 0\).

A single rule is always directed; to allow rewriting in both directions, TransCal would add two rules, which can be abbreviated by using an equal sign = instead of →: \(a \ast (b+c) = a \ast b + a \ast c\) is equivalent to \(a \ast (b+c) \rightarrow a \ast b + a \ast c\) and \(a \ast b + a \ast c \rightarrow a \ast (b+c)\).

Def. Like ‘Let’, but also adds the right-hand side term to the E-PEG, allowing subsequent actions to operate on it.

It is denoted as e.g. \(f \ x \rightarrow x + 2\ [++]\), and similarly has a bidirectional variant, \(f \ x = x + 2\ [++]\).

Locate. Locates a pattern in the program and gives it a new simple name, an anchor, which will be used to reference it in future actions. It is denoted by using special names.
that designate anchors as opposed to regular program identifiers, e.g \( x \ast \_ \rightarrow \_ \). The anchor \( \_ \) then refers to the matched term in the program.

**Elaborate.** Takes a term to rewrite and a pattern to find in the search space of terms that are equivalent to it via the rewrite rules. To find the pattern, it activates all the rewrite rules on the program until a pattern is located or it reaches a point of saturation, in which no new rewrites can be applied to the E-PEG.

A single application of **Elaborate** may trigger numerous rules, and it tries to apply them exhaustively, continuously testing for the goal pattern. The input to the action does not include which rules to apply; the action will try all available rules, essentially in a breadth-first manner. The user does not even have to be aware of all the rules that exist at the particular point in time when the action is invoked.

For example, \( x \ast (n + 3) \rightarrow \_ + \_ \) would elaborate the term to \( x \ast n + x \ast 3 \). In this case, the action resulted in applying the rewrite rule \( a \ast (b + c) \rightarrow a \ast b + a \ast c \). The holes in the pattern can be filled by any term generated during the rewrite. Of course, there may be more than one match, in which case **Elaborate** stops on the first match.

**Generalize.** Takes an expression and extracts from it a new function and a corresponding rewrite rule. In software engineering, this action is called “extract to a method”; the new function is obtained by replacing some of the constants occurring in the expression with bound variables, and these variables become the function’s arguments.

As an example, the action \( n \ast (n + 1) \rightarrow f n \) creates the function \( f := \lambda n. n \ast (n + 1) \). Along the same lines, and by employing a few rewrites along the way, \( n \ast (n+1) \rightarrow f (n+1) \) creates \( f := \lambda n. (n - 1) \ast n \).

Variables that are declared in an outer scope relative to the extracted section are added as extra arguments (these are often called closure arguments). This action is very useful in the process of finding similar sections and transform them into recursive sections as well as for other important purposes. Another instance of **Generalize** is shown in the next example.

**Example 2.4.1.** We will demonstrate the actions by applying them to an example program. The program is called nodup. It checks if a list contains distinct values, meaning that no value occurs in it more than once. Its type signature is \( \text{nodup} : \forall A.\ list\ A \rightarrow bool \), and its full definition is given by:

\[
\text{nodup} \ l = \text{match} \ l \ with \\
| [] \Rightarrow \text{true} \\
| x :: xs \Rightarrow \neg (\text{elem} \ x \ xs) \land \text{nodup} \ xs
\]

We will focus on the second branch, which is more interesting and relevant to optimize. We start with the given definition at line 1 of Figure 2.5 and apply TransCal actions iteratively. Next to **Elaborate** actions, the rewrite rules that were employed in the elaboration are listed, referring to the list of standard rules in Appendix A; numbers
in parentheses refer to rules that were defined by previous actions. The entire rewrite trace is also shown, for clarity.

The **Generalize** action (line 9) had the form $\{x\} \parallel \text{elems } xs \land \text{nodup } xs \rightarrow \text{nodup'} \{x\} \; xs$ and created a new function, **nodup'**, and its accompanying rule. The name $xs$ is preserved in the second parameter to **nodup'**, and the compound term $\{x\}$ is extracted as a new parameter, **acc**.

The result is a new auxiliary function, **nodup'**, which gets an accumulator of type set. This accumulator provides $O(1)$-time access to look up previous values, instead of the list function **elem**, which runs in time $O(n)$. As a result, **nodup'** runs in linear time, whereas the original implementation of **nodup** was quadratic. The revised program therefore invokes **nodup'** in the branch corresponding to the case $x::xs$. 


<table>
<thead>
<tr>
<th>Program being revised</th>
<th>Action</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 nodup (x::xs) = ¬elem x xs ∧ nodup xs</td>
<td></td>
<td>Def</td>
</tr>
<tr>
<td>2 ¬x ∈ elems xs ∧ nodup xs</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>3 x ∉ elems xs ∧ nodup xs</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>4 {x}</td>
<td></td>
<td>elems xs ∧ nodup xs</td>
</tr>
<tr>
<td>5 nodup' acc xs = (acc</td>
<td></td>
<td>elems xs) ∧ nodup xs</td>
</tr>
<tr>
<td>6 xs = x'::xs'</td>
<td></td>
<td>Let</td>
</tr>
<tr>
<td>7 {x}</td>
<td></td>
<td>elems (x'::xs') ∧ nodup (x'::xs')</td>
</tr>
<tr>
<td>8 {x}</td>
<td></td>
<td>elems (x'::xs') ∧ ¬elem x' xs' ∧ nodup xs'</td>
</tr>
<tr>
<td>9 {x}</td>
<td></td>
<td>elems (x'::xs') ∧ ¬x' ∈ elems xs' ∧ nodup xs'</td>
</tr>
<tr>
<td>10 {x}</td>
<td></td>
<td>elems (x'::xs') ∧ x' ∉ elems xs' ∧ nodup xs'</td>
</tr>
<tr>
<td>11 {x}</td>
<td></td>
<td>elems (x'::xs') ∧ {x'}</td>
</tr>
<tr>
<td>12 {x}</td>
<td></td>
<td>({x'} ∪ elems xs') ∧ {x'}</td>
</tr>
<tr>
<td>13 ({x}</td>
<td></td>
<td>{x'}) ∧ (x</td>
</tr>
<tr>
<td>14 {x'}</td>
<td></td>
<td>{x} ∧ ({x} ∪ {x'})</td>
</tr>
<tr>
<td>15 {x'}</td>
<td></td>
<td>{x} ∧ nodup' ({x} ∪ {x'}) xs'</td>
</tr>
<tr>
<td>16 x' ∉ {x} ∧ nodup' ({x} ∪ {x'}) xs'</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>17 nodup' acc (x'::xs') = x' ∉ acc ∧ nodup' (acc ∪ {x'}) xs'</td>
<td></td>
<td>Generalize</td>
</tr>
<tr>
<td>18 nodup' acc [] = true</td>
<td></td>
<td>Elaborate (5)</td>
</tr>
</tbody>
</table>

End result:

```plaintext
nodup l = match l with
  | [] => true
  | x :: xs => nodup' {x} xs

nodup' acc l = match l with
  | [] => true
  | x' :: xs' => ~(x' ∈ acc) /
                   nodup' (acc ∪ {x'}) xs'
```

Figure 2.5: Rewriting steps in the development of the example program nodup
Chapter 3

Complexity Analysis Through Rewrite

The previous chapter have set the theoretical foundation as well as the data structure that are in use when working with rewriting rules in TransCal System. This chapter discusses Worst Case Runtime Analysis by describing several examples that demonstrate its roles. In order to introduce the WCRA, there is a need to set two ground rules:

- WCRA rewrite rules will be added on top of the existing rewrite rules, using designated identifiers to refer to time and space complexity measurements.
- A new action will be used for analyzing the complexity of the programs in the space of equivalent programs described by an E-PEG, and extracting the best candidate.

3.1 Simple rules of WCRA

In TransCal, we separate the different dimensions of information to different rewrite rule to keep each one of them their capabilities. That way, different dimensions like computation, types and WCRA, allowing looser coupling between different concerns so that both the automated search and the user can focus on one of them at a time. For that purpose, we will create for every expression rewrite rule a new WCRA rewrite rule, leaving the original rule intact. Thanks to that methodology, the original system can work as usual and the new rules can be applied where necessary, contributing additional information when it is required.

While inserting the new rewrite rules for analyzing time complexity, we discovered that there is a need for information about space complexity as well. This is due to the fact that computations over lists, and other recursive data structures, usually depend on their sizes — where the size is best characterized in terms of memory allocation, or space. For more complex structures, such as trees, the set of rules may be extended with additional rules for other important measurements, such as their depth.
As an example, consider the function \texttt{elem} that was introduced in Figure 2.5, for searching in a list. As mentioned, this function runs in linear time. We therefore introduce the following rule:

\[
\text{ctime(} \texttt{elem x l} \text{)} = \text{ctime(} x \text{)} + \text{ctime(} l \text{)} + \text{cspace(} l \text{)}
\]

As shown, the time to search in a list for existence of element is depends on the list’s length, \text{cspace(} l \text{)}. To that we add the cost of computing the list itself, as well as computing the element to search for. For rewrite rules like this and many others, we added basic space-complexity rules for lists and constants.

All constants (like True, False, 0, 1) have a space complexity of 0:

\[
\text{cspace(} \texttt{false} \text{)} = \text{cspace(} \texttt{true} \text{)} = \text{cspace(} 0 \text{)} = \text{cspace(} 1 \text{)} = 0 \quad (3.1)
\]

From these, compound values can be constructed by means of \textit{algebraic data types}. Each data types has a list of \textit{constructors}, and a value is constructed by applying them, often recursively. Therefore, the rules will also be applied recursively to the arguments of each constructor, and the cost will be embodied in the rule associated with each constructor. We elected to characterize the space complexity of a data structure by the total number of constructor applications that are involved in its creation. Leaf (nullary) constructors are assigned zero space, and each data constructor is assigned one unit of space.

For example, lists are formed by the algebraic data type \texttt{list} that has two constructors: \texttt{nil ([])} and \texttt{cons (::)}. The singleton value, [], which is an empty constructor, is handled as a constant:

\[
\text{cspace(} [] \text{)} = 0
\]

The recursive constructor :: is counted for 1 unit plus the space occupied by its data arguments, recursively. Because of that, we added a space rewrite rule which depends on more space information:

\[
\text{cspace(} x::xs \text{)} = 1 + \text{cspace(} xs \text{)}
\]

Additionally, we added a general purpose space rewrite rule for list complexity space which depends on one of its properties - length:

\[
\text{cspace(} l: \texttt{list} \text{)} = \text{len(} l \text{)}
\]

This rewrite rule will help us to gain more information for other rewrite rules and connect already exists usage of lists because \texttt{len} is an existing function.

All of this information, the length and the connection between the components, are used both to analyze functions in Section 3.2 and to order programs in Chapter 4.

It should be considered that the rewrite rules for the WCRA have a performance time
which is the accumulation of the calculation performed by them and their parameters analysis time. The rewrite rules for the WCRA are built by their own calculation WCRA and their parameters WCRA. For example, the operation “add” (+) takes 1, but to that we need to add the time it takes to calculate each parameter. The example above of contains, depends its own time, the list’s size, and both x’s WCRA and l’s WCRA.

3.2 Analyzing function’s WCRA

In most cases “Simple WCRA’s logic” for describing the complexity involved in the domain of rewrite systems. Going step by step in the complexity level, this section is discussing the use of the WCRA for recursive functions for the worst-case scenario. WCRA of functions should be made as rewrite rules, so the appropriate place to create them is in an action - a more powerful tool to even create new rewrite rules. Those rewrite rules will exist side-by-side with the function’s rewrite rules for expressions.

To simplify our statement and examples we want handle bi-recursive functions but the TransCal system can be extended to handle it in the future. Additionally, this action handles lists with ”cons” as the only recursive value.

To start the process, the action needs to know the name of the function and the number of its arguments. In order to track the use of the recursive function a temporary \(ctime\) function is created, “F”, that includes all the arguments of the recursive function. In the future result, not all the argument will be used, but we might use them “F” describes the function’s WCRA. To evaluate the WCRA it will need some, if not all, the function’s parameters.

The “F” WCRA is assigned to both the function call and to any recursive call. To test that the rewrite process is performed, with the temporary “F” rewrite rule included, until no change in values is received. The expected results are to get a new equation between the function call and each of the recursive calls marked with “F”. By analyzing this equation we can create the temporary “F” rewrite rule, and from a description and definition of WCRA to create the real values and the different relation in the parameters. This analyzing is easy for non-recursive function, because the equation on one side holds “F” and on the other side the explicit relation of the parameters. The hard work in the analysis is changing a withdrawing rule to a simple rule. With mathematical tools which are used in regular static analysis we can solve any withdraw rule and get a simple rule. By data structure builders, we can understand that a list is getting smaller, and the stopping place would be on the singleton value NIL.

The first step of the algorithm (Algorithm 3.1) is to choose a fresh symbol. This symbol will be used for a function which return a \(ctime\) of the given definition’s header (left side of the equation). Then, we create a rewrite rule with the new symbol. The premise is the given definition’s header, and the conclusion is the symbol with all the arguments of the definition. This rewrite rules is added to all the rewrite rule which
Algorithm 3.1 Analyze Function’s WCRA

Input: a definition of the form \( f \, x_1 \, x_2 \, \cdots \, x_k = e \)

1. \( F := \) fresh function symbol
2. Added rewrite rule: \( f \, x_1 \, x_2 \, \cdots \, x_k \rightarrow F(x_1, x_2, \cdots, x_k) \)
3. Apply all available rewrite rules to \( e \).
4. Find \( ctime(e) \) in resulting E-PEG.
5. If not found, return “unknown”.
6. Let \( t = \) expression representing \( ctime(e) \) from E-PEG.
7. Create the recurrence: \( F(x_1, \cdots, x_k) = t \)
8. Solve recurrence.

Output: “unknown” or a definition in form \( ctime(f \, x_1 \, x_2 \, \cdots \, x_k) = ct \)

are executed all together until we reach a fixed point. After reaching a fixed point, the \( ctime \) of the right hand-side in the definition (the expression) is calculated. If it does’nt exist, no \( ctime \) can be calculated.

The \( ctime \) can be calculated from a recurrence. We solve simple cases of recurrence. Those cases are either constants and lists. If the recurrence is not recursive then it is a trivial case of constant \( ctime \). If the recurrence is recursive, we change plus to mul, and mul to exponent.

To understand this process, we will follow two examples of recursive functions involving list ADTs.

Example 3.2.1. Another basic example is \( \text{map} : \forall A \, B. \, (A \rightarrow B) \rightarrow \text{list } A \rightarrow \text{list } B \). Its definition is:

\[
\text{map } f \, l = \text{match } l \text{ with } \\
| [] \rightarrow [] \\
| x :: xs \rightarrow (f \, x) :: (\text{map } f \, xs)
\]

This definition is recursive, and there is one sub-call to \( \text{map} \) in the second branch of the expression. As a consequence, \( \text{map} \) has a linear run-time complexity, whenever its argument \( f \) has constant run-time complexity.

TransCal employs several rules to reach this conclusion about the run-time complexity of \( \text{map} \). The first step is to create a function \( F \) denoting the time complexity of a particular call to \( \text{map} \), and then compute a recurrence relation that will characterize it by analyzing the body of its definition. This analysis and the resulting conclusion are shown in Figure 3.1.

From the recursive equation it reached, the TransCal system will understand the non-recursive equation as it has done before using the list constructor \( \text{cons} \). The last
**Figure 3.1**: Inferring the time complexity of the body of `map` using WCRA rules

The equation in Figure 3.1 will be changed to:

\[ F(f, x::xs) = 2 + \text{ctime}(f, x) + F(f, xs) \]

Where the decreasing parameter of the recurrence is the second \((x::xs \text{ to } xs)\). Now, assuming some other rule for the complexity of a given \(f\), e.g. \(\text{ctime}(f, x) = c\) for some constant \(c\), we can solve the recurrence using the measure of space complexity that was defined for lists:

\[ F(f, l) = (2 + c) \cdot \text{cspace}(l) \]

Giving the time complexity for the body of `map`. The full \(\text{ctime}(\text{map } f \ l)\) will include also the time complexity of each argument for a total of—

\[ \text{ctime}(\text{map } f \ l) = \text{ctime}(f) + \text{ctime}(l) + (2 + c) \cdot \text{cspace}(l) \quad (3.2) \]

**Example 3.2.2.** The function `filter : \forall A. (A -> bool) -> list A -> list A` is canonically defined as follows:

```haskell
filter p l = match l with
  | [] => []
  | x :: xs => if (p x) then x :: (filter p xs)
  else filter p xs
```

This definition is recursive, and there are two sub-calls to `filter` in the second branch of the expression. However, only one of them is actually invoked in every
**Figure 3.2:** Inferring the time complexity of the body of \texttt{filter} using WCRA rules

<table>
<thead>
<tr>
<th>Inferred \textit{ctime}</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{ctime}(\text{match } l \text{ with } \ldots) = \max{\text{ctime}(\emptyset) + \text{ctime}(\emptyset),) (\text{ctime}(x::xs) + \text{ctime}(\text{if } (p \ x) \ldots)})</td>
<td>1</td>
</tr>
<tr>
<td>(\text{ctime}(l) = 0)</td>
<td>\textit{atom}</td>
</tr>
<tr>
<td>(\text{ctime}(\emptyset) = 0)</td>
<td>3</td>
</tr>
<tr>
<td>(\text{ctime}(x::xs) = 1 + \text{ctime}(x) + \text{ctime}(xs))</td>
<td>2</td>
</tr>
<tr>
<td>(\text{ctime}(x) = 0)</td>
<td>\textit{atom}</td>
</tr>
<tr>
<td>(\text{ctime}(xs) = 0)</td>
<td>\textit{atom}</td>
</tr>
<tr>
<td>(\text{ctime}(\text{if } (p \ x) \text{ then } x::(\text{filter } p \ xs) \text{ else } \text{filter } p \ xs) =)</td>
<td></td>
</tr>
<tr>
<td>(\text{ctime}(p \ x) + \max{\text{ctime}(x::(\text{filter } p \ xs)), \text{ctime}(\text{filter } p \ xs)})</td>
<td>1</td>
</tr>
<tr>
<td>(\text{ctime}(x::(\text{filter } p \ xs)) = 1 + \text{ctime}(x) + \text{ctime}(\text{filter } p \ xs))</td>
<td>2</td>
</tr>
<tr>
<td>(\text{ctime}(\text{filter } p \ xs) = F(p, xs))</td>
<td>\textit{IH}</td>
</tr>
<tr>
<td>(\Rightarrow \text{ctime}(\text{match } l \text{ with } \ldots) = 1 + \text{ctime}(p \ x) + \max{1 + F(p, xs), F(p, xs)} =)</td>
<td></td>
</tr>
<tr>
<td>(2 + \text{ctime}(p \ x) + F(p, xs))</td>
<td></td>
</tr>
<tr>
<td>(\Rightarrow \text{Recurrence: } F(p,1) = 2 + \text{ctime}(p \ x) + F(p, xs))</td>
<td></td>
</tr>
</tbody>
</table>

As a consequence, \texttt{filter} has a linear run-time complexity.

TransCal system activates several rules to reach a conclusion about the run-time complexity of \texttt{filter}. The first step is to create the \(F\) function as shown as Algorithm 3.1. Then it can activate several rules as shown in Figure 3.2.

From the recursive equation it reached the TransCal system will understand the non-recursive equation. It is done with the list constructor cons \((::)\) and the information that:

\[
l = x::xs
\]

The last equation in Figure 3.2 will be changed to:

\[
F(p,x::xs) = 2 + \text{ctime}(p \ x) + F(p, xs)
\]

And the change parameter which is getting smaller is \(l\) to \(xs\). This \(F\) will be removed and the rest will be multiplied by its \(espace\):

\[
F(p,1) = (2 + \text{ctime}(p \ x)) \cdot espace(1)
\]

This is the body of \texttt{filter}. The full \(\text{ctime}(\text{filter } p \ 1)\) will include also the time
complexity of each argument and will be:

\[
\text{ctime}(\text{filter } p \ 1) = \text{ctime}(p) + \text{ctime}(1) + (2 + \text{ctime}(p \ x)) \cdot \text{cspace}(1) \quad (3.3)
\]

This process can fail if we miss some WCRA information. In that case, we don’t have any fall back and can only wait for future information to be created or added by the user.
Chapter 4

Comparing WCRAs

After the WCRA was successfully traced, the following step is to find, among all the generated programs variations, the implementations with the best performance. This is done by means of comparing the WCRAs that were computed, using the constructed E-PEG. An order is induced over the set of \( ctime \) terms that are available, and the minimal elements are sought. The order is not total; some \( ctime \) pairs may be incomparable, in the sense that our analysis cannot determine that one is strictly smaller. As a consequence, there may be multiple minimal implementations with different complexities.

4.1 Comparison Rules

The WCRA rules generate a family of expressions, that can be described by the following syntactic categories:

\[
E ::= \text{cspace}(\text{term}) \mid \text{constant} \mid E + E \mid E * E \mid \text{max}\{E^+\}
\]

Comparison rules determine the order only within a category. One exception is the constant category, which is always considered to be smaller than non-constant terms. The comparison is performed as follows:

1. Constants — non-negative integers.

2. \( \text{cspace} \) — specify the size of some data structure that is involved in the computation.

3. Composite complexity — expressions that contain other complexity sub-expressions. This allows the construction of arbitrarily large expression that depend on \( \text{cspace} \) variables, as well as \( ctime \) of other functions being called.

The ordering has to handle all the basic values. Because of the different value, the current ordering mostly compare the same types of values or has some basic comparison between different types. Because of it, and sometime the lack of information, the ordering is not full and we might not be able to compare everything.
For example, in Dijkstra’s algorithm for shortest path in a graph with positive weights on the edges, one can use different types of data structures to track “opened” and “closed” nodes. Different data structures will give different time complexity and will perform better in different cases. By using a heap, we get $O(|E| + |V|^2)$, and if the graph is dense, meaning that $O(|E|) = O(|V|^2)$, the result is $O(|V|^2)$. Using of Fibonacci-heap will produce an implementation with time complexity $O(|E| + |V| \cdot \log(|V|))$. This is more suitable for when the graph is sparse, $O(|E|) = O(|V|^2)$, because the algorithm’s complexity is then $O(|V| \cdot \log(|V|))$. 

As shown for $ctime$, we need to the structure of $cspace$:

$$E ::= constant \mid E + E$$

These two categories are sufficient, since all the rules for $cspace$ add up constants based on the number of constructor applications. With the given building blocks we can start to compare between them. Comparing $cspace$ is easy - $A < B$ if $A$ contained in $B$ recursively, $A$ is a constant and $B$ is not or $A$ is a smaller constant then $B$. Otherwise, they are incomparable.

After we defined the compression of $cspace$, we can continue with $ctime$. As mentioned above, we compare only terms of the same syntactic category, but all of the non-constants are larger than any constant.

Comparing the WCRA in each types:

- Constants are compared in their natural order, using the regular $<$ over integers.
- $cspace$ terms are compared as using the substructure criterion explained above. There are cases where it is not possible to compare due to insufficient information; This is one way to create several equivalent cases and make the ordering partial.
- Composite expressions can only be compared if they have the same head symbol (+, *, max). We rely on the composition operators being associative, commutative, and monotone. As a consequence, we can flatten nested applications of + and * and consider the set of operands for each of the expressions being compared.
  If one is a superset of the other, then it is concluded that this term is larger.
  Otherwise, we match operands pairwise; e.g., $t_1 + t_2 < t_3 + t_4$ when $t_1 < t_3$ and $t_2 < t_4$, or $t_1 = t_3$ and $t_2 < t_4$, or vice versa.

Example 4.1.1. Examples of comparing:

- $0 < a - 0$ is the smallest number and is smaller than anything.
- constant $< cspace(l)$ - any constant is smaller than $cspace$.
- $cspace(xs) < cspace(l)$ - can’t be decided. We might know that $l = x::xs$ and that leads to the needed information.
• \(c\text{space}(ys) + 1 < c\text{space}(l)\) - can’t be decided. We might know that \(l = x::y::ys\) and that leads to the needed information.

• \(c\text{space}(l) + 1 < c\text{space}(l) + 2\) - This is a complex type, +. We know that \(c\text{space}(l)\) is the same in both sides and than the constants are compared in constants’ way.

From the above, the conclusion is that the order is partial as sometimes there is not a possibility for choosing which implementation is better and the user will need to check them and choose manually. At least we reduce the number of possible programs and left the minimal number we are able.

4.2 Reconstructing the best program

With the comparison rules we presented, reconstructing all the minimal programs is possible. Extracting program terms from the E-PEG is a recursive process. By starting from the root node, we reconstruct the programs by taking all the hyperedges entering that node, and continuing with the sources of each edge. This is essentially an inverse DFS on the E-PEG during which each node is assigned a set of reconstructed terms. Nullary hyperedges — having no sources — construct atomic, leaf terms (variables and literals).

We take the original reconstruct process and modify it to generate the minimal programs instead of all programs. We build only programs with which our WRCA has associated some complexity. In every expression, we record the rule that generated it, and we do the same with the WCRA rewrite rules. By following the recorded inference steps, we can build a subgraph that led to the conclusion of a particular \(c\text{time}\) expression; doing this recursively reveals all the program terms that were involved in the process. From this subgraph we use the original DFS extraction. After we reconstructed all the implementations of an expression we can compare them to each other and choose the minimal ones. The preliminary subgraph filtering step helps decrease the number of programs drastically, by avoiding expressions that are provably non-minimal and would not be chosen. The comparison is done for each expression and the best optimized implementations of the expression are taken.

\textit{Example 4.2.1.} A simple example will take the program of the expression \(\text{len}(l) * 0\) (visualized at Figure 2.2) after activating the rewrite rule \(a * 0 >> 0\) (visualize at Figure 2.4) to get the result \(\text{len}(l) * 0 = 0\). The E-PEG of those programs is shown at Figure 2.3.
Calculation of multiply \textit{ctime} with inferred rules:

\begin{center}
\begin{tabular}{lcl}
\textbf{inferred} & \textbf{rule} \\
\hline
ctime(\text{len}(l)\times b) = & 1 + ctime(\text{len}(l)) + ctime(0) = 5 \\
\hline
1 + 1 + ctime(1) + ctime(0) = 6 \\
1 + 1 + 0 + 0 = 2 & \textbf{Atom} \\
\end{tabular}
\end{center}

We also find the expression 0 which has \textit{ctime}(0) = 0. Comparing the two, we can see that the expression with the constant is better than the multiply one and would be chosen. This expression can be the final program or to be part of a bigger expression. The building of the bigger expression is done in the same process.
Chapter 5

Empirical Evaluation

In this chapter we will discuss examples that we tested in the Transcal system with the $ctime$ and $cspace$ additions described in Chapter 3 and Chapter 4.

Those examples are small and sufficient in the synthesizer and prove the necessity of our WCRA. We show both linear reduction and big O notation reduction.

5.1 Case Study

Example 5.1.1. First example, and a famous functional example on lists, is filter-map. We execute filter function on a list and then map all the result. The definition is:

\[
\text{filterMap } f \ p \ l = \text{map } f \ (\text{filter } p \ l)
\]

Using the time complexity rules from Chapter 3 we can obtain the following time complexity bound. To do so we will complete the rule needed for both filter as shown in Equation 3.3 and map in Equation 3.2.

\[
\begin{align*}
ctime(\text{filterMap } f \ p \ l) &= ctime(\text{map } f \ (\text{filter } p \ l)) \\
&= ctime(f) + ctime(\text{filter } p \ l) + (2 + ctime(f \ x)) \cdot cspace(\text{filter } p \ l) \\
&= ctime(f) + ctime(p) + ctime(l) + (2 + ctime(px)) \cdot cspace(l) + \\
&(2 + ctime(fx)) \cdot cspace(\text{filter } p \ l) \\
&= ctime(f) + ctime(p) + ctime(l) + (4 + ctime(px) + ctime(fx)) \cdot cspace(l)
\end{align*}
\]

We want to get more efficient version of $\text{filterMap}$. As can be seen from its implementation, in its current form, it creates a temporary list through application of $\text{filter}$, which is then passed as an argument to $\text{map}$, creating a second list. This building time of the intermediate list is wasteful and could be avoided. By avoiding the temporary list we reduce the running time by a constant. We avoid it by “fusing” the function call of filter and map to a single recursive call we reduced. The time complexity
inferred rule

filterMap f p l = Start point
map f (filter p l) = filterMap definition
match (filter p l) with | [] => [] | x :: xs => map definition
  (f x) :: filterMap(f p xs) =
match l with | [] => [] | x :: xs => filter definition
  if(p x) then (f x)::filterMap(f p xs) else filterMap(f p xs) =
⇒ ctime(match l with ...) =

Figure 5.1: Inferring the time complexity of the body of map using WCRA rules

of the list constructor :: is given by the rule:

\[
ctime(x::xs) = 1 + ctime(x) + ctime(xs)
\]

(5.1)

Therefore, by reducing the number of executions of :: we can obtain a faster version.

To enable the analysis one will add an outsider information that:

\[
cspace(filter p l) = cspace(l)
\]

The new definition we find in Figure 5.1 is exactly as expected and was found even before this paper and proven before:

filterMap f p l = match l with
  | [] => []
  | x :: xs => (if(p x) then filterMap f p xs else (f x) :: (filterMap f p xs))

The new WCRA we added (Figure 5.2) is also found and is:

\[
ctime(filterMap f p l) = ctime(f) + ctime(p) + ctime(1) + (1 + ctime(p x) + ctime(f x)) \cdot cspace(l)
\]

As we can see the change is in 1 WCRA as we also expected. The time to generate this information is around 2 minutes and only 1 best program was found. The size of the graph in the end was around 500 nodes and it depends on the order of the rules which were executed (number of rounds until the required information for WCRA was enough).

Example 5.1.2. The second example is no-dup, a function which changes a list to a list with unique elements. Its definition is:

32
Table 5.2: Inferring the time complexity of the body of filter-map using WCRA rules.

<table>
<thead>
<tr>
<th>inferred ctime</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ctime}(\text{match } l \text{ with } \ldots) = \text{ctime}(l) + \max {\text{ctime}([], \text{ctime}(x::xs) + \text{ctime}(\text{if } (p \ x) \text{ then } \ldots)} \text{ \ (1)}$</td>
<td></td>
</tr>
<tr>
<td>$\text{ctime}(l) = 0$</td>
<td>Atom</td>
</tr>
<tr>
<td>$\text{ctime}([], \text{time}) = 0$</td>
<td>Atom</td>
</tr>
<tr>
<td>$\text{ctime}(x::xs) = 1 + \text{ctime}(x) + \text{ctime}(xs)$</td>
<td>Atom</td>
</tr>
<tr>
<td>$\text{ctime}(x) = 0$</td>
<td>Atom</td>
</tr>
<tr>
<td>$\text{ctime}(xs) = 0$</td>
<td>Atom</td>
</tr>
<tr>
<td>$\text{ctime}(\text{if } (p \ x) \text{ then } \ldots) = \text{ctime}(p \ x) + \max {\text{ctime}(\text{filterMap } f \ p \ xs), \text{ctime}(\text{filterMap } f \ p \ xs)}$</td>
<td>IH</td>
</tr>
<tr>
<td>$\text{ctime}(\text{filterMap } f \ p \ xs) = F(f, p, xs)$</td>
<td>IH</td>
</tr>
<tr>
<td>$\text{ctime}(f \ x) :: (\text{filterMap } f \ p \ xs)$</td>
<td>$1 + \text{ctime}\left(f \ x + \text{filterMap } f \ p \ xs\right)$</td>
</tr>
<tr>
<td>$\Rightarrow \text{ctime}(\text{match } l \text{ with } \ldots) = \text{ctime}(p \ x) + \max {F(f, p, xs), 1 + \text{ctime}(f \ x) + F(f, p, xs)} = 1 + \text{ctime}(p \ x) + \text{ctime}(f \ x) + F(f, p, xs)$</td>
<td>$\Rightarrow \text{Recurrence}: F(f, p, l) = 1 + \text{ctime}(p \ x) + \text{ctime}(f \ x) + F(f, p, xs)$</td>
</tr>
</tbody>
</table>
nodup l = match l with
| [] => [] /
| x :: xs => (if (contains xs x) then nodup xs else x :: (nodup xs))

And this naive implementation’s WCRA:

\[(2 + cspace(1)) * cspace(1)\]

The naive implementation is comparing each element to the previous elements and add it only if it was not found. We don’t have a request for ordering in the result. We expect this implementation to be better by using an accumulator of set – another data structure. Searching in a set is much faster and we know we will see an improvement. The full work to reach the optimized version is shown at Example 2.4.1. The new nodup implementation is:

\[
\text{nodup } l = \text{nodup'} \ l \ \{} \ \{}
\]

\[
\text{nodup'} \ l \ \text{acc} = l \ \text{match} \ [\] \Rightarrow \text{acc} /
\]

\[
x :: xs \Rightarrow (\text{if (elems acc x) then nodup'} \ xs \ \text{acc}
\]

\[
\text{else nodup'} \ xs \ \{\text{acc} \ \text{\setminus} \ \{x\}\})
\]

A new auxiliary function, nodup’, was created, which uses an accumulator as can be expected. The new WCRA is also better (full way is shown at Figure 5.3):

\[3 * cspace(1)\]

We can see that the new program found, which was also chosen by the search, is better in terms of big-O notation. The time required to find this program and analyze it is 1 minute and the process produces 2 candidate programs, both with the same time complexity. (They differ only in the order of the operands to set-union, \(\cup\).) The size of the constructed E-PEG was 300 vertices. The exact size slightly depends on the order of execution of the rewrite rules, since TransCal terminates as soon as sufficient WCRA information is produced.
inferred $ctime$

$ctime(\text{nodup }1) = ctime(\text{nodup }'1')$

$ctime(\text{match }1 \text{ with } \ldots) = \max \{ ctime(\text{[]}), ctime(\text{[]}) \}$

$ctime(x::xs) + ctime(\text{if (elems acc x) then\ldots})$

$ctime(\text{[]} ) = 0$

$ctime(x::xs) = 1 + ctime(x) + ctime(xs)$

$ctime(x) = 0$

$ctime(xs) = 0$

$ctime(\text{if (elems acc x) then\ldots}) = ctime(\text{elems acc x}) +$

$\max \{ ctime(\text{nodup}' xs \text{ acc}) + ctime(\text{nodup}' xs \text{ acc}\cup\{x\}) \}$

$ctime(\text{elems acc x}) = 1 + ctime(\text{acc}) + ctime(x)$

$ctime(\text{acc}) = 0$

$ctime(\text{nodup}' xs \text{ acc}) = F(xs, \text{acc})$

$ctime(\text{nodup}' xs \text{ acc}) = 1 + F(xs, \text{acc}\cup\{x\})$

$\Rightarrow ctime(\text{match }1 \text{ with } \ldots) = 2 + \max \{ F(xs, \text{acc}), 1 + F(xs, \text{acc}\cup\{x\}) \}$

$\Rightarrow \text{Recurrence: } F(1) = 3 + F(xs, \text{acc}\cup\{x\})$

Figure 5.3: Inferring the time complexity of the body of $\text{nodup}$ using WCRA rules
Chapter 6

Conclusion

This thesis presented a structured approach to handle effectively the instruction counter for constants and lists using the TransCal system. The first benefit of our approach is that it removes the need to think about optimization ordering. When applying optimizations sequentially, ordering is a problem because one optimization, say A, may perform some transformation that will irrevocably prevent another optimization, say B, from triggering, when in fact running B first would have produced the better outcome. This so-called phase ordering problem is ubiquitous in compiler design. In our approach, however, the compiler writer does not need to worry about ordering, because optimizations do not destructively update the program – they simply add equality information. Therefore, after an optimization A is applied, the original program is still represented (along with the transformed program), and so any optimization B that could have been applied before A is still applicable after A. Thus, there is no way that applying an optimization A can irrevocably prevent another optimization B from applying, and so there is no way that applying optimizations will lead the search astray.

As a result, compiler writers who use our approach do not need to worry about the order in which optimizations run. Better yet, because optimizations are allowed to freely interact during equality saturation, without any consideration for ordering, our approach can discover intricate optimization opportunities that compiler writers may not have anticipated, and hence would not have implemented in a general-purpose compiler.

The second benefit of our approach is that it enables global profitability heuristics. Even if there existed a perfect order to run optimizations in, compiler writers would still have to design profitability heuristics for determining whether or not to perform certain optimizations such as in-lining. Unfortunately, in a traditional compilation system where optimizations are applied sequentially, each heuristic decides in isolation whether or not to apply an optimization at a particular point in the compilation process. The local nature of these heuristics makes it difficult to consider the effect of future optimizations. Our approach, on the other hand, allows the compiler writer to design profitability heuristics that are global in nature. In particular, rather than choosing whether or
not to apply an optimization locally, these heuristics choose between fully optimized versions of the input program. Our approach makes this possible by separating the decision of whether or not a transformation is applicable from the decision of whether or not it is profitable.

Future research and analysis are required on more types of data structures which allows for automatic handling. For some of the more advance data structures, i.e. trees, there is a need for extending the methods that Mathematica can handle, such as divide for trees recursion.

Furthermore, this thesis has presented how a regular, simple recursion is solved. In many cases a recursion is not in a single function, instead it passes several functions. These complex cases are beyond the scope of this thesis, and may require a more elaborate system to be solved.

The current machine to user communication is done through a text base interface. As the user has a crucial involvement in the programs optimizations it is highly recommended that a modern IDE is used. This IDE allows the user to concentrate on the content understanding instead of concentrating on the process by providing better experience and clear understanding.

There is vast research done on this field, different approaches have been taken but there is a lack of a generalization theory that allows for a unified approach toward solving the problem. It seems that the solution chosen depends on the constrain set forth to solve, in this thesis the constrain is worst-case scenario, but when a different constrain is chosen, i.e. statistically average, a different solution is taken which will have an impact on the CPU and complexity. It is also important for the solution selection what is the WCRA that is chosen, for example different debuggers and run-time tools may use different ones.

Yet another point for taught is the information gathered in the use of the WCRA in automatic search. This information can be put to use for additional cases, for example use the information for rules ordering by their counter improvement thus activate them in an order from the best impact to the worst impact.
Appendix A

Rewrite rules used in this paper

A.1 Program Expression Rewrite rules

1. \(\neg\neg x = x\)
2. \(\neg false = true\)
3. \(x == y = y == x\)
4. \(x' \in \{x\} = (x == x')\)
5. \(elem x (x' :: xs') = (x == x') \land elem x xs'\)
6. \(\neg(x == y) = x \neq y\)
7. \(\neg(x \in s) = x \notin s\)
8. \(\{x\} || xs = x \notin xs\)
9. \(\neg(x \lor y) = (\neg x \land \neg y)\)
10. \(\neg(x \land y) = (\neg x \lor \neg y)\)
11. \(\{x\} \cup \{x'\} || xs = (\{x\} || xs) \land (\{x'\} || xs)\)
12. \(elem x xs = x \in elems xs\)
13. \(elems (x :: xs) = \{x\} \cup elems xs\)
14. \(\{x\} || elems xs \land \{x'\} || elems xs = (\{x\} \cup \{x'\}) || elems xs\)
15. \(x + y = y + x\)
16. \(y:*x = y++(x::[])\)
17. \((x::xs)++ys = x:(xs++ys)\)
18. \(take(xs, 0) >> []\)
19. \texttt{take(xs,len(xs)) >> xs}

20. \texttt{take(xs++ys, x) >> take(xs, min(len(xs), x)) ++ take(ys, bounded\_minus(x, len(xs)))}

21. \texttt{range\_exclude(x, y) ++ range\_exclude(y, z) >> range\_exclude(x, z)}

22. \texttt{range\_exclude(x, y + 1) = range\_include(x, y)}

23. \texttt{range\_include(x, x) = x::[]} 

\textbf{A.2 Space complexity rewrite rules}

1. \texttt{cspace(false) = 0}
2. \texttt{cspace(true) = 0}
3. \texttt{cspace(\[]) = 0}
4. \texttt{cspace(x::xs) = 1 + cspace(xs) + cspace(xs)}

\textbf{A.3 Time complexity rewrite rules}

1. \texttt{ctime(match arg with pat\_1 \Rightarrow body\_1 | ...) =}
   \texttt{ctime(arg) + max\{ctime(pat\_1) + ctime(body\_1), \ldots\}}
2. \texttt{ctime(x::xs) = 1 + ctime(x) + ctime(xs)}
3. \texttt{ctime(\[]) = 0}
4. \texttt{ctime(elems(s, x)) = 1 + ctime(s) + ctime(x)}
5. \texttt{ctime(a\*b) = 1 + ctime(a) + ctime(b)}
6. \texttt{ctime(len(l)) = 1 + ctime(l)}
Bibliography


המימושים

הтяרה העקרית היא שבכל שיווק נוסח יオリジנלי בפורמט יעיל את מספר המימושים. המitorio, בכדי להימנע את מספר האפקטיביות שלו, מחלה הרחבת מספר הכותבים בעשרות פעמים. 

cדי להדריך את האטגר, זומタイ ה PEG-א, שהחשוב בשיבוץ פיתוח. הזדמנות להRgb רכיב שמיתושım. שיפור בمهرת נטורל הוא מאפי

ולו לייצר את זה המימושים, שמספרים לאפקטיביות, ברורה עלייה.

מעכרת שכנותינו היא מערכת על מאגר חיקויים שלDER להכין את הטור. הזדמן מהמודלים המполнен. באת כלOPSIS, ההכנה של הסטים של המימושים, וברצון של חיקויים שלDER לאקספונציאלי,الف

המודלים והם יצר מושך חיות.

לערב שטראטיגיה אמאכרות והורכות הקיימות של כל לדנה, ואת היסודות ליצ分かる. מחוז

דוחות. כפי שטרטוגם בבהלה, מנסים חיקוי שכנות חיתוש בניוורי ה-PEG מתכונת העתיד, במיתשלים כדי

השכנות כתיבים ניתן מ打交 לברך של כל היקומים ישות להmethodName גם כניצנים חיתוש וה المنتدى. זכויות חיסון השכנות החיתוש אינני מתכונת לא להתוכן של תחומי היקומים, חיץ ב שינויים.

לבבדת איבר בישמה היא:  

c\text{time}(ELEM \times L) = \text{cspace}(L) + c\text{time}(x) + c\text{time}(l)

חק זהبريעם, את גולד הרשימה, ולהח חיתוך השכבות לא טור סביר ונחק

שכפות למוקם.

כל תיפוס תונים יש רישום של בנאיים, ערכ נבנה בברחת. להח חיתוך תק подпис

באמום קורסיבים לכל פרוטו לכל בנה.

בחנה האג-social ומייפלין רכ ברישום ובפוןיעים, אך נית המתיח ביעד מפרסם

אלגברי שעמד מש上がり בוץ.

חק השכנות החיתוש הוא ב시스ים ומודדים סבכים חיתוש, אבל על חיקוי שכתוב

קימי. כנואר מכסיים פקיצת חיתוש, ואחח חיתוך חיתוש, ערכי ולנה עזור את

הטיסות. גזירתה החלינו החיתוש שילש שבעת שבשופט לאتنوع נמצאות

טיסות מחוסר מדינ.

ארחיה ישמטה את התוכנית בברחות,ナンיהすぎים לברזור את המימוש החי טוב. כי

לבחór את המימושים הכי טוב לפנייה דרכ להלוויית ביין. העבורה מתמזגת השואות גוס

עבור מחומק וצמיד עד. 

לאחרי מיתוסים בכל הרשות_notes של דכת בלֶנת את המימושים הכי יעיל. הביניות

רכוסיביות, כלכל שလוריקה מימושים פוחת טובים, ובcorp את מNos ועגילים לכל המנהלים

העובשים שזכרים שילופים.
תקציר

מידי ישנם מיליונים, ואולי יותר, שורות קוד מתכות בצבעים. אחת מהיות שיטה זו,
לאמחתי שלה י’: כל תוכנה, בכל תקינה התרחע או אךגר בלעדים בתכונה ותוחפוסית ב يول.
בייחוד שמשתמשי אדם כריעה.

ניתן ليירכ חכם תוחם איסוף קרוב ליעל, שלבל התחום ואו לירצ מזרד שיכל
ליצר מימוות שקולים לתוךית תוחנה. הלבל בעינו ואל חתונות ביסיקתת ניורה.

ByText על מכת עליזותר את דרשות התוקבוקים. על מכת שחברת ״ hiệp איסוף חכמה
והזרת דא ליבות הקבוקים התושבים. התוצרו, ואלו מתוחפשו ברעיה של ספירת
הפולח המבוצעת על ידי התוכנית. התוכן ברית הת(SQL מפותר המדה, היא ליצך
בוחר Mahmدار את כל המימוות של התוכנית ולהבדל כל את החת מככל שיטה סטסי
כדי לזרו את הטוית בוחר.

עבודת גמר או סוקטת בבעחה לדרד מישון קטבל המימוות היעיל בוחר בקם בצבע
תוכנית תחנה. השיטה המוזגת מבוצעת על מקרר לכלל ספקית את המידה והדרשו.
בעבודת, או מדיניות מערער אוומיאנותי להמשלת המבוסס על שחפת מ COURT תוחנה
накראות, או מתוכי את המערער או מחנה התוכנית מבסיים ומ konuştu
אחת, וחכמים מיכרו כדי פעולות שהושMahon בביצוע. מבנה התוכני
הניקוי ש𝑹ームרש או המערער או הפורפר אמדיך
הניקוי של השמעת או המערער או הפורפר אמדיך
ו䰲��וט הידיעות לשני המצע עלייה הפרדב בין ממידי שגויים ק móc הביטויים.

הטיפוסים שלחט והמכים תמוהת ייעודיים: hüppside — hüppside
המכים טלייה להתרח פורקטי ל驽רט אחרובית אלocard
הית置换 הלוחות פורקטי ל驽רט השטיפות. פורקטי השטיפות למחלקה למחלקה בראינאיסיאליות התוכנית
. hüppside

 Pvase

המעורכים מתתאת בתוניס לייעל ל capitalists האירים המימוות Owned. מבנה
הנווטים הנבכר ליגון כל המימוות שapyז במלוך החיפשות הוא
רגשה E-PEG הנווטים במענית הנווטים, PEG, גון במענית קדימת. בגנף זה כהッシ מורכז
בסטיסי של המנה הנווטים, PEG, מבנה הנווטים, PEG, גון במענית קדימת. בגנף זה כהشي מורכז
מבח הפריטים, כלומר איסוף רזה לייעל ביבי במענית התוכנית קרוי כלות את אות ברה
קרור힉. פורקטי התוכנית והותא פורקטי לייעל גלובליים פורקטיים לסטת יותר
בוח מיניים. היציג הים שלגר ואמר טגייג טגיפס כעד גורף את הלואיר

i
המחקר בוצע בהנחייתו של פרופסור שחר יצחק, בפקולטה למדעי המחשב.

תודה

אני רצה להודות למנהלת השപלה, למשנהי וולברן.

הכרת חזרה מועילה לעצמי ועל מيمن מחקר זה.
ניתוח משאבים
בṢינתזה של תוכנה

חיבר על מחקר
לשםMISSION לתוכנית מחקר
 groupName מחשב

הוגש על ידי

חתימת

הוסף למסגרת
מכון טכנולוגי בישראל
 Technion - Computer Science Department - M.Sc. Thesis MSC-2020-23 - 2020
ניתוח משאבי
בסינתזה של תוכנה

תומר.CLבר