Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.

http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.

http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Resource Bound-Analyse von Lisp-Programmen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Clemens Danninger
Matrikelnummer 1127840

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass.Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Mitwirkung: Dr.techn. Moritz Sinn, MSc

Wien, 27. Februar 2016

Clemens Danninger   Florian Zuleger

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Resource Bound Analysis of Lisp Programs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Clemens Danninger
Registration Number 1127840

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Assistance: Dr.techn. Moritz Sinn, MSc

Vienna, 27th February, 2016

Clemens Danninger   Florian Zuleger

Erklärung zur Verfassung der Arbeit

Clemens Danninger
Karl-Schwed-Gasse 98/2, 1230 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. Februar 2016

Clemens Danninger

Acknowledgements

This work would not have been possible without the collaboration and help of several people.

Particularly, I would like to thank Antonio Flores Montoya, whom I collaborated with on this project, for his work on CoFloCo and on the Lisp frontend, and for enlightening discussions on both concrete implementation issues and more general aspects of resource bound analysis; Moritz Sinn for guidance, discussions, and for his feedback on several drafts of this thesis; Florian Zuleger for his advice, as well as for the opportunity to work on this topic; Ivan Radiček for help with formalizing semantics and constructing proofs; and Tomer Kotek for informative discussions.

Kurzfassung

Resource Bound Analysis ist ein Bereich der Programmanalyse, der sich mit der Bestimmung von Bounds (Grenzen) für die Menge an bestimmten Ressourcen, die beim Durchlauf eines Programms verbraucht werden, beschäftigt. Bounds sind Ausdrücke über die Eingabeparameter eines Programms, die für beliebige Kombinationen von Parametern korrekt sein sollten, was durch einfaches Testen nicht erreichbar ist. Die Analysetypen, die hier behandelt werden, sind statisch (d.h. es ist nicht notwendig, das Programm auszuführen, um einen Bound zu bestimmen) und vollständig automatisiert. Bounds für viele verschiedene Arten von Ressourcen sind von Interesse, darunter Laufzeit, Stack und Heap-Ausnutzung oder wie oft eine bestimmte Funktion aufgerufen wird. Resource Bound Analysis hat zahlreiche Anwendungsmöglichkeiten in der Entwicklung von Embedded- und Echtzeitsystemen, Komplexitätsanalyse von Algorithmus-Implementierungen, Security und weiteren, die wir detaillierter beschreiben.

Viele neuere Ansätze und Implementierungen zielen auf imperative Programme ab. Im Gegensatz dazu konzentriert sich diese Arbeit auf funktionale Programme, wo Wiederholung (der wichtigste Aspekt in der Resource Bound Analysis) als rekursive Funktionsaufrufe statt als Schleifen implementiert ist. Da Ressourcenanalyse ein umfangreiches und komplexes Thema ist, beschränken wir unsere Analyse auf die Berechnung von Bounds für die Anzahl an Funktionsaufrufen und damit, wie oft jeder Teil des Programms ausgeführt wird. Wir legen allerdings dar, dass damit das Kernproblem gelöst ist und eine Erweiterung auf andere Ressourcentypen auf dieser Grundlage relativ einfach wäre.

Wir präsentieren eine Boundanalyse für eine seiteneffektfreie Untermenge von Common Lisp mit Funktionen erster Ordnung. Die Analyse baut auf CoFloCo auf, einem existierenden Boundanalysesystem, das Bounds aus einem System von Cost Relations berechnet. Wir beschreiben eine Übersetzung von Lisp in Cost Relations, die die Berechnung korrekter Bounds für Lisp-Programme mit polynomieller Komplexität ermöglicht.

Wir berichten über Experimente, die mit einem umfangreichen Benchmark durchgeführt wurden, das aus zusammen mit dem Theorembeweiser ACL2 verteilten Lisp-Programmen besteht. Insgesamt analysierten wir 19,491 Funktionen in 1,934 Dateien. Das Benchmark enthält Code für Modelle, die für Theorembeweise in industriellen Anwendungen wie etwa Hardware- und Mikrocodeverifikation verwendet wurden. Dasselbe Benchmark wurde außerdem bereits für die Evaluation des Terminationsbeweisers CCG verwendet, der in

ACL2 enthalten ist. Wir untersuchen die erhaltenen Bounds und asymptotischen Komplexitätsergebnisse sowie die Laufzeit der Analyse als wichtigen Aspekt für praktische Anwendungen von Programmanalyse, um die Eignung unseres Ansatzes zu beurteilen und verbleibende (technische wie fundamentale) Probleme zu identifizieren.

Zuletzt leiten wir aus diesen Experimenten sowie aus Beobachtungen während der Implementierung aussichtsreiche Ansätze für weitere Untersuchungen ab.

Abstract

Resource bound analysis is a kind of program analysis concerned with determining bounds on the amount of a certain resource consumed when running a program. Bounds are expressions over the input arguments of a program, and should be sound for any combination of parameters, which cannot be achieved through simple testing. The kinds of analyses considered here are static (meaning that it is not necessary to run the program in order to determine a bound) and fully automated. Bounds for many different types of resources are of interest, such as running time, stack and heap usage of the program, or the number of times a specific function is called. Resource bound analysis has numerous applications in embedded and real-time systems development, complexity analysis of algorithm implementations, security etc., which we will describe in more detail.

Many recent approaches and implementations are aimed at imperative programs. In contrast, this work focuses on functional programs, where repetition (the most critical aspect for resource bound analysis) is implemented as recursive function calls rather than as loops.

Since sound resource analysis is generally a wide and challenging topic, the scope of this work was limited to computing bounds on the number of function applications only, i.e., how often any given part of the program is executed. However, we posit that this solves the core problem, and that other measures could be based on this foundation with relative ease.

We present a bound analysis for a first-order side-effect-free subset of Common Lisp. The analysis is built on CoFloCo, an existing bound analysis system with support for recursive programs, which derives bounds from a system of cost relations. We describe a translation from Lisp to cost relations, which enables calculating sound bounds for Lisp programs with polynomial complexity.

We report on experiments performed on a large benchmark consisting of the Lisp programs provided together with the ACL2 theorem prover. Overall, we analyzed 19,491 functions in 1,934 files. The benchmark includes code for models which have been used for theorem proving in industrial applications such as hardware and microcode verification. The same benchmark has also previously been used to evaluate the CCG termination prover built into ACL2. We investigate the reported bounds and asymptotic complexity results, as well as the running time of the analysis, as an important aspect

of practical program analysis, in order to assess the suitability of our approach, and to discover remaining (technical or fundamental) issues.

Finally, promising avenues for further investigation are derived from these experiments and from observations made during the implementation work.

Contents

CHAPTER 1

Introduction

Automatic resource (or cost) analysis is a type of automated (mechanical) program analysis which determines the amount of a certain resource consumed when running a program (referred to as the *cost*). The cost is defined in terms of a metric, such as the number of execution steps required to evaluate the function, or the consumption of some resource during execution (memory, stack space, etc.). The cost of specific operations within the program, such as function calls, is described by a *cost model*. For example, the cost model used in this work assigns cost 1 to function calls and conditional branches.

The analysis presented in this work, as well as the related works which are discussed, all work statically, meaning that it is not necessary to actually execute the program in order to obtain the desired results. This means that the result will be valid for any concrete function invocation with any combination of argument values, i.e., not dependent on a (necessarily limited) selection of test parameters. It is therefore possible to assert that a result will be correct in all cases, which is not possible with testing alone.

The resource consumption of a program is often hard for programmers to understand intuitively, especially for long sections of code with complex interactions. Manual analysis quickly becomes highly laborious, and errors which might significantly affect the result are likely to occur. Therefore, a completely automated and provably correct resource analysis is highly desirable.

Resource analysis has various applications in embedded and real-time systems (where provably sufficient resources and time limits are critical for safety), scheduling [SC95], complexity analysis of algorithm implementations, optimization, static analysis of energy consumption [Gre+15], more generally in proving safety- and security-related properties of programs [HDW17], and many others.

To demonstrate what our analysis is aiming at, consider a concrete example as shown in Listing 1. In this case, it is quite easy to see that one recursive call to `listlen` will be made for each element of the list – that is, the number of recursive calls, and therefore the complexity, is linear in terms of the length of the list. However, it is easy to imagine a scenario where this would not be so simple. For example, the length of `x` may depend on the result of some other, much more complex calculation.

```lisp
(defun listlen (x)
  (if (consp x)
    (+ 1 (listlen (cdr x)))
    0))
```

Listing 1: A function which determines the length of the list `x` by recursively calling itself once for each list element.

A resource or cost bound is an expression over the (input) arguments of a function, containing only basic arithmetic operations (most importantly not containing the functions being analyzed), and evaluating to a (usually non-negative) value which is an upper bound on the cost of the function. In the example, a valid upper bound for the execution step metric would be the length of the argument `x` times the cost of each recursive call (which can be easily determined as the sum of the costs for a constant set of function calls), plus constant costs for the initial call and the end of the list.

In addition to such upper bounds, it may also be desirable to determine lower bounds, which are relevant for different applications. However, this work focuses on upper bounds.

Obtaining upper bounds on the consumption of resources such as execution time is an undecidable problem [Weg75; MML97]: when attempting to calculate a bound, it must implicitly be decided whether the program terminates (i.e., whether a bound actually exists). However, this is impossible in the general case, as the halting problem [Tur36] proves. Consequently, any sound bound analysis is necessarily incomplete. In our work, we aim at a sound but incomplete analysis, meaning that there may be functions for which the analysis fails to obtain a bound. However, it turns out that many programs are reasonably simple, and it is indeed possible to calculate bounds for them, as evidenced by the experimental results presented in Section 5.2.

The degree of accuracy of a bound, i.e., how small the difference between the bound and the actual cost is, is referred to as *precision*. Bounds need not necessarily be precise. While high precision is of course desirable, there is often a trade-off between precision and running time of the analysis. Also, achieving maximum precision in corner cases may require a disproportionate amount of complexity in the analysis. Furthermore, depending on its intended use, a bound expression which is shorter and easier to understand might actually be preferable to a more precise, but much more complicated one.

It is also important to note that in our analysis, the bound is calculated for an idealized model of execution corresponding directly to the source code being executed on an abstract machine, and does not factor in the underlying machine model. The exact cost in terms of running time, memory etc. depends on a variety of factors, which would be complicated to analyze and cannot always be determined from the source code alone. For example, a compiler may or may not make specific optimizations, which might significantly change the cost (and even the asymptotic complexity) of a function, or cause it to depend on unpredictable outside factors.

1.1 Motivation

Originally, the field of automated resource analysis began with analyzing functional programs (see, e.g., [Weg75]). On the other hand, recent work has mainly been focused on imperative programs (see Chapter 6 for a discussion of some recent approaches), and much progress has been made in that area. Imperative programming is currently more widespread in real-world applications, especially safety-critical ones such as real-time systems.

However, imperative languages such as C have many properties and constructs which make resource analysis particularly challenging – for one, side effects are a core operating principle and may occur anywhere in the program. Reference constructs such as pointers introduce complications which usually necessitate making certain “sanity assumptions”. For example, a “reasonable” composition and use of the functions being analyzed is assumed, e.g., no writes outside array bounds, pointer arguments to functions being distinct, etc. (see, e.g., Sinn et al. [SZV14; SZV15]). These sanity assumptions further complicate the comparison between analysis implementations. Since different systems make different assumptions, the results are not directly comparable.

For programmers, functional programming has various advantages compared to imperative methods. Most importantly, the elimination of side effects has the potential of reducing security and concurrency issues, while also greatly simplifying static program analysis. Many functional languages also have features such as strong static typing with automatic type inference (which is used as a basis for many approaches to resource analysis), and higher-order functions, which aid modularization [Hug89]. Functional programming paradigms are also gaining relevance as traditionally imperative languages, such as Java [UFM14], are adding features which enable functional-style programming. As a consequence, functional languages are also an interesting target for resource analysis.

1.1.1 Advantages of Lisp for resource analysis

In this work we therefore analyze functional programs, but use an approach and implementation which was previously used for analyzing imperative programs, such as Java and C. This enables us to leverage recent achievements in the imperative domain for analyzing functional programs. Our analysis processes a simplified first-order subset of Lisp [McC65], a dynamically typed functional programming language, one of the first of its kind and still in use today. Lisp has a variety of properties which make it an interesting and particularly suitable target for resource analysis. Most importantly, resource analysis is much simpler than for imperative programs, as the issues described above (side effects, pointers etc.) are largely avoided. In the targeted Lisp subset, as in functional programming in general, repetition (the most critical aspect for resource analysis) is implemented through recursive function calls rather than loops, leading to a more regular program structure. Side effects occur only optionally rather than as a core operation, or not at all (in the subset of Lisp we analyze), and there are no pointers. Therefore, no sanity assumptions are required, meaning that an analysis can be sound without having to handle unexpected interactions within the program. Due to this simplicity, it is much easier to develop a completely sound resource analysis, which in turn will be comparable to other implementations on an equal basis.

Lisp also has advantages over more advanced functional languages, where program analysis is concerned. Unlike in Haskell, for example, function arguments are not lazily evaluated – with the exception of conditional branches, all program code is executed eagerly. This evaluation strategy is much easier to analyze, as it can be assumed that any code not explicitly marked as depending on a decision (i.e., contained in a conditional branch) will be executed in full (see also Section 6.5). Furthermore, Lisp code is very easy to parse, since it consists only of function definitions, function applications, and constant literals as basic elements.

An alternative approach to focusing on the high-level language is to analyze the underlying model of execution, which is still an imperative one in the vast majority of cases (in the form of machine code being executed on a general-purpose processor, or bytecode on virtual machines). Indeed, this approach has been followed in COSTA (bytecode analysis [Alb+07]), Loopus (analysis on LLVM intermediate code [SZV15]), and other systems. However, since this machine code is itself an imperative program with few restrictions, it is subject to the same issues described above – the advantages of the functional source program, such as its regular structure and exclusion of side effects, are lost. In contrast, our approach uses a clean, declarative abstraction which largely preserves the properties of the original program.

Furthermore, like many other, more recent languages such as Python and JavaScript, Lisp is not statically typed. While this is a disadvantage in some respects, as static type information is helpful for certain types of analyses, the corresponding advantage is that our analysis would be readily adaptable to other dynamic languages. While not dependent on types, the analysis could optionally make use of explicit type information in order to analyze languages where such information is available more effectively. In fact, our analysis implicitly uses type information for analyzing Lisp. Semantic type checks such as `integerp` constrain the range of possible values of a variable (in this case, enforcing that it is an integer and not, e.g., a list), and this information is used by our analysis in the same way as any other constraint.

Finally, an important goal for this work was to identify challenges and obstacles in implementing such an analysis, and to propose opportunities for future work.

1.2 Challenges

From the description above, as well as from analysis of related work, two main challenges can be identified, which we deal with in this work:

- Performing sound resource bound analysis of programs written in Lisp, a dynamically typed and relatively unstructured language, where deterministic inference of a single type for each expression is not possible.
- Evaluating a bound analysis system on a large, realistic benchmark (as opposed to individual academic examples, or small sets of test code written solely for evaluating resource analysis). There is a distinctive lack of such evaluations in the literature.

1.3 Overview

By analyzing a limited subset of Lisp, we focus on a language which is relatively simple, free from complications such as side effects, and admits sound resource bound analysis without requiring unsound assumptions.

We base our analysis an existing approach and corresponding tool (CoFloCo [FH14; Flo16], described in detail in Chapter 4). CoFloCo has previously been used for analyzing imperative programs with good results, whereas we now apply it to programs written in a functional language (Lisp). CoFloCo calculates bounds for so-called cost relations, which are a declarative abstraction of programs. Furthermore, CoFloCo can directly analyze recursive programs, and performs amortized analysis (see Section 4.5).

In order to analyze Lisp programs, we develop a translation of Lisp code to abstract cost relations, for which CoFloCo then calculates bounds. This involves the abstraction of important properties of structural values to integers, using so-called measures.

We then perform a large-scale evaluation of a bound analysis system for a functional language. There is a distinctive lack of such evaluations, even though they are needed as a basis for comparisons between approaches. The evaluation uses a large set of Lisp programs distributed with the ACL2 automatic proving system [KM97; KMM00; KM16]. The benchmark consists of a diverse suite of purely functional and, most critically given the limitations of our approach, side-effect-free and first-order code. The latter is especially important, as our approach cannot handle higher-order functions, but they are normally quite prevalent in regular Lisp application code, or indeed any other functional language. Overall, this benchmark consists of 5,615 Lisp files containing 2,146,352 SLOC[1]. Most importantly, this code does not consist of artificially constructed or academic examples: the benchmark includes code for models which have been used for theorem proving in industrial applications such as hardware [Hun+10] and microcode [Rus+05] verification.

[1]determined using David A. Wheeler's 'SLOCCount'

In conclusion, Lisp is a good demonstrator platform for the purposes of program analysis, despite its lack of presence in modern software engineering, which is still dominated by imperative languages.

1.4 Contributions

The contributions made in this work fall into the following main categories:

- A large-scale evaluation of a bound analysis system for a first-order subset of the functional language Lisp. The benchmark code on which the evaluation is performed was written for unrelated purposes, giving more weight to the results.
- An application of an existing approach for bound analysis, implemented in the tool CoFloCo [FH14; Flo16], to dynamically typed functional programs. Based on test results, we also identify issues and opportunities for improvements in CoFloCo itself.
- An resource bound analysis which, due to the simplicity of the target language (such as freedom from side effects), is sound (with a single well-defined exception, which could be easily remedied using a method we propose separately). This, in conjunction with the large-scale evaluation described above, could form a basis for comparisons, since other bound analysis systems for functional languages could be adapted to Lisp without requiring them to agree on a specific set of (unsound) assumptions.
- An investigation of cases where our approach is sufficient, and proposals for extensions which would result in bounds for more functions and/or higher-precision bounds.

1.5 Approach & Methodology

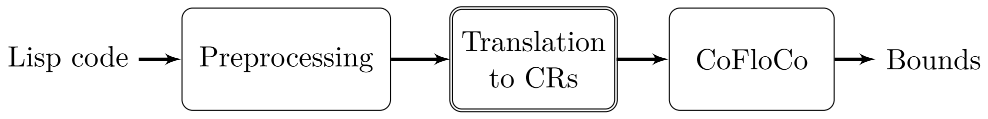

Figure 1.1: The high-level structure of the analysis process. The translation to cost relations (CRs) constitutes the main part of this work.

In this work, we analyze Lisp programs in order to determine upper bounds on the total number of execution steps performed. We implement and describe a Lisp frontend for CoFloCo [Flo16], an existing resource analysis system with support for recursive programs, which has so far mostly been used for resource analysis of imperative programs.

Figure 1.1 shows the three main steps of the analysis. First, a preprocessing step simplifies and normalizes the original Lisp code. A Lisp-specific frontend then translates the result to so-called cost relations, abstract definitions describing program behavior and cost semantics.

Generating cost relations which accurately represent the real program behavior, which is necessary for obtaining precise bounds, is an almost arbitrarily complex task. Therefore, any such translation constitutes a compromise between precision and complexity.

The fundamental implementation of a translation of the Lisp program structure was developed in collaboration with Antonio Flores Montoya. The author then extended the translation with measures, in order to make it suitable for non-integer programs. Furthermore, the author developed an extension to generate constraints for multiply recursive functions, according to automatically verified properties. The author also developed a formalization of the translation together with proofs for its correctness, as shown in Section 2.5.

From the cost relations, the bound analysis tool CoFloCo determines the upper bounds on the total number of execution steps. CoFloCo supports recursive function calls directly, which is a considerable advantage for analyzing functional languages. Our analysis is capable of analyzing a subset of Lisp which is largely identical to that processed by ACL2 – most importantly, both our theory and implementation support only side effect-free first order functions.

We then applied this analysis to the large-scale benchmark suite from the ACL2 distribution. The author performed evaluations and analyzed the results. Running tests allowed us to iteratively improve the translation to cost relations, as well as the analysis itself. Due to its size, the benchmark exhibits a far-ranging coverage of corner cases, which were identified and resolved incrementally. Based on the results and on lessons learned during the implementation, the author then drew conclusions concerning the performance and limitations of the analysis. As the results show, our implementation performs well, despite following a relatively simple approach. Finally, the author identified possible directions for further work, which are described in Chapter 7.

1.6 Structure of this work

The remainder of this work is structured as follows: Chapter 2 gives a specification of Lisp and cost relations, describes how the former is translated to the latter, and provides a correctness proof for this translation. Specific aspects of the implementation are described in Chapter 3. An overview of CoFloCo is given in Chapter 4. Chapter 5 describes the experiments which were performed, and presents results along with conclusions which can be drawn from them. Chapter 6 discusses related work in more detail, and describes specific commonalities and differences. In Chapter 7, remaining issues and opportunities for future work are discussed. Finally, Chapter 8 sums up the results of this work with a brief conclusion.

CHAPTER 2

From Lisp to cost relations

As explained in Section 1.5, Lisp programs must be translated to cost relations for analysis. In order to obtain meaningful results, this step should retain the resource properties of the original program – specifically, the resulting bounds should be sound with respect to the original program's cost, and as precise as possible.

2.1 Brief overview of Lisp

Lisp is a functional programming language developed from 1958 on by McCarthy et al. [McC79; McC65]. Lisp code is composed of S-expressions, i.e., nested expressions enclosed in brackets, which can be interpreted as either lists or function applications consisting of a function name as the first element and zero or more arguments. For example, `(foo 1 2)` applies the function `foo` to the arguments `1` and `2`. Conditional expressions follow the same format, e.g., `(if (> x 0) x (- 0 x))` (which returns the absolute value of `x`), as do function definitions:

```
(defun foo (a b) (* (+ a b) 5))
```

This function sums the two arguments `a` and `b` and multiplies the result by 5. Note that no explicit return statement is necessary, as the body of the function is an expression which evaluates to an integer value. It is important to note that all expressions are functions, including arithmetic operators such as `+` and `*`. Declarations of recursive functions do not require any special constructs:

```
(defun foo (x) (if (> x 0) (+ x (foo (- x 1))) 0))
```

Classically, the only element for building data structures is a pair of values a and b, usually written as a **.** b and created by the function `cons` (the tuple is often called a cons pair). Its elements are accessed using the functions car (first element) and cdr (second element). Lists are the most important (and defining) data type in Lisp. They are constructed as a sequence of cons pairs, where the first element of the pair contains the list element, and the second contains either a reference to the next cons pair or nil (the empty element), e.g., 1 . (2 . (3 . nil)). Likewise, binary trees are built from cons pairs as nodes containing references to the child elements, and n-ary trees can be built by treating each tree node as a list.

Any value which is not a cons pair is an atom. This includes symbols, which are essentially names which can be treated as unique values, but have no further inherent meaning. By convention, the symbol t stands for *true* and nil for *false* as well as "no value" and "empty list". It is important to note that checks (such as if) consider any value other than nil to be true, not only t. Literals such as integers, rational numbers and characters are also atoms.

A preceding single quote (') indicates that the following code should be taken as a literal S-expression (i.e., "data") rather than being evaluated as a function. This is used for symbols (marking them as uninterpreted values, such as 'foo, rather than attempting to look up a variable named foo), and for building constant lists/structures, as in '(1 (2 3) 4) (equivalent to building structures dynamically using cons).

let expressions define new variables which are visible within the given body, as in

```
(defun foo (x)
  (let ((y (bar x)) (z (baz x 10)))
    (qux y (* y 2) z (+ z 5))))
```

(qux is some other function). Not only does this result in more compact code, it also means that bar and baz are evaluated only once each, even though their results are used twice later.

Function arguments are generally evaluated eagerly (i.e., without respect to whether they will actually be required for further calculations). The one important exception is if, where the true and false branches will only be evaluated if the condition is true or false respectively.

Lisp is not statically typed, nor does it support explicit type annotations. However, values *are* typed, allowing types to be checked at runtime using functions such as integerp and consp, and thus enable type-dependent behavior.

2.1.1 Limited subset of Lisp

The ACL2 input language is a limited subset of Common Lisp [Ste90], a popular dialect and extension of Lisp. The semantics underlying the language and its predefined functions are largely the same, but higher-order functions, as well as advanced features

$$\begin{aligned}
\mathit{program} &:= \mathit{fndef}^* && (2.1)\\
\mathit{fndef} &:= \texttt{(defun } \mathit{sym}\ \texttt{(}\mathit{sym}^*\texttt{)}\ \mathit{sexpr}\texttt{)} && (2.2)\\
\mathit{sexpr} &:= \texttt{(}\mathit{sym}\ \mathit{sexpr}^*\texttt{)} && (2.3)\\
&\ \texttt{(if }\ \mathit{sexpr}\ \mathit{sexpr}\ \mathit{sexpr}\texttt{)} && (2.4)\\
&\ \texttt{(let ((}\mathit{sym}\ \mathit{sexpr}\texttt{)}^*\texttt{)}\ \mathit{sexpr}\texttt{)} && (2.5)\\
&\ \mathit{sym} && (2.6)\\
&\ \mathit{literal} && (2.7)\\
\mathit{literal} &:= \mathit{integer\ literal} && (2.8)\\
&\ \texttt{'}\ \mathit{lit\text{-}sexpr} && (2.9)\\
\mathit{lit\text{-}sexpr} &:= \mathit{integer\ literal} && (2.10)\\
&\ \mathit{sym} && (2.11)\\
&\ \texttt{(}\mathit{lit\text{-}sexpr}^*\texttt{)} && (2.12)
\end{aligned}$$

Figure 2.1: Grammar of the simplified input language which is generated by a preprocessing step and processed by our tool.

specific to Common Lisp (such as structures implemented as special constructs not based on `cons`), are not supported by ACL2.

In the interest of simplicity, we base our input on the simplified Lisp syntax used internally by ACL2 and define some additional restrictions. Most notably, we exclude other value types, such as strings and rational numbers. Fig. 2.1 shows the definition of the language which is processed by our tool. *fn* stands for the name of the function, such as `foo`, which is applied to the arguments. A *constant* may be either an atom (integer literal or quoted symbol), or a list literal (which is equivalent to a series of nested `cons` pairs). Additional constructs, such as `let` and `cond` (a conditional form which checks multiple cases sequentially, essentially a nested if ... else if ... else if ...), are converted to this form automatically and can therefore be processed correctly, but do not need to be considered specifically for further analysis.

2.1.2 Syntax

Eq. (2.2) a function definition, consisting of a function name, a list of argument names, and a body.

Eq. (2.3) an application of a function (identified by a symbol) to a list of argument values.

Eq. (2.4) a conditional expression, consisting of an expression representing the condition, and two branches. The expression evaluates the second branch if the value of the condition is `nil`, and to the first branch otherwise.

Eq. (2.5) evaluates an s-expression under the given list of new assignments of values to symbols (variable names).

Eq. (2.6) a symbol name which evaluates to the value assigned to the corresponding variable.

Eq. (2.7) a literal.

Eq. (2.8), Eq. (2.10) a literal integer value.

Eq. (2.9) an uninterpreted expression, which is treated as a data value rather than being evaluated.

Eq. (2.11) an uninterpreted symbol.

Eq. (2.12) a literal s-expression, which is a list (equivalent to the result of nested `cons` applications). The value of an empty list is the symbol `nil`.

2.1.3 Values

$$\begin{aligned} \mathit{val} \;:=\; & \mathit{integer} && (2.13)\\ & \mathit{sym} && (2.14)\\ & \mathit{val}\ \texttt{.}\ \mathit{val} && (2.15)\end{aligned}$$

Eq. (2.13) a signed integer of arbitrary size.

Eq. (2.14) a symbol.

Eq. (2.15) a `cons` pair. A proper list is a nested tree of pairs where the rightmost element is the symbol value `nil` (therefore, `nil` is an empty list).

2.1.4 Semantics

Function application

BF is the set of basic functions. Any function not in this set is a normal function which, if called, must be defined in the program P. It has a list of argument names $\vec{an}$ and a *body*. For a function application, each argument expression args_i is evaluated to a value av_i, and these values are assigned to the argument names in a new variable assignment α', under which the body is evaluated, yielding the final result r. The total cost is the sum of the cost of a function application itself (c_{app}), the evaluation of the body and of each argument.

$$\text{L-app}\;\frac{\begin{array}{ccc} P[\mathit{fn}] = \texttt{(defun}\ \mathit{fn}\ \ \vec{an}\ \ \mathit{body}\texttt{)} & n = |\vec{an}| & \mathit{args}_i \Downarrow^{\alpha}_{c_i} \mathit{av}_i\ \forall i \in 1..n \\ \alpha' = [\mathit{an}_i \mapsto \mathit{av}_i \mid i \in 1..n] & \mathit{body} \Downarrow^{\alpha'}_{c} r & \mathit{fn} \notin \mathit{BF} \end{array}}{\texttt{(}\mathit{fn}\ \ \vec{\mathit{args}}\texttt{)} \Downarrow^{\alpha}_{c_{\text{app}}+c+c_1+\cdots+c_n} r}$$

Basic functions such as `+` and `car` are defined directly and do not have a (Lisp) body, i.e., do not depend on any other code. Otherwise, an application of a basic function is similar to a normal function:

$$\text{L-app-bf}\ \frac{\begin{array}{c} \mathit{args}_i \Downarrow^{\alpha}_{c_i} \mathit{av}_i\ \forall i \in 1..n \\ \alpha' = [\mathit{an}_i \mapsto \mathit{av}_i \mid i \in 1..n] \qquad \mathit{BF}[\mathit{fn}] = \langle n, f, c\rangle \end{array}}{(\mathit{fn}\ \ \vec{\mathit{args}}) \Downarrow^{\alpha}_{c+c_1+\cdots+c_n} f(\mathit{av}_1, \ldots, \mathit{av}_n)}$$

The set of basic functions *BF* is described as a mapping of function names to triples $\langle n, f, c\rangle$ of a (mathematical) function f with n arguments (which are Lisp values), which describes the behavior of the basic function and returns another Lisp value, and a cost c assigned to the basic function:

$$\mathit{BF} = \begin{cases} \texttt{+} & \mapsto \langle 2, f(x_1, x_2) = x_1 + c_2, c_{\texttt{+}}\rangle \\ \texttt{-} & \mapsto \langle 2, f(x_1, x_2) = x_1 - c_2, c_{\texttt{-}}\rangle \\ \texttt{car} & \mapsto \langle 1, f(x_1) = \begin{cases} a & \text{if } x_1 = a\ \texttt{.}\ b \\ \texttt{nil} & \text{if } x = \texttt{nil} \end{cases}, c_{\texttt{car}}\rangle \\ \texttt{cdr} & \mapsto \langle 1, f(x_1) = \begin{cases} b & \text{if } x_1 = a\ \texttt{.}\ b \\ \texttt{nil} & \text{if } x = \texttt{nil} \end{cases}, c_{\texttt{cdr}}\rangle \\ \texttt{cons} & \mapsto \langle 2, f(x_1, x_2) = x_1\ \texttt{.}\ x_2, c_{\texttt{cons}}\rangle \end{cases}$$

Only a few basic functions are listed here, the rest are defined analogously. Note that undefined behavior (e.g., applying `car` to an integer) is not permitted.

`if`

In an `if` expression, either branch a or b is evaluated, depending on the value of the condition x:

$$\text{L-if}\ \frac{x \Downarrow^{\alpha}_{c_x} v_x}{\text{L-if-true}\ \dfrac{v_x \neq \texttt{nil} \qquad a \Downarrow^{\alpha}_{c_a} v_a}{(\texttt{if}\ \ x\ \ a\ \ b) \Downarrow^{\alpha}_{c_{\text{if}}+c_x+c_a} v_a} \qquad \text{L-if-false}\ \dfrac{v_x = \texttt{nil} \qquad b \Downarrow^{\alpha}_{c_b} v_b}{(\texttt{if}\ \ x\ \ a\ \ b) \Downarrow^{\alpha}_{c_{\text{if}}+c_x+c_b} v_b}}$$

`let`

A `let` expression is very similar to a function application, except that the variable assignment α' passed to the body is an extension rather than a complete replacement of the initial assignment α, and the body is defined within the expression itself rather than as a separate function.

$$\text{L-let}\frac{\begin{array}{c} n = |\vec{defs}| \qquad (defs_i = \texttt{(}dn_i \;\; de_i\texttt{)} \land de_i \Downarrow^{\alpha}_{c_i} dv_i)\ \forall i \in 1..n \\ \alpha' = \alpha \leftarrow [dn_i \mapsto dv_i \mid i \in 1..n] \qquad\qquad body \Downarrow^{\alpha'}_{c} r \end{array}}{\texttt{(let}\ \vec{defs}\ \ body\texttt{)} \Downarrow^{\alpha}_{c_{\text{let}}+c+c_1+\cdots+c_n} r}$$

$\alpha_1 \leftarrow \alpha_2$ denotes the combination of variable assignments α_1 and α_2, where names which exist in both assignments are reassigned to the values in α_2.

Variable reference

Referencing a variable simply retrieves the value v of variable x from the variable assignment α. Note that we admit only valid programs, so references to undefined variables cannot occur.

$$\text{L-var}\frac{v = \alpha[x]}{x \Downarrow^{\alpha}_{c_{\text{var}}} v}$$

Literal values

Literals are even simpler, merely returning the literal value v:

$$\text{L-lit}\frac{}{v \Downarrow^{\alpha}_{c_{\text{lit}}} v}$$

2.2 Cost relations

The programs processed by our analysis are expressed in the simplified Lisp subset described above. However, CoFloCo (the bound analysis tool which we build on) expects its input to be represented as systems of so-called cost relations, based on which it then calculates resource bounds (see Chapter 4).

A cost relation is an abstraction of a function which represents the function's cost and behavioral semantics. The details of expressing Lisp programs through cost relations are described in Section 2.5.

A cost relation consists of one or more cost equations, which are 4-tuples of the form

$$\text{eq}(fn(\vec{a}), c, \vec{calls}(\vec{a} \cup \vec{v}), \vec{constrs}(\vec{a} \cup \vec{v}))$$

$\vec{a}$ is a list of integer variables representing the arguments of the function predicate (such that a_i is the i-th argument), and $\vec{v}$ is a set of new variables which are internal to the cost equation.

- $fn(\vec{a})$ is a term describing the function's signature (i.e., its name fn and list of parameters $\vec{a}$). Note that these signatures take the form of logical predicates – thus, there is no special return value.

- c is the (constant) cost associated with the function itself (specifically, not including the cost of any other function calls).
- $\vec{calls}(\vec{a} \cup \vec{v})$ is a (possibly empty) set of function calls made by the function *fn*. The parameters of the calls are in $\vec{a} \cup \vec{v}$, i.e., each call has the structure $fn'(\vec{p})$ where fn' is the function being called with a list of parameters $\vec{p} \subseteq \vec{a} \cup \vec{v}$. The total cost $C(ce)$ for a cost equation $cr = \text{eq}(..., c, \vec{calls}, ...)$ is then defined as

$$c + \sum_{\gamma \in \vec{calls}} C(\gamma)$$

  with $C(\gamma)$ being the cost calculated for each call γ.
- $\vec{constrs}(\vec{a} \cup \vec{v})$ is a (possibly empty) set of linear constraints over $\vec{a} \cup \vec{v}$. The constraints limit the combinations of argument values to which the cost equation is applicable.

While in cost relations, function signatures are specified as predicates, the analysis described here processes code written in Lisp, a language based on functions rather than predicates. For the sake of clarity, representations of cost relations shown in this work therefore split the input and output parameters of the function signatures, as in $fn(\vec{in} \to \vec{out})$, where $\vec{in}$ are the (input) arguments and $\vec{out}$ are return values. However, this difference affects only the representation, and is equivalent to the definition given above with $\vec{a} = \vec{in} \cup \vec{out}$.

A cost relation is a set of one or more cost equations for the same function, i.e., where the function signature (meaning the function name and the number of parameters) is the same.

In brief, constraints and function calls together define the semantics of the function in relation to the arguments, while the combination of the cost of the function itself and that of its function calls define the function's cost semantics.

As a simple example, consider a cost equation for a function which increments its integer argument by 1:

```
(defun incr (x) (+ x 1))
```

The cost relation for this function is

$$\text{eq}(\texttt{incr}(a \to r), 1, [\texttt{+}(a, 1 \to r)], [\,])$$

Note that the addition is a function call. The cost has been defined as 1 here, as a function application counts as one execution step. However, the cost of course depends on the cost metric. With the function application metric, the cost cannot be negative, but in general, cost relations can also specify a negative cost.

Further note that the direct inclusion of the constant argument 1 in the call `+`$(a, 1 \rightarrow r)$ is a shorthand for `+`$(a, x \rightarrow r)$ with the constraint $x = 1$, with the equivalent cost equation

$$\mathrm{eq}(\texttt{incr}(a \rightarrow r), 1, [\texttt{+}(a, x \rightarrow r)], [x = 1])$$

Alternative behaviors of a function can be modeled by specifying a cost relation consisting of more than one cost equation for the same function, but with different constraints, as in:

$$\mathrm{eq}(\texttt{abs}(x \rightarrow x), 1, [\,], [x \geq 0])$$
$$\mathrm{eq}(\texttt{abs}(x \rightarrow r), 1, [\texttt{-}(x \rightarrow r)], [x < 0])$$

In this example, the constraints restrict the applicability of the cost equations to arguments with positive and negative value respectively. In the second cost equation, the unary negation is also a function call.

If the cost equations of a cost relation are not fully mutually exclusive, this is treated as nondeterminism, and the option which maximizes the (upper) bound will be selected by the analysis. This is used for specifying behavior which cannot be fully defined in cost relations, such as equality checking of lists.

As described above, calls to other functions are declared in a list within the cost equation(s). Nested function calls (where a function is called with an argument which is the result of another function call) are contained in the same list, and new variables are introduced for the return values. For example, a function which calls another function on the result of the function `incr` (defined as in the previous example), such as

```
(defun incr-double (x) (* (incr x) 2)
```

has the cost relation

$$\mathrm{eq}(\texttt{incr-double}(a \rightarrow r), 1, [\texttt{incr}(a \rightarrow x), \texttt{*}(x, 2 \rightarrow r)], [\,])$$

x is a new variable which represents the return value of the call to `incr`, which is then multiplied by 2. For recursive functions, the recursive call is represented in the same way as any other function call.

`+` and `*` are defined as *basic functions*. A basic function is one which does not call other functions or itself recursively (it is “atomic”), and whose cost therefore does not depend on that of any other function. Instead, the cost is defined explicitly as part of the desired cost model. For the Lisp frontend, basic functions are typically Common Lisp and ACL2 built-ins such as `car`, `>=` and `eq`. In the examples above, `abs`, `incr` and `incr-double` are non-basic functions, as they call other functions. All non-basic functions are compositional and built from basic ones, meaning that their cost can be analyzed given that of the basic functions. Therefore, the cost values assigned to the basic functions, together with the specified cost behavior of language elements such as conditions and function calls, define the cost model of the analysis.

2.2.1 Semantics of cost relations

$$\text{CR-eval}\;\frac{\begin{array}{c}\mathrm{eq}(\mathit{fn}(\vec{a}), c, \vec{\mathit{calls}}, \vec{\mathit{constrs}}) \qquad n = |\vec{\mathit{calls}}| \\ \vec{a} = [a_1, \dots, a_m] \qquad \alpha = [a_i \mapsto x_i \mid i \in 1..|\vec{a}|] \\ \vec{b_i} = [b_{i,j} \mid j \in 1..|\vec{b_i}|] \quad \vec{v} = \bigcup_{i=1}^{n} \vec{b_i} \setminus \vec{a} \quad \beta = [v_i \mapsto \chi_i \mid i \in 1..|\vec{v}|] \\ \vec{y_i} = \left[(\alpha \cup \beta)\,[b_{i,j}] \mid j \in 1..|\vec{b_i}|\right] \quad \forall i \in 1..n \\ \mathit{fn}_i(\vec{y_i}) \downarrow c_i \text{ where } \mathit{calls}_i = \mathit{fn}_i(\vec{b_i}) \text{ and } \mathrm{eq}(\mathit{fn}_i(\vec{d_i}), \dots) \quad \forall i \in 1..n \\ \alpha \cup \beta \models \varphi \quad \forall \varphi \in \vec{\mathit{constrs}}\end{array}}{\mathit{fn}(x_1, \dots, x_m) \downarrow c + c_1 + \dots + c_n}$$

The operation $\mathit{fn}(x_1, \dots, x_n) \downarrow c$ evaluates a cost relation call for a function fn with argument values $x_1, \dots, x_n$ to a cost c. Note that the evaluation (and thus the cost c) is nondeterministic, since more than one cost equation might be applicable for some cost relation in the evaluation. The cost of an infeasible call (i.e., with a combination of argument values such that none of the cost equations of some subsequently called cost relation are applicable) is undefined.

Note again that the $(\vec{in} \to \vec{out})$ syntax for function arguments is merely a syntactic convention intended to clarify the meaning of the arguments in the context of Lisp. The list $\vec{a}$ of predicate arguments is equivalent to the concatenation of the two, i.e., $\vec{a} = \vec{in} \,||\, \vec{out}$, where $||$ is the concatenation of lists. This notation does not affect the semantics of the cost relations, and is omitted here for simplicity.

The set of internal variables $\vec{v}$ consists of additional (new) variables other than the arguments $\vec{a}$. β is an assignment of these variables to some integer values χ_i such that, together with α, all constraints are fulfilled, and all calls have valid evaluations.

Note that the names of the argument variables of the callee cost relations for fn_i are not relevant, since arguments are matched by position, not name (hence the names in $\vec{d_i}$ are not necessarily equal to $\vec{b_i}$).

Constraints

Cost relations may contain only linear constraints, that is, linear relations (equality, less-or-equal and greater-or-equal) between linear expressions. Constraints φ hold under a variable assignment α, that is, $\alpha \models \varphi$:

$$\text{CO-eq}\;\frac{x_1 =^{\alpha} v_1 \qquad x_2 =^{\alpha} v_2 \qquad v_1 = v_2}{\alpha \models x_1 = x_2}$$

$$\text{CO-leq}\;\frac{x_1 =^{\alpha} v_1 \qquad x_2 =^{\alpha} v_2 \qquad v_1 \leq v_2}{\alpha \models x_1 \leq x_2} \qquad \text{CO-geq}\;\frac{x_1 =^{\alpha} v_1 \qquad x_2 =^{\alpha} v_2 \qquad v_1 \geq v_2}{\alpha \models x_1 \geq x_2}$$

The available linear expressions are addition, subtraction, negation, multiplication by a constant, accessing a variable of the cost equation, and a constant integer value. A linear expression is equal to an integer value under a variable assignment α (here written $=^\alpha$):

$$\text{CX-add}\frac{x_1 =^\alpha v_1 \quad x_2 =^\alpha v_2 \quad y =^\alpha v_1 + v_2}{x_1 + x_2 =^\alpha y}$$

$$\text{CX-sub}\frac{x_1 =^\alpha v_1 \quad x_2 =^\alpha v_2 \quad y =^\alpha v_1 - v_2}{x_1 - x_2 =^\alpha y}$$

$$\text{CX-neg}\frac{x_1 =^\alpha v_1 \quad y =^\alpha -v_1}{-x_1 =^\alpha y} \qquad \text{CX-mul}\frac{x_1 =^\alpha v_1 \quad y =^\alpha c * v_1}{c * x_1 =^\alpha y}$$

$$\text{CX-var}\frac{\alpha[x] = v}{x =^\alpha v}$$

$$\text{CX-const}\frac{}{c =^\alpha c}$$

2.3 Measures

So far, only pure integer programs have been considered, for which a direct representation in equivalent cost relations was described. However, this is clearly not sufficient for a language such as Lisp, where programs make heavy use of data structures (in the case of Lisp, `cons` pairs). Consider the function `listlen` in Listing 2, for example: clearly, its cost is proportional to the length of its list argument.

```
(defun listlen (x)
  (if (consp x)
    (+ 1 (listlen (cdr x)))
    0))
```

Listing 2: A function which determines only the length of a list, without regard to the individual elements.

However, the analysis performed by CoFloCo handles only integer variables and cannot directly deal with functions such as `listlen`, whose cost depends on properties of structures. Therefore, structures must be mapped to integers, which can then be represented in cost relations and thus enables analysis of non-integer programs. This is done using *measures* (also called *norms*) – functions which describe important characteristics of structural values by mapping them into integers. These measures are then used within cost relations as regular integer variables, meaning that CoFloCo does not require any specific support for structural values. Each Lisp variable is then represented by several measure variables in the cost relations.

For this work, we use a fixed set of three measures:

- integer
- list length (the number of `cons` pairs when following only the `cdr`)

$$
\begin{aligned}
\mathit{int}(x) &= \begin{cases} x & \text{if } x \text{ is an integer} \\ 0 & \text{if } x = \texttt{nil} \\ 1 & \text{if } x = \texttt{t} \\ \mathit{sym\_id}(x) & \text{if } x \text{ is a symbol} \\ \bot & \text{otherwise} \end{cases} \\
\mathit{length}(x) &= \begin{cases} 1 + \mathit{length}(b) & \text{if } x = a\ \texttt{.}\ b \\ 0 & \text{otherwise} \end{cases} \\
\mathit{size}(x) &= \begin{cases} 1 + \mathit{size}(a) + \mathit{size}(b) & \text{if } x = a\ \texttt{.}\ b \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 2.2: The formal definitions of the measures.

- structure size (i.e., the number of `cons` pairs within the entire structure, also called *term-size* [AGG13])

The formal definitions of the measures are given in Fig. 2.2.

It is important to note that these are indeed (signed) integers, and not limited to natural numbers, which is necessary for analyzing programs whose bound depends on signed integers. However, the size and length measures are of course restricted to positive values, which is explicitly enforced through constraints added to the cost relations which describe the basic functions.

From the definitions, it immediately follows that the length is always less than or equal to the size, since the former counts only a subset of `cons` pairs. `nil`, which is always the second element of the last `cons` pair in a well-formed list, has a length and size of 0 according to the definitions, which leads to the equivalence with the usual and intuitive understanding of list length. It is also important to note that the integer measure is undefined for lists.

Symbols, including the booleans values `t` and `nil`, are mapped to integers by the injective function *sym_id*. `t` and `nil` are always assigned the integer values 1 and 0 respectively, while all other symbols are uniquely assigned successive natural numbers ≥ 2 by *sym_id*, in the order in which they are encountered within the program (the actual value is not relevant, as long as exactly one unique integer value is assigned to each symbol).

In this document, we use the convention of identifying the measures of a variable x by subscripts x_{int}, x_{len} and x_{size} for integer value, list length and size respectively. The symbol $\bot$ stands for undefined values, such as the integer value of a list.

In the following, a few examples of the representation of Lisp values by measures are shown (note that literal lists `'(...)` are equivalent to nested `cons` pairs):

$$\begin{aligned}
\texttt{42} &\rightarrow x_{\text{int}} = 42, x_{\text{len}} = x_{\text{size}} = 0 \\
\texttt{'(5 6 7)} &\rightarrow x_{\text{int}} = \bot, x_{\text{len}} = x_{\text{size}} = 3 \\
\texttt{'(2 (3 (4 5)) 6)} &\rightarrow x_{\text{int}} = \bot, x_{\text{len}} = 3, x_{\text{size}} = 7 \\
\texttt{t} &\rightarrow x_{\text{int}} = 1, x_{\text{len}} = x_{\text{size}} = 0
\end{aligned}$$

Some types of values, such as rational and floating-point numbers, are not supported in CoFloCo and therefore cannot be handled. Strings are also not supported, but adding a measure for them (such as string length) should not be an issue – strings could be treated in a very similar way to lists, or even considered a special case of lists.

As an example, the following cost relations describe the function `listlen` (Listing 2) with measures:

$$\begin{aligned}
&\text{eq}(\texttt{if1}(x_{\text{len}} \rightarrow r_{\text{int}}), 1, [\texttt{consp}(x_{\text{len}} \rightarrow c_{\text{int}}), \texttt{cdr}(x_{\text{len}} \rightarrow y_{\text{len}}), \texttt{listlen}(y_{\text{len}} \rightarrow rs_{\text{int}}), \\
&\quad \texttt{+}(1, rs_{\text{int}} \rightarrow r_{\text{int}})], [c_{\text{int}} = 1]) \\
&\text{eq}(\texttt{if1}(x_{\text{len}} \rightarrow 0), 1, [\texttt{consp}(x_{\text{len}} \rightarrow c_{\text{int}})], [c_{\text{int}} = 0]) \\
&\text{eq}(\texttt{listlen}(x_{\text{len}} \rightarrow r_{\text{int}}), 1, [\texttt{if1}(x_e ll \rightarrow r_{\text{int}})], [\,])
\end{aligned}$$

Note that this representation omits variables for irrelevant measures in order to simplify the cost relations (Section 3.2.1 describes how a very similar step is performed as part of the actual analysis).

2.4 Bounds

A (sound) upper resource bound for a Lisp expression e is a closed-form expression $b([a_{i\text{int}}, a_{i\text{len}}, a_{i\text{size}} \mid i \in 1..n])$ over the measures for the variables $\Theta = \{a_1, \ldots, a_n\}$ which are visible within e, such that, for any variable assignment

$$\alpha = [a_i \mapsto v_i \mid i \in 1..n]$$

which is valid for e, if the Lisp evaluation of e has cost c (i.e., $e \Downarrow_c^\alpha \ldots$), then the bound expression b evaluates to a cost value c_b given the corresponding values for the measures:

$$c_b = b([\mathit{int}(v_i), \mathit{length}(v_i), \mathit{size}(v_i) \mid i \in 1..n])$$

such that $c_b \geq c$.

We assume that the calculation of bound expressions b from cost relation systems, which is performed by CoFloCo, is correct (proofs are provided by Flores-Montoya and Hähnle [FH14]). That is, if $e \rightsquigarrow^\Theta \text{eq}(f(\ldots), \ldots)$ (i.e., the expression e is translated to a cost equation for a function f corresponding to e), then for all possible executions $f(v_{1\text{int}}, v_{1\text{len}}, v_{1\text{size}}, \ldots, r_{\text{int}}, r_{\text{len}}, r_{\text{size}}) \downarrow c_{cr}$ of the cost relation, it holds that $c_b \geq c_{cr}$.

Thus (given the correctness of the translation steps, which is proved in Section 2.5), the expression b calculated for the generated cost relation for f is sound, meaning that the cost c_b it evaluates to is an overapproximation of the cost c of the original Lisp code:

$$c_b \geq c$$

2.5 Translating Lisp to cost relations

In order to calculate bounds using CoFloCo, the Lisp code must be translated to cost relations.

The translation from a Lisp expression e to a cost relation (one or more cost equations) cr is written as $e \leadsto^{\Theta} cr$, where Θ is the set of Lisp variables which are visible within e. In the interest of simplicity, we add cost relation variables for the entire set Θ, rather than determining which variables are actually required (as explained in Section 3.2.1, superfluous variables can be removed later).

$\mathrm{mn}(\Theta)$ generates names for measure variables, i.e., cost relation variables corresponding to the measures of the set Θ of Lisp variables: $\mathrm{mn}(\Theta) = [x_{\mathrm{int}}, x_{\mathrm{len}}, x_{\mathrm{size}} \mid x \in \Theta]$.

Correctness invariant

Of course, the behavior and cost of the original code should be soundly represented in the result of the translation. This is formally expressed by the following invariant, which must be shown to hold for each generated cost relation (i.e., in each translation step):

For any valid Lisp expression e, if it evaluates to a value v under cost c: $e \Downarrow_c^{\alpha} v$, and a translation $e \leadsto^{\Theta} ce$ where $ce = \mathrm{eq}(f(\dots), \dots, \dots, \dots)$ with $\Theta = \mathrm{names}(\alpha)$ exists, then there is at least one execution of the generated cost equation ce whose cost c_{cr} is at least as high as the cost c of the Lisp evaluation:

$$f(\vec{mv} \to \vec{rv}) \downarrow c_{cr} \text{ such that } c_{cr} \geq c$$

with input values $\vec{mv}$ (the measures of the Lisp value assignments in α) and output values $\vec{rv}$ (the measures of the return value v):

$$\begin{aligned} \vec{mv} &= [\mathit{int}(\alpha[x]), \mathit{length}(\alpha[x]), \mathit{size}(\alpha[x]) \mid x \in \Theta] \\ \vec{rv} &= [\mathit{int}(v), \mathit{length}(v), \mathit{size}(v)] \end{aligned}$$

Since the cost specified by the bound expression is at least as high as the maximum of the cost c_{cr} of *any* possible execution (see Section 2.4), including the (at least) one execution with $c_{cr} \geq c$, it follows that if this invariant holds for all steps in a translation, the bound will also be at least as high as the cost c of the Lisp evaluation. Thus, as stated in Section 2.4, proving the correctness of the translation also ensures that any bound which is sound for the translated cost relations is also sound for the original Lisp expression.

It is important to note that in the rest of this work, some degree of simplification of the generated cost relations ("inlining" of calls, see Section 2.6) is implicitly assumed. Separate cost relations are generally retained only for the called bodies in function applications, and for the branches in `if` statements.

2.5.1 Function application

$$\text{T-app}\ \frac{\begin{array}{c} n = |\vec{args}| \quad P[fn] = \texttt{(defun } fn\ \Theta'\ body\texttt{)} \quad \vec{m} = \text{mn}(\Theta) \quad \vec{m}' = \text{mn}(\Theta') \\ args_i \leadsto^{\Theta} \text{eq}(f_i(\vec{m} \to \vec{s_i}), c_i, \ldots, \ldots) \quad \forall i \in 1..n \\ \vec{r_i} = [r_{i\text{int}}, r_{i\text{len}}, r_{i\text{size}}] \quad \forall i \in 1..n \qquad \vec{r} = [r_{\text{int}}, r_{\text{len}}, r_{\text{size}}] \\ body \leadsto^{\Theta'} \text{eq}(fn(\vec{m}' \to \vec{s}), c_b, \ldots, \ldots) \end{array}}{\texttt{(}fn\ \vec{args}\texttt{)} \leadsto^{\Theta} \text{eq}(app_{\#}(\vec{m} \to \vec{r}), c_{\text{app}}, [fn(||_{i=1}^{n} \vec{r_i} \to \vec{r}) \cup \bigcup_{i=1}^{n} f_i(\vec{m} \to \vec{r_i})], [\,])}$$

where $||_i^n$ is the ordered concatenation of lists, $app_{\#}$ is a new name, and the measure variables in $\vec{r}$ and the $\vec{r_i}$ are new variables. The list $\vec{m}'$ contains the cost relation variables corresponding to the measures for each argument in the callee's argument list Θ'.

Of course, the variables in the caller and callee cost relations need not have the same names, hence the outputs $\vec{s}$ of fn need not be equal to the outputs $\vec{r}$ of $app_{\#}$.

Correctness proof

$$\texttt{(}fn\ \vec{args}\texttt{)} \leadsto^{\Theta} \text{eq}(app_{\#}(\vec{m} \to \vec{r}), c_{\text{app}}, [fn(\mathop{||}_{i=1}^{n} \vec{r_i} \to \vec{r}) \cup \bigcup_{i=1}^{n} f_i(\vec{m} \to \vec{r_i})], [\,])$$

For any valid assignment α of values to argument names, and the corresponding list of measure values

$$\vec{mv} = [int(\alpha[x]), length(\alpha[x]), size(\alpha[x]) \mid x \in \text{names}(\alpha)]$$

according to L-app,

$$\texttt{(}fn\ \vec{args}\texttt{)} \Downarrow^{\alpha}_{c_{\text{app}}+cl+cl_1+\cdots+cl_n} v$$

with $body \Downarrow^{\alpha'}_{cl} v$, there must be an evaluation of the generated cost relation function $app_{\#}$ with input values $\vec{mv}$, such that the cost ct of the CR evaluation is at least as high as the cost $c_{\text{app}} + cl + cl_1 + \cdots + cl_n$ of the actual Lisp execution:

$$app_{\#}(\vec{mv} \to \vec{rv}) \downarrow ct \text{ with } ct \geq c_{\text{app}} + cl + cl_1 + \cdots + cl_n$$

and whose output values $\vec{rv}$ are equal to the measures of the Lisp return value v:

$$\vec{rv} = [int(v), length(v), size(v)]$$

By the correctness invariant under the assignment α, it follows from T-app that there exist executions with correct costs for each argument:

$$args_i \Downarrow^{\alpha}_{cl_i} av_i \text{ implies } f_i(\vec{mv} \to \vec{rv}_i) \downarrow cc_i \text{ with } cc_i \geq cl_i \quad \forall i \in 1..n$$

where $\vec{rv_i}$ contains the correct measures for av_i for each $i \in 1..n$:

$$\vec{rv_i} = [\mathit{int}(av_i), \mathit{length}(av_i), \mathit{size}(av_i)] \quad \forall i \in 1..n$$

According to CR-eval and T-app, the total cost ct of evaluating the generated cost relation for $app_{\#}$ is $c_{\text{app}} + cc + cc_1 + \cdots + cc_n$ (c_{app} is the cost parameter of the generated cost equation, cc is the cost of evaluating the cost relation of the body, and cc_i the cost for argument i). Since c_{app} is the same in Lisp and the cost equation, and the cost correctness of the arguments ($cc_i \geq cl_i\, \forall i \in 1..n$) was shown above, it only remains to be shown that $cc \geq cl$.

According to the correctness invariant, it holds that (with $\vec{mv}' = ||_i^n \vec{rv_i}$):

$$fn(\vec{mv}' \to \vec{rv}) \downarrow cc \text{ such that } cc \geq cl$$

since $body \Downarrow_{cl}^{\alpha'} v$ according to L-app and $body \rightsquigarrow^{\Theta'} \mathrm{eq}(fn(\vec{m}' \to \vec{s}), c_b, \ldots, \ldots)$ according to T-app – note that $\mathrm{names}(\alpha') = \vec{an} = \Theta'$ and $\alpha' = [an_i \mapsto av_i \mid i \in 1..n]$ according to L-app.

2.5.2 Basic function application

The translation of basic function applications is again very similar to that of normal functions, except that, rather than translating the callee's body, a predefined cost relation for the basic function fn being called is used. These cost relations are defined in the Section 2.5.3. Note that the application step itself has cost 0, as the cost is defined by the basic function's cost relation.

$$\text{T-app-bf}\ \frac{\begin{array}{c} \vec{m} = \mathrm{mn}(\Theta) \\ args_i \rightsquigarrow^{\Theta} \mathrm{eq}(f_i(\vec{m} \to \vec{s_i}), c_i, \ldots, \ldots) \quad \forall i \in 1..n \\ \vec{r_i} = [r_{i\text{int}}, r_{i\text{len}}, r_{i\text{size}}] \quad \forall i \in 1..n \qquad \vec{r} = [r_{\text{int}}, r_{\text{len}}, r_{\text{size}}] \\ BF[fn] = \langle n, fl, cl \rangle \qquad \mathrm{eq}(fn(\vec{m}' \to \vec{s}), c_b, \ldots, \ldots) \end{array}}{(fn\ \ \vec{args}) \rightsquigarrow^{\Theta} \mathrm{eq}(app_{\#}(\vec{m} \to \vec{r}), 0, [fn(||_{i=1}^{n} \vec{r_i} \to \vec{r}) \cup \bigcup_{i=1}^{n} f_i(\vec{m} \to \vec{r_i})], [\,])}$$

Correctness proof With correctness of the cost relations for the basic functions being established (see Section 2.5.3), the correctness of T-app-bf can be shown.

For a basic function fn so that $BF[fn] = \langle n, fl, clb \rangle$ and for any valid assignment α of values to argument names, and the corresponding list of measure values

$$\vec{mv} = [\mathit{int}(\alpha[x]), \mathit{length}(\alpha[x]), \mathit{size}(\alpha[x]) \mid x \in \mathrm{names}(\alpha)]$$

according to L-app-bf,

$$(fn\ \ \vec{args}) \Downarrow_{clb + cl_1 + \cdots + cl_n}^{\alpha} v \text{ where } v = fl(vl_1, \ldots, vl_n)$$

there must be an evaluation of the generated cost relation function $app_{\#}$ with input values $\vec{mv}$, such that the cost ct of the CR evaluation is at least as high as the cost $clb + cl_1 + \cdots + cl_n$ of the actual Lisp execution:

$$app_{\#}(\vec{mv} \rightarrow \vec{rv}) \downarrow ct \text{ with } ct \geq clb + cl_1 + \cdots + cl_n$$

and whose output values $\vec{rv}$ are equal to the measures of the Lisp return value v:

$$\vec{rv} = [int(v), length(v), size(v)]$$

According to CR-eval and T-app-bf, the total cost ct of evaluating the generated cost relation for $app_{\#}$ is $ct = ccb + cc_1 + \cdots + cc_n$. The proof for the correctness of the costs $cc_1, \ldots, cc_n$ of executing the argument cost relations ($cc_i \geq cl_i$) is identical to that for T-app. As shown in the proof for the basic functions (Section 2.5.3), $ccb = clb$, therefore the cost is correct.

For the output values, it can be proved analogously to the case of T-app that $\vec{rv}_1, \ldots,$ $\vec{rv}_n$ are correct. Furthermore, the correctness of the cost relations for the basic functions is proved in Section 2.5.3, so it directly follows that there is an execution of the basic function fn such that (with $\vec{mv}' = ||_i^n \vec{rv}_i$) $fn(\vec{mv}' \rightarrow \vec{rv})$, establishing the correctness of the output values.

2.5.3 Basic function definitions

A fixed set of cost relations consisting of cost equations describing the behavior of the basic function is added for each translation. Again, cost relations are provided here only for a selection of basic functions, with the rest being defined similarly (in the interest of brevity, only the relevant measures are shown):

$$
\begin{aligned}
&\text{eq}(\texttt{+}(x_{\text{int}}, y_{\text{int}} \rightarrow z_{\text{int}}), c_{\texttt{+}}, [\,], [z_{\text{int}} = x_{\text{int}} + y_{\text{int}}])\\
&\text{eq}(\texttt{-}(x_{\text{int}}, y_{\text{int}} \rightarrow z_{\text{int}}), c_{\texttt{-}}, [\,], [z_{\text{int}} = x_{\text{int}} - y_{\text{int}}])\\
&\text{eq}(\texttt{car}(x_{\text{size}} \rightarrow y_{\text{size}}), c_{\texttt{car}}, [\,], [x_{\text{size}} > 0, y_{\text{size}} \leq x_{\text{size}} - 1])\\
&\text{eq}(\texttt{car}(x_{\text{size}} \rightarrow y_{\text{size}}), c_{\texttt{car}}, [\,], [x_{\text{size}} = 0, y_{\text{size}} = 0])\\
&\text{eq}(\texttt{cdr}(x_{\text{len}}, x_{\text{size}} \rightarrow y_{\text{len}}, y_{\text{size}}), c_{\texttt{car}}, [\,], [x_{\text{size}} > 0, y_{\text{len}} = x_{\text{len}} - 1, y_{\text{size}} \leq x_{\text{size}} - 1])\\
&\text{eq}(\texttt{cdr}(x_{\text{size}} \rightarrow y_{\text{size}}), c_{\texttt{cdr}}, [\,], [x_{\text{size}} = 0, y_{\text{size}} = 0])\\
&\text{eq}(\texttt{cons}(x_{\text{len}}, x_{\text{size}}, y_{\text{len}}, y_{\text{size}} \rightarrow z_{\text{len}}, z_{\text{size}}), c_{\texttt{cons}}, [\,],\\
&\quad [z_{\text{len}} = y_{\text{len}} + 1, z_{\text{size}} = x_{\text{size}} + y_{\text{size}} + 1])
\end{aligned}
$$

Correctness proof The correctness of the cost relations for the basic functions with respect to their Lisp definitions shown in Section 2.1.4 must be verified.

For each fn so that $BF[fn] = \langle n, fl, clb \rangle$ and any combination $vl_1, \ldots, vl_n$ of Lisp values for which $fl(vl_1, \ldots, vl_n) = v$, there must be an execution $fn(\vec{mv} \rightarrow \vec{rv}) \downarrow ccb$ of the cost relation for *fn* (where $\vec{mv} = [int(vl_i), length(vl_i), size(vl_i) \mid i \in 1..n]$) such that $ccb \geq clb$ and $\vec{rv} = [int(v), length(v), size(v)]$.

The cost is correct in all cases, since each cost equation has the same cost as the definition in *BF* and (being a basic function) calls no other functions, thus $ccb = clb$.

To prove that the output values are correct, it must be shown that there exists an execution in which they are equal to the measures of the actual return value.

For `+` and `-`, this is trivial, since all relevant values are integers.

For functions acting on lists, measures must be taken into account. If $x = a\ \texttt{.}\ b$, then according to the definitions of the measures in Section 2.3, $length(x) > 0$, $size(x) > 0$, $length(x) = length(b) + 1$ and $size(x) = size(a) + size(b) + 1$. If $x = \texttt{nil}$, then $length(x) = size(x) = 0$.

The correctness of the output values for `car`, `cdr` and `cons` can now be easily shown. Take `car` as an example, with Lisp value function

$$f(x_1) = \begin{cases} a & \text{if } x_1 = a\ \texttt{.}\ b \\ \texttt{nil} & \text{if } x = \texttt{nil} \end{cases}$$

(meaning that the Lisp evaluation has $v = a$ if $vl_1 = a\ \texttt{.}\ b$, and $v = \texttt{nil}$ if vl_1 is an empty list), and cost relation

$$\begin{aligned} &\text{eq}(\texttt{car}(x_{\text{size}} \to y_{\text{size}}), c_{\texttt{car}}, [\,], [x_{\text{size}} > 0, y_{\text{size}} \leq x_{\text{size}} - 1]) \\ &\text{eq}(\texttt{car}(x_{\text{size}} \to y_{\text{size}}), c_{\texttt{car}}, [\,], [x_{\text{size}} = 0, y_{\text{size}} = 0]) \end{aligned}$$

It must be shown that there is an execution

$$\texttt{car}(\vec{mv} \to \vec{rv}) \text{ with } \vec{mv} = [int(vl_1), length(vl_1), size(vl_1)]$$

such that $\vec{rv} = [rv_{\text{int}}, rv_{\text{len}}, rv_{\text{size}}]$ is $[int(v), length(v), size(v)]$.

This follows from the behavior of the measures described above:

- If $vl_1 = a\ \texttt{.}\ b$, then $size(vl_1) > 0$ and the first cost equation can be applied, where $x_{\text{size}} > 0$ and $y_{\text{size}} \leq x_{\text{size}} - 1$, thus in the output values, $rv_{\text{size}} \leq size(vl_1)$, meaning that there is a possible execution where $rv_{\text{size}} = size(vl_1)$.
- If $vl_1 = \texttt{nil}$, then $size(vl_1) = 0$ and the second cost equation can be applied, where $x_{\text{size}} = 0$ and $y_{\text{size}} = 0$, corresponding to $rv_{\text{size}} = 0 = size(\texttt{nil})$ in the output values.

The proofs for the other basic functions are analogous.

2.5.4 `if` expression

`if` expressions are translated to a cost relation consisting of two separate cost equations, one each for the *true* and *false* branch.

$$\text{T-if}\;\frac{\begin{array}{c}\vec{m} = \text{mn}(\Theta) \qquad \vec{r} = [r_{\text{int}}, r_{\text{len}}, r_{\text{size}}] \qquad \vec{r_x} = [r_{x\text{int}}, r_{x\text{len}}, r_{x\text{size}}] \\ x \rightsquigarrow^{\Theta} \text{eq}(f_x(\vec{m} \to \vec{s_x}), c_x, \vec{calls_x}, \vec{constrs_x}) \\ a \rightsquigarrow^{\Theta} \text{eq}(f_a(\vec{m} \to \vec{s_a}), c_a, \vec{calls_a}, \vec{constrs_a}) \\ b \rightsquigarrow^{\Theta} \text{eq}(f_b(\vec{m} \to \vec{s_b}), c_b, \vec{calls_b}, \vec{constrs_b})\end{array}}{\texttt{(if } x\ a\ b\texttt{)} \rightsquigarrow^{\Theta} \begin{array}{l}\text{eq}(\mathit{if}_{\#}(\vec{m} \to \vec{r}), c_{\text{if}}, [f_x(\vec{m} \to \vec{r_x}), f_a(\vec{m} \to \vec{r})], [r_{x\text{int}} \neq 0]) \\ \text{eq}(\mathit{if}_{\#}(\vec{m} \to \vec{r}), c_{\text{if}}, [f_x(\vec{m} \to \vec{r_x}), f_b(\vec{m} \to \vec{r})], [r_{x\text{int}} = 0])\end{array}}$$

where $\mathit{if}_{\#}$ is a new name, and the measure variables in $\vec{r}$ and $\vec{r_x}$ are new variables. Here as well, the variable names in different cost relations may differ, hence the $\vec{r}$ and $\vec{s}$ are not necessarily equal.

Correctness proof For any valid assignment α of values to argument names, and the corresponding list of measure values

$$\vec{mv} = [\mathit{int}(\alpha[x]), \mathit{length}(\alpha[x]), \mathit{size}(\alpha[x]) \mid x \in \text{names}(\alpha)]$$

according to L-if, where $x \Downarrow^{\alpha}_{cl_x} v_x$, either $\texttt{(if } x\ a\ b\texttt{)} \Downarrow^{\alpha}_{c_{\text{if}}+cl_x+cl_b} v_b$ if $v_x = \texttt{nil}$, or $\texttt{(if } x\ a\ b\texttt{)} \Downarrow^{\alpha}_{c_{\text{if}}+cl_x+cl_a} v_a$ otherwise,

there must be an evaluation of the generated cost relation function $\mathit{if}_{\#}$ with input values $\vec{mv}$, such that the cost ct of the CR evaluation is at least as high as the cost $c_{\text{if}} + cl_x + cl_a$ resp. $c_{\text{if}} + cl_x + cl_b$ of the actual Lisp execution:

$$\mathit{if}_{\#}(\vec{mv} \to \vec{rv}) \downarrow ct \text{ with } ct \geq c_{\text{if}} + cl_x + \begin{cases} cl_b & \text{if } v_x = \texttt{nil} \\ cl_a & \text{otherwise}\end{cases}$$

and whose output values $\vec{rv}$ are equal to the measures of the Lisp return value v_a resp. v_b:

$$\vec{rv} = [\mathit{int}(v), \mathit{length}(v), \mathit{size}(v)] \text{ where } v = \begin{cases} v_b & \text{if } v_x = \texttt{nil} \\ v_a & \text{otherwise}\end{cases}$$

By the correctness invariant under the assignment α, it follows from T-if that there exist executions with correct costs for x and a resp. b:

$$x \Downarrow^{\alpha}_{cl_x} v_x \text{ implies } f_x(\vec{mv} \to \vec{rv_x}) \downarrow cc_x \text{ with } cc_x \geq cl_x$$

$$y \Downarrow^{\alpha}_{cl_y} v_y \text{ implies } f_y(\vec{mv} \to \vec{rv_y}) \downarrow cc_y \text{ with } cc_y \geq cl_y \text{ where } y = \begin{cases} b & \text{if } v_x = \texttt{nil} \\ a & \text{otherwise}\end{cases}$$

where the output values contain the correct measures:

$$\vec{rv}_x = [\mathit{int}(v_x), \mathit{length}(v_x), \mathit{size}(v_x)]$$

$$\vec{rv}_y = [\mathit{int}(v_y), \mathit{length}(v_y), \mathit{size}(v_y)] \text{ where } y = \begin{cases} b & \text{if } v_x = \texttt{nil} \\ a & \text{otherwise} \end{cases}$$

According to CR-eval and T-if, the total cost ct of evaluating the generated cost relation for $\mathit{if}_{\#}$ is

$$ct = c_{\text{if}} + cc_x + \begin{cases} cc_a & \text{if } \delta \models r_{x\text{int}} \neq 0 \\ cc_b & \text{if } \delta \models r_{x\text{int}} = 0 \end{cases} \text{ where } \delta = \{r_{x\text{int}} \mapsto rv_{x\text{int}}, \dots\}$$

δ is the assignment of values $\vec{mv}||\vec{rv}$ to variables $\vec{m}||\vec{r}$. c_{if} is the cost parameter of the generated cost equation, cc_x is the cost of evaluating the cost relation for the condition, and cc_a and cc_b the cost for the *true* and *false* branch respectively. c_{if} is the same in Lisp and the cost equation, and the cost correctness of the branches was shown above.

Therefore, it only remains to be shown that the correct cost equation for $if_{\#}$ is guaranteed to be evaluated, so that $cc_y \geq cl_y$ and the output values are correct, i.e., $\vec{rv}_y = [\mathit{int}(v), \mathit{length}(v), \mathit{size}(v)]$ (which is equivalent to $\vec{rv} = \vec{rv}_y$) for $y = a$ resp. $y = b$.

To do so, it must be shown that the cost equation for the *false* branch is guaranteed to be evaluated if $v_x = \texttt{nil}$, and the one for the *true* branch otherwise:

- If $v_x = \texttt{nil}$, then according to the correctness invariant, $rv_{x\text{int}} = 0$ in some execution, therefore $\delta \models r_{x\text{int}} = 0$ according to CO-eq, CX-var and CX-const, and thus $cc_b \geq cl_b$ and $\vec{rv} = \vec{rv}_b$ according to CR-eval.

- Otherwise, however, the one unsoundness in this translation is encountered: $\delta \models r_{x\text{int}} \neq 0$ may not be fulfilled for assignments α under which $v_x = 0$ (i.e., the condition x evaluates to the integer value 0), since in this case, the correctness invariant only guarantees that there is an execution of f_x such that $rv_{x\text{int}} = 0$ (which, in turn, follows from the definition of measure *int* in Section 2.3, specifically that both $\mathit{int}(0) = 0$ and $\mathit{int}(\texttt{nil}) = 0$). As a result, only the wrong cost equation (the one for the *false* branch, with cost cc_b and output values $\vec{rv}_b$) will provably be evaluated when executing the generated cost relations. This can possibly result in an incorrect cost result if the cost equation for the *true* branch is not evaluated, such as when $cl_b < cl_a$.

  However, as we argue in Section 3.3 (where this problem is explained further), it is unlikely to occur in usual Lisp code. Furthermore, in Section 7.1.4, we propose a way of avoiding this issue and ensuring complete soundness of the analysis.

2.5.5 `let` expression

$$\text{T-let}\;\frac{\begin{array}{c} n = |\vec{defs}| \quad \Theta' = \Theta \cup_{||} [dn_i \mid (dn_i \;\; de_i) \in \vec{defs}] \quad \vec{m} = \text{mn}(\Theta) \quad \vec{m}' = \text{mn}(\Theta') \\ de_i \rightsquigarrow^{\Theta} \text{eq}(f_i(\vec{m} \to \vec{s_i}), c_i, \ldots, \ldots) \text{ where } defs_i = (dn_i \;\; de_i) \quad \forall i \in 1..n \\ \vec{r_i} = [r_{i\text{int}}, r_{i\text{len}}, r_{i\text{size}}] \quad \forall i \in 1..n \qquad \vec{r} = [r_{\text{int}}, r_{\text{len}}, r_{\text{size}}] \\ body \rightsquigarrow^{\Theta'} \text{eq}(fn(\vec{m}' \to \vec{s}), c_b, \ldots, \ldots) \end{array}}{\begin{array}{l} (\texttt{let}\;\; \vec{defs}\;\; body) \rightsquigarrow^{\Theta} \\ \qquad \text{eq}(let_{\#}(\vec{m} \to \vec{r}), c_{\text{let}}, [fn(\vec{m} \mathbin{||} \mathop{||}_{i=1}^{n} \vec{r_i} \to \vec{r}) \cup \bigcup_{i=1}^{n} f_i(\vec{m} \to \vec{r_i})], [\,]) \end{array}}$$

Correctness proof Analogous to T-app, with a minor difference being that the list Θ' of new names is appended to Θ rather than completely replacing it.

2.5.6 Variable reference

$$\text{T-var}\;\frac{x \in \Theta}{x \rightsquigarrow^{\Theta} \text{eq}(var_{\#}(\text{mn}(\Theta) \to x_{\text{int}}, x_{\text{len}}, x_{\text{size}}), c_{\text{var}}, [\,], [\,])}$$

where $var_{\#}$ is a new name, and x_{int}, x_{len} and x_{size} are among the measure variables $\text{mn}(\Theta)$.

Correctness proof Cost correctness is trivially fulfilled, as both the Lisp rule L-var and the cost equation generated in T-var have constant cost c_{var}, and the cost equation has no calls.

The correctness of the output values also arises directly from the definitions: for any α where $x \Downarrow^{\alpha}_{c_{\text{var}}} v$ with $v = \alpha[x]$ according to L-var, and a cost relation call $var_{\#}(\vec{mv} \to \vec{rv})$ where

$$\vec{mv} = [int(\alpha[x]), length(\alpha[x]), size(\alpha[x]) \mid x \in \text{names}(\alpha)]$$

$x \in \Theta = \text{names}(\alpha)$ and thus $x_{\text{int}}, x_{\text{len}}, x_{\text{size}} \in \text{mn}(\Theta)$. $\vec{rv} = [int(v), length(v), size(v)]$ where $v = \alpha[x]$ then follows immediately.

2.5.7 Literal

$$\text{T-lit}\;\frac{\vec{m} = \text{mn}(\Theta) \qquad \vec{r} = [r_{\text{int}}, r_{\text{len}}, r_{\text{size}}]}{x \rightsquigarrow^{\Theta} \text{eq}(lit_{\#}(\vec{m} \to \vec{r}), c_{\text{lit}}, [\,], [r_{\text{int}} = int(x), r_{\text{len}} = length(x), r_{\text{size}} = size(x)])}$$

where $lit_{\#}$ is a new name, and *int*, *length* and *size* are the measures defined in Section 2.3, each of which returns an integer value. The measure variables r_{int}, r_{len} and r_{size} are new variables.

Correctness proof The cost is again trivially correct, as both the Lisp rule L-var and the cost equation generated in T-var have constant cost c_{lit}, and the cost equation has no calls.

For the Lisp evaluation $v \Downarrow^{\alpha}_{c_{\text{lit}}} v$, there must be a corresponding cost relation call $\mathit{lit}_{\#}(\vec{mv} \to \vec{rv})$ such that $\vec{rv} = [\mathit{int}(v), \mathit{length}(v), \mathit{size}(v)]$. Note that no variables are accessed, so the values in α and $\vec{mv}$ are irrelevant.

From the constraints $r_{\text{int}} = \mathit{int}(x), r_{\text{len}} = \mathit{length}(x), r_{\text{size}} = \mathit{size}(x)$ in the cost equation for $\mathit{lit}_{\#}$ (generated in T-lit), rules CO-eq, CX-var and CX-const in conjunction with CR-eval, it follows that $\vec{rv} = [\mathit{int}(x), \mathit{length}(x), \mathit{size}(x)]$, and since $x = v$, the condition is fulfilled.

2.6 Simplification

Cost equations can be simplified by "inlining" calls, that is, removing a call from a cost equation, placing the callee's calls and constraints directly within the caller cost equation, and renaming the variables:

$$\text{S-call}\;\frac{\begin{array}{c} \mathit{fn}'(x_1, \ldots, x_n) = \mathit{calls}_m \qquad \mathrm{eq}(\mathit{fn}'(x'_1, \ldots, x'_n), c', \vec{\mathit{calls}}', \vec{\mathit{constrs}}') \\ \mathit{calls}_s = [\mathit{calls}_1, \ldots, \mathit{calls}_{m-1}, \mathit{calls}_{m+1}, \ldots, \mathit{calls}_n] \cup \vec{\mathit{calls}}'[x'/x] \\ \vec{\mathit{constrs}}_s = \vec{\mathit{constrs}} \cup \vec{\mathit{constrs}}'[x'/x] \end{array}}{\mathrm{eq}(\mathit{fn}(A), c, \vec{\mathit{calls}}, \vec{\mathit{constrs}}) \dashrightarrow \mathrm{eq}(\mathit{fn}(A), c + c', \vec{\mathit{calls}}_s, \vec{\mathit{constrs}}_s)}$$

where $[x'/x]$ renames variables in x' to the corresponding ones in x.

Note that the simplification may result in multiple simplified cost equations being generated, if there are multiple cost equations in the callee's cost relation (as is the case for `if`, which is always translated to a cost relation with two cost equations as shown in Section 2.5.4).

The simplification step $\dashrightarrow$ does not affect the cost or output values, and its correctness follows directly from the semantics of cost relations (see CR-eval in Section 2.2.1).

According to CR-eval, evaluating a callee $f_i(B_i)$ implies (through the argument variables Y_i) equality of the callee's argument variables D_i and the caller's variables B_i which f_i is called on. Since $[x'/x]$ in the simplification renames the variables in $D_m = [x'_1, \ldots, x'_n]$ to $B_m = [x_1, \ldots, x_n]$, this equality is preserved.

For the cost, it therefore holds that

$$c + c_1 + \cdots + c_n = (c + c') + (c_{\vec{\mathit{calls}}'} + c_1 + \cdots + c_{m-1} + c_{m+1} + \cdots + c_n)$$

where $c_m = (c' + c_{\vec{\mathit{calls}}'})$, and $c_{\vec{\mathit{calls}}'}$ is the sum of the costs of evaluating the calls in $\vec{\mathit{calls}}'$.

The argument values of the call are correct as well, since (as demonstrated above) the variables are correctly renamed and thus the values are preserved.

2.7 Type-dependent bounds

Lisp is not statically typed, and as a consequence, arguments passed to functions may generally be of any type. Function behavior may be type-dependent, or certain argument types may simply trigger an error resulting in the evaluation being aborted. Our approach can obtain correct results for functions with type-dependent behavior (where the type is not known) by calculating the bound as the maximum of the options, and using different measures to express the respective cost. As an example, see `type-dependent` in Listing 3, which calculates a factorial if the argument is an integer, and the list length otherwise. Thus, the cost depends on the integer value of `x` (if it is an integer) or its list length (if it is a list). The bound calculated for `type-dependent` is $\max([5 * x_{\text{len}}, 1, \text{nat}(x_{\text{int}}) * 6 + 1]) + 6$.

```
(defun factorial (n)
  (if (> n 0)
    (* n (factorial (- n 1)))
    1))

(defun type-dependent (x)
  (if (integerp x)
    (factorial x)
    (listlen x)))
```

Listing 3: The cost of `type-dependent` depends on the type of `x`, i.e., whether it is an integer or a list. The definition of `listlen` is given in Listing 2.

2.8 Cost models

The previous sections, such as the translation specification in Section 2.5, were kept very general, with cost parameters such as c_{app} for the cost of a function application, or $c_{\texttt{car}}$ for the basic function `car`.

However, as explained previously (see Chapter 1), the cost metric used in this work counts only the number of evaluation steps (function calls and `if` decisions). Despite its simplicity, this metric is sufficient for describing how often any given part of the program is executed: since conditional branches are translated as separate cost relations, each cost equation corresponds to an execution of all calls it contains.

Thus, the core problem of counting the number of recursive calls of functions in a functional program has been solved. Based on this foundation, the analysis could be extended to other kinds of resource metrics (such as peak memory consumption, number of memory accesses, number of times a specific function is called, etc.) with relative ease. The only necessary modification would be to specify the costs for basic functions and syntactic constructs (`if` et al.).

CHAPTER 3

Implementation

The previous chapter described the basics of cost relations, and how they are generated from Lisp programs. In the following, implementation details of the analysis, including specific issues and solutions, will be discussed.

3.1 Preprocessing

The preprocessing step uses existing ACL2 code (specifically a component of the CCG termination analyzer) to preprocess Lisp input into the simplified (or normalized) form described in Section 2.1.1. This step removes comments and whitespace, expands macros, and converts `cond` blocks to nested if statements. The resulting code is easier to parse and process, as it is no longer necessary to handle various different control structures, let forms etc. ACL2 also checks for certain kinds of errors, such as use of undefined variables and functions. Given that the code performing this preprocessing step is an integral part of ACL2, a mature and well-tested application, we assume that it correctly preserves the semantics of the program (and thus its cost).

ACL2 input is structured into "books" (essentially Lisp files), which contain functions, proof obligations etc.. Books include both extensive libraries and several collections of actual programs and models. The books included with ACL2 have previously been used as a benchmark for the CCG termination analysis, which is integrated in ACL2.

After the preprocessing step, the generated simplified Lisp code is converted to cost relations for CoFloCo, as described in Section 2.5.

3.1.1 Locally defined functions

Books often include other books, which causes function definitions from the included books to appear in the preprocessed output. If those same functions were counted

$$\mathrm{eq}(\texttt{equal}(a_{\mathrm{int}}, 0, 0, b_{\mathrm{int}}, 0, 0 \rightarrow 1, 0, 0), 1, [\,], [a_{\mathrm{int}} = b_{\mathrm{int}}])).$$
$$\mathrm{eq}(\texttt{equal}(a_{\mathrm{int}}, 0, 0, b_{\mathrm{int}}, 0, 0 \rightarrow 0, 0, 0), 1, [\,], [a_{\mathrm{int}} + 1 \leq b_{\mathrm{int}}])$$
$$\mathrm{eq}(\texttt{equal}(a_{\mathrm{int}}, 0, 0, b_{\mathrm{int}}, 0, 0 \rightarrow 0, 0, 0), 1, [\,], [a_{\mathrm{int}} \geq b_{\mathrm{int}} + 1])$$
$$\mathrm{eq}(\texttt{equal}(a_{\mathrm{int}}, a_{\mathrm{len}}, a_{\mathrm{size}}, b_{\mathrm{int}}, b_{\mathrm{len}}, b_{\mathrm{size}} \rightarrow r_{\mathrm{int}}, 0, 0), 1, [\,], [0 \leq r_{\mathrm{int}}, r_{\mathrm{int}} \leq 1])$$

Figure 3.1: Cost relation for Lisp's built-in `equal` function. The first three cost equations are applicable if both arguments are integers. The input arguments a and b of the last cost equation are unconstrained, meaning that it applies in all other cases. In this equation, the result r_{int} is not specified precisely (it is either 0 or 1), corresponding to the unknown result of the comparison.

separately for each file they are included in, they would be counted multiple times. Since our experiments are intended to determine for which percentage of functions within the benchmark our analysis can calculate bounds, this would distort the results. In order to avoid such repetitions, only those functions which are defined "locally" (i.e., directly in the file being analyzed) should be counted.

In the preprocessing step, annotations are added for those functions, and the translation adds entry point declarations for the corresponding cost relations. CoFloCo then treats only cost relations with such declarations as entry points for its analysis, meaning that bounds are reported only for those cost relations.

3.2 Issues and solutions

3.2.1 Program complexity

A general problem is that transformed programs can get very complex. Typically, this is due to a combinatorial explosion of possible execution paths, especially if there is a large number of conditional branches, or many cost equations for a cost relation.

An example of a complex cost relation is `equal`, which performs a deep equality comparison on two values of arbitrary type. This means that, within our analysis, it is possible to reliably determine equality only if the integer values are the same (since the measures cannot express the contents of a structure). Non-equality is guaranteed if either the integer values or the length/size measures are different. Hence, there are four different cases (i.e., cost equations), as shown in Fig. 3.1.

Therefore slicing (see Section 3.2.1) – and more generally simplification – is important for reducing the computational complexity of the analysis. Some simplifications are already performed as part of the translation to cost relations. However, simplification is also performed directly in CoFloCo, which prunes paths which are made infeasible by earlier constraints. Generally, while improvements to CoFloCo have resolved many problematic cases, the complexity is often inherent and not easily (or at all) reduced. Such input cases can result in the creation of large numbers of cost equations (thousands) and intermediate variables (tens of thousands), which severely impact the running time of the analysis (see Chapter 4 for details on the analysis process).

Functions containing nested conditional blocks with type checks are a frequently occurring example of high path complexity. The large number of branches is due to the lack of strong typing in Lisp: for functions where behavior depends on the type of the argument(s), such as `equal`, types must be checked explicitly. In fact, in the context of the analysis, it is only known whether length/size is 0 or greater than 0, i.e., whether the argument is an atom or a list.

A significant contributor to complexity is the inclusion of unneeded measure variables in cost equations. The approach used for our analysis depends on "guessing" one or more measures (here, just a constant set) and adding corresponding variables to cost equations, which are then passed to CoFloCo for analysis. Generally speaking, the more variables need to be processed, the higher the complexity of the analysis task, as the size of the constraint system increases accordingly. Slicing can remedy this effect to some degree, as described in Section 3.2.1. Another technique, based on the dynamic addition of required measures according to inferred type information, is proposed in Section 7.1.1.

Slicing

As described in the previous section, the effort required for calculating bounds increases with the number of variables, many of which may not actually be helpful for calculating a bound. In order to reduce the impact of this issue, simple slicing (variable elimination) is performed during the translation. Unused and undefined variables are eliminated by propagating information on "important" variables (i.e., those which are actually used in the cost relation) and then removing all others. This method is very simple and does not add significant computational effort. Furthermore, slicing does not affect the soundness or precision of the final cost results (bounds), as it only removes variables which are unused/undefined, and thus reduces the size of the problem CoFloCo has to handle. More advanced constant propagation might further add to the advantages gained by slicing, but this has not been investigated yet.

3.2.2 Complexity of bounds

In practical usage, issues remain even where bounds are successfully calculated. The bound expressions calculated by CoFloCo are sometimes very complicated, especially for functions with many indirect function calls. Such expressions are far too large to be easily comprehended, while the degree of precision they represent is also unlikely to be meaningful for use cases where a human needs to interpret the bound. On the other hand however, while the asymptotic complexity is reported separately (and is inherently very simple), it might be insufficiently precise for making useful comparisons. While CoFloCo does perform some arithmetic simplifications on the bounds, this would have to be extended to produce truly human-readable bounds (see Section 7.2 for additional discussion).

3.2.3 Distinction between input and output variables

As mentioned in the definition of resource bounds in Chapter 1, bounds should be expressed over the arguments of a function only – a bound expression which is dependent on the function's return value would not be useful, as this value cannot be easily determined in general.

However, the strict distinction between arguments and return values which exists in most programming languages (including Lisp) is not inherent to cost relations, in which functions are represented as predicates (the notation used in this work, where inputs and outputs are separated by an arrow, is merely a visual representation intended to assist the reader). It is therefore necessary to explicitly declare which variables in a cost equation are inputs and outputs respectively, using a separate declaration (`input_output_vars`). For Lisp, this is straightforward, since functions have only one output value (the return value) with its three corresponding measure variables. In the presence of such a declaration, CoFloCo will report *infinity* if it cannot determine a bound which does not include the output arguments.

3.2.4 Explicit determinism

Since side effects are not possible in the simplified input language, all functions are deterministic. However, as CoFloCo is not limited to side-effect-free languages, it cannot assume determinism. This is problematic for cost relations which are not fully deterministic, i.e., whose cost equations are not mutually exclusive (see Section 2.2). In this case, CoFloCo does not conclude that two calls to the same function with the same arguments will return the same result.

Consider a trivial example such as `(if (integerp x) (integerp x) 5)`. Since the cost relation for `integerp` is nondeterministic, the call `(integerp x)` in the condition could return `t`, so that the *true* branch will be evaluated, but then the second call to `integerp` might return `nil`, which is clearly impossible according to the Lisp semantics.

This issue can be remedied to some degree by explicitly enforcing identical behavior of identical calls within cost relations. To do so, the output measures of all calls to the same function and with identical input arguments within one cost equation are unified by adding an explicit equality constraint to the cost equation. However, this does not cover all such cases, as it does not span across different cost relations. This is acceptable however, since this issue does not cause bounds to be unsound, although it may prevent them from being found.

3.2.5 Relations between function calls

A general problem of cost relations is that they cannot directly represent semantic relations between different cost relations. For example, the sum of the size measures for the results of two calls `(car a)` and `(cdr a)` never exceeds that of the common input argument `a`. This fact follows from the behavior of those functions as defined in Lisp/ACL2 in conjunction with the definition of the size measure as given in Fig. 2.2. However, the cost relations contain only the information that the size decreases ($r_{\text{size}} < x_{\text{size}}$), but the relationship between the two cost relations is not represented:

$$\text{eq}(\texttt{car}(x \to r), 1, [\,], [x_{\text{len}} \geq 1, r_{\text{size}} < x_{\text{size}}])$$
$$\text{eq}(\texttt{cdr}(x \to r), 1, [\,], [x_{\text{len}} \geq 1, r_{\text{len}} = x_{\text{len}} - 1, r_{\text{size}} < x_{\text{size}}])$$

CoFloCo is not able to automatically identify this relationship either. Therefore, this information must be supplied externally, in the form of constraints added to the cost relations which contain the function calls.

In order to resolve this common and straightforward instance of the more general issue, a very simple heuristic is applied. It searches for cost relations which contain, for some input argument `x`, one call to `(car x)` and another to `(cdr x)`. In the benchmark, this is a frequently occurring pattern among functions with multiple recursion (see Section 3.2.6), in the form of what is essentially tree iteration, where the termination condition is `(endp x)`. A (synthetic) example of such a function is shown in Listing 4.

Where such a case is found, we add the constraint $y_{\text{size}} + z_{\text{size}} + 1 = x_{\text{size}}$ (where y is the `car` and z the `cdr` of x), which corresponds precisely to the definition of the size measure given in Section 2.3. The corresponding cost measure is

$$\text{eq}(f(x_{\text{size}}, \dots), c, [\texttt{car}(x_{\text{size}} \to y_{\text{size}}), \texttt{cdr}(x_{\text{size}} \to z_{\text{size}}), f(y_{\text{size}}, \dots), f(z_{\text{size}}, \dots)], [\,])$$

to which the constraint $y_{\text{size}}+z_{\text{size}}+1 = x_{\text{size}}$ is added (for the sake of simplicity, variables for the other measures are omitted here).

However, this detection is purely syntactic and thus fails to detect all cases of this `car-cdr` relationship, as well as more complex relations such as in the evens-odds merge sort implementation shown in Listing 7.

3.2.6 Multiple recursion

Such inter-function relations are particularly important for multiple recursion, since in this context, the dependencies between the arguments passed to the recursive functions determine the asymptotic complexity.

Multiple recursion means that more than one recursive call may be made within one invocation of a function. An example of such a function is shown in Listing 4. We define recursion of degree n (or n-ary recursion) as follows: for all possible execution paths within a single invocation of a function f, that same function f is called at most n times in any path. A formal definition of the recursion degree is given in Fig. 3.2.

Following this definition, a multiply recursive function has a degree $n > 1$. This is an important difference to single recursion ($n = 1$), and the reason why extensions to the usual methods are required: it is no longer possible to simply define the bound as the

```
(defun treesum (tr)
  (cond
    ((consp tr)
        (+ (treesum (car tr)) (treesum (cdr tr))))
    ((endp tr) 0)
    (t tr)))
```

Listing 4: A binary recursive algorithm which recurses through the given tree and sums up the (integer) leafs. For each call to `treesum`, two recursive calls are made, where one call receives the left branch (i.e., the `car`), and the other the right one (i.e., the `cdr`).

$$\begin{aligned}
rd(f, var) &= 0 \\
rd(f, literal) &= 0 \\
rd(f, (fn\ \ \vec{args})) &= \begin{cases} 1 & \text{if } fn = f \\ 0 & \text{otherwise} \end{cases} + \sum_{a \in \vec{args}} rd(f, a) \\
rd(f, \texttt{(if}\ \ c\ \ a\ \ b\texttt{)}) &= rd(f, c) + \max(rd(f, a), rd(f, b)) \\
rd(f, \texttt{(let}\ \ \vec{defs}\ \ body\texttt{)}) &= rd(f, body) + \sum_{\texttt{(}dn\ \ de\texttt{)} \in \vec{defs}} rd(f, de)
\end{aligned}$$

Figure 3.2: Algorithm for determining the recursion degree $rd(f, b)$ of a given body b of (simplified) Lisp code with respect to a function f.

number of times a measure can decrease/increase along a single sequential control flow, as is done in difference constraint-based systems such as Loopus [SZV15]. Instead, it is necessary to consider that a call may subsequently branch into multiple recursive calls.

In order to calculate bounds for multiply recursive functions, it is necessary to analyze the relations between the arguments of the recursive functions. The detection of `car` and `cdr` relationship can enable this in some cases. For example, it enables analysis of quicksort with a single partition function (Listing 5), although the result is a bound over the size, not the length[1].

If information on the relations between function calls is explicitly stated in the cost relations, CoFloCo can use this information for calculating the bound. If the arguments which control the recursion depth are non-overlapping – more specifically, if the sum of their lengths/sizes is at most that of the corresponding argument passed into the caller – the complexity is polynomial, assuming that each recursive call has itself only polynomial cost. This follows directly from the master theorem (see, e.g., [Cor+01]). In Listing 4, `treesum` fulfills this property: the `car` and cdr do not overlap (the results are separate parts of the input structure), and a bound which is linear in terms of the

[1]While the two are equal for lists of integers (which is perhaps the most frequent use of sorting functions), this is not true in the general case, meaning that the bound may be imprecise.

```
(defun int-partition (p x)
  (if (endp x)
    nil
    (let ((parts (int-partition p (cdr x))))
      (if (>= (car x) p)
        (cons (car parts) (cons (car x) (cdr parts)))
        (cons (cons (car x) (car parts)) (cdr parts))))))

(defun int-qsort-partition (x)
  (cond ((endp x)
         nil)
        ((endp (cdr x))
         (list (car x)))
        (t (let ((parts (int-partition (car x) (cdr x))))
             (append (int-qsort-partition (car parts))
                     (cons (car x)
                           (int-qsort-partition (cdr parts))))))))
```

Listing 5: A quicksort implementation using a single call to a partition function which splits the input list into values less than respectively greater or equal than the pivot element. With added constraints connecting the results of `car` and `cdr`, as described in Section 3.2.5, a bound over the size of the input `x` (i.e., x_{size}) can be determined for `int-qsort-partition`.

size of the tree can be obtained. This information is stated explicitly as a constraint, as described in Section 3.2.5 above.

```
(defun treesum (tr)
  (cond
    ((consp tr)
        (+ (treesum (cdr tr)) (treesum (cdr tr))))
    ((endp tr) 0)
    (t tr)))
```

Listing 6: A definition which is almost identical to the one in Listing 4, but where both recursive calls receive the `cdr` of the input as the argument.

Listing 6 shows an example where this is *not* the case: compared to `treesum` in Listing 4, the minor change of passing `(cdr x)` to *both* recursive calls completely changes the asymptotic cost of the function. Clearly, the sum of the arguments of the recursive calls is potentially greater than the input argument – indeed, in the worst case, the sum may almost double in each recursive call. This is easy to see when considering a list input, i.e., a degenerate tree having only right branches: for each element of the list, two recursive function calls occur, each of which receives the rest of the list as an argument. Rather than being linear in terms of the size, the cost is exponential.

Proving non-increasing sum of arguments

However, as mentioned in Section 3.2.5, CoFloCo cannot always determine such relations between multiple calls. While the simple `car`/`cdr` heuristic is enough for many multiply recursive functions, there are many other cases where it is not sufficient (although it may still be required). Divide-and-conquer algorithms where the division step is more involved (such as merge sort) are one instance of this issue. Listing 7 shows a specific example of an integer-only adaptation of a merge sort function[2] contained in the benchmark suite. In such cases, the hardest task is not necessarily to analyze the behavior of the divide step (i.e., the recursive calls) but rather the conquer (here “merge”) step, whose cost depends on the recursive calls’ *return values*. Nonetheless, explicit relations between the recursive arguments can also aid CoFloCo in analyzing the return values.

```
(defun int-merge2 (x y)
  (if (endp x)
      y
    (if (endp y)
        x
      (if (< (car x) (car y))
          (cons (car x)
                (int-merge2 (cdr x) y))
        (cons (car y)
              (int-merge2 x (cdr y)))))))

(defun int-msort (x)
  (if (endp x)
    nil
    (if (endp (cdr x))
      (list (car x))
      (int-merge2 (int-msort (evens x))
                  (int-msort (odds x))))))
```

Listing 7: Modified integer-only merge sort function. `int-merge2`, which merges two sorted lists such that the result is sorted, has linear cost in terms of the lengths of the two input lists. The bound calculated for `int-msort` is quadratic in terms of the list length x_{len} (note that bounds such as $x_{\text{len}} \log x_{\text{len}}$ cannot be calculated).

For such multiple recursion patterns, we use a different and more methodical approach. Here, we attempt to prove that the sum of the measures for the arguments passed to

[2]`msort` in `sorting/msort.lisp`

$$
\begin{aligned}
&\mathrm{eq}(\texttt{if2}(x_{\text{len}} \to r_{\text{len}}), 1,\\
&\quad [\texttt{cdr}(x_{\text{len}} \to xs_{\text{len}}), \texttt{endp}(xs_{\text{len}} \to c_{\text{int}}),\\
&\quad\ \texttt{evens}(x_{\text{len}} \to ev_{\text{len}}), \texttt{int-msort}(ev_{\text{len}} \to evs_{\text{len}}),\\
&\quad\ \texttt{odds}(x_{\text{len}} \to od_{\text{len}}), \texttt{int-msort}(od_{\text{len}} \to ods_{\text{len}}),\\
&\quad\ \texttt{int-merge2}(evs_{\text{len}}, ods_{\text{len}} \to r_{\text{len}})],\\
&\quad [x_{\text{len}} \geq 0, r_{\text{len}} \geq 0, ce_{\text{int}} = 0, c_{\text{int}} = 0])\\
&\mathrm{eq}(\texttt{if1}(x_{\text{len}} \to r_{\text{len}}), 1, [\texttt{endp}(x_{\text{len}} \to c_{\text{int}}), \texttt{if2}(x_{\text{len}} \to r_{\text{len}})], [c_{\text{int}} = 0])\\
&\mathrm{eq}(\texttt{int-msort}(x_{\text{len}} \to r_{\text{len}}), 1, [\texttt{if1}(x_{\text{len}} \to r_{\text{len}})], [\,])
\end{aligned}
$$

Figure 3.3: Simplified cost relation for the path of `int-msort` in which the two recursive calls occur. Irrelevant variables, as well as the non-recursive cost equations, have been omitted.

the recursive calls is less than or equal to the corresponding input measure. If this is the case, a constraint stating this fact is added to the cost equation containing the recursive calls. With the aid of this constraint, it becomes possible for CoFloCo to derive a bound without having to prove this property during the bound analysis (assuming, of course, that such a bound exists and that its calculation succeeds). This proof step must be performed for each argument (which is matched by position) and for each measure. Note that this approach works even if the arguments of the recursive calls are derived from the input arguments in a complex semantic rather than purely structural way, as long as the condition is met. Of course, this method is also not limited to binary recursion, and can in principle handle any degree of recursion without modifications.

Continuing with the example, the cost relation for the most interesting part of the `int-msort` function – the branch where the two recursive calls occur – is given in Fig. 3.3. In order to produce a bound depending only on the length of `x`, it is necessary to show that the combined length of the recursive call arguments (i.e., the number of even and odd elements) is not greater than the input argument. This is expressed by the constraint $ev_{\text{len}} + od_{\text{len}} \leq x_{\text{len}}$, the validity of which must be proved in order to ensure that the resulting bound will still be sound. Conditions which must be met in order to reach the recursive call sites are added as preconditions to the proof. This is important because such conditions carry important information on the context in which the recursive functions are called, and may be necessary for the proof to succeed.

For proving that the sum of the measures in the branches in non-increasing, we use the ACL2 theorem prover. Since ACL2 is intended for (semi-)automatically proving properties of Lisp programs, it is equipped with mechanisms to perform complex proofs involving induction [KM97] and can apply various proof heuristics without user interaction. ACL2 is started once for each input file, and the definitions of all functions in that file are added (with termination proofs disabled, as they are not needed here). Failures, e.g., due to function redefinitions, are silently ignored. Where multiply recursive functions are encountered during the translation to cost relations, proof obligations are constructed and submitted to the prover, which returns whether or not the proof succeeded.

One drawback of this method is that due to relying on ACL2, it is limited to one specific input language (Lisp). If the proofs were instead attempted directly on already generated cost relations, this limitation would be removed, and the method would be compatible with CoFloCo frontends for any language, without further modifications. Since the cost relations are self-contained logical specifications, the proof obligations could in principle be verified using general-purpose automated theorem provers. However, this would require support for recursive definitions and automatic inductive reasoning, for which the current state of the art is not yet sufficient. Furthermore, the cost relations do not specify the complete semantics of Lisp (see, e.g., explicit determinism in Section 3.2.4 and the relation between `car` and `cdr` in Section 3.2.5). While the abstraction is sound, it may not be precise enough in many instances.

For `int-msort` (Listing 7), the following Lisp code is generated as a proof obligation for ACL2 (note that the conditions are inverted, as the recursive calls are located in the *false* branches of the two `if` blocks):

```
(implies (and (not (endp x))
              (not (endp (cdr x))))
         (<= (+ (len (evens x)) (len (odds x)))
             (len x)))
```

If the proof succeeds, the constraint $ev_{\text{len}} + od_{\text{len}} \leq x_{\text{len}}$ is added to the cost equation, thus explicitly stating the property which was just proven. Otherwise, no modification is made, and the translation proceeds as normal. An important point for this heuristic is that both positive and negative results should be obtained very quickly – as part of the translation to cost relations, it is intended as an optimistic and speculative attempt, but not a critical part of the analysis. Therefore, failure to prove a constraint, as well as any kind of error, is again simply ignored.

With the added constraints, CoFloCo can calculate a bound for the `int-msort` example. The bound has quadratic complexity in terms of x_{len}, which is of course not very precise – the actual complexity of merge sort being $O(n \log n)$ – but the best result which can be obtained within the context of our analysis, which does not support logarithmic bounds. Another important point for accuracy is that the bound does not depend on any other measure, specifically not on x_{size}.

It is important to note that this analysis currently does not work across different cost relations, e.g., where one recursive call is located within an `if` branch and another outside of it. However, this is merely for technical reasons and should not pose any conceptual difficulties.

As currently implemented, this method also cannot handle cases where the bound or termination for such a multiply recursive function depends on more than one argument (e.g., on the difference between two arguments), as proofs are generated only for single arguments. However, this is a decision intended to reduce the number of proofs rather than a conceptual limitation, and it is doubtful that such cases occur in significant numbers within the benchmark.

3.3 Remaining discrepancies and inaccuracies

Certain issues arise due to the way ACL2 implements specific functions and macros. Since ACL2 is a theorem prover, its goal is to prove or disprove logic statements expressed in Lisp code, whereas the cost of executing this code is generally irrelevant. Therefore, it might be advantageous to generate more or different code than a real Lisp implementation would have to run, in order to simplify proofs. On the other hand, our tool is quite the opposite, and as a result, the calculated asymptotic cost might be different from the asymptotic execution cost in a typical list environment. To work around these discrepancies, we generally retained the definitions from ACL2, but used modified definitions/semantics where the difference was too large.

`or` The `or` macro[3] is a notable example: `or` is short-circuiting in Common Lisp, returning the first argument which is different from `nil`, and preventing execution of any further ones. ACL2 expands `(or x y)` to `(if x x y)`, such that `x` is executed twice if its result is true. This leads to issues especially if `x` is a recursive function call, meaning that the two calls will cause the function to be multiply recursive. However, this conforms neither to the intent of the programmer, nor to how a usual Common Lisp environment would execute this code, and therefore results in a (possibly asymptotic) overestimation of the actual cost.

We initially attempted to modify ACL2's implementation of the `or` macro, but this proved technically problematic, so instead we modified the translation routine: where an `if` branch contains a call to the same function and with the same arguments as the condition, the code is only counted once, and the result is reused.

A somewhat similar problem occurs with `return-last`[4], which returns the value of its last argument. The other argument may have side effects (such as timing an evaluation or printing output) or contain faster code for execution rather than theorem proving. For example, consider the following call, where `fast-union` uses an asymptotically faster algorithm than `union` (the flag `'mbe1` indicates that the second argument should be executed, the last one used for proving):

[3]`http://www.cs.utexas.edu/users/moore/acl2/v7-0/manual/index.html?topic=COMMON-LISP____OR`

[4]`http://www.cs.utexas.edu/users/moore/acl2/v7-0/manual/index.html?topic=ACL2____RETURN-LAST`

```
(return-last 'mbe1 (fast-union x y) (union x y))
```

We chose to modify uses of `return-last`, such that only the last argument is considered for the analysis. Given that our cost model and analysis do not permit side effects, and that we generally focus on the (easier to analyze) logic mode, this is advantageous. On the other hand, counting the cost of both arguments of `return-last` would distort the result, such as by introducing unintended multiple recursion in some instances.

Basic functions

Basic functions are fundamental for describing the semantics of Lisp functions, and their specification has a significant impact on the performance and even correctness of the bound calculation. For our analysis, we assigned a constant cost of 1 execution step to all basic functions. However, it is important to note that the actual cost may depend on the implementation – for at least some of these functions, an execution in a real Lisp environment may not actually take a constant number of steps. As an example, the `length` function is defined as a basic function with cost 1 in our definitions, but according to ACL2's definition of `length`[5,6], the cost is linear in terms of the length of the argument.

It is also important to note that the definition of the "execution step" metric is not necessarily proportional to execution time. For example, arithmetic operations are considered to have constant cost, even though ACL2 supports arbitrary-length integers, meaning that these operations take non-constant time. While it would be possible to represent this as well (e.g., for an addition `(+ a b)`, the cost could be $\max(\log_2 a, \log_2 b)$), our implementation cannot handle logarithms directly, and the resulting bounds would be hard to understand. In the interest of simplicity and clarity, we therefore chose the execution step metric, which defines the basic arithmetic operations as taking only a single step.

Further issues arise due to functions which cannot be properly modeled outside of a Lisp environment (e.g., internals like `gc$`, which invokes the garbage collector). We leave the behavior of such functions undefined and assume that they do not occur in the input.

Error states

Error states (both from explicitly triggered errors such as calls to the (pseudo-) function `hard-error`[7], and function calls with invalid arguments, such as calling `cdr` on an integer) are ignored. It is assumed that they do not occur in a well-defined function call (in ACL2, any such errors cause the execution to abort immediately). Basic function

[5] `http://www.cs.utexas.edu/users/moore/acl2/v7-0/manual/index.html?topic=COMMON-LISP____LENGTH`

[6] The issue here is that the ACL2 `length` function performs an additional check for strings, which cannot be correctly modeled by our analysis.

[7] `http://www.cs.utexas.edu/users/moore/acl2/v7-0/manual/index.html?topic=ACL2____HARD-ERROR`

constraints are defined such that they are undefined for invalid inputs, which means that those inputs are not possible within the constraint system.

Assuming that an error does not occur should result in an overestimate of the resource cost, and therefore in a sound bound. Since execution aborts when an error occurs, less code is executed than otherwise, meaning that the cost according to the execution step metric is lower.

On the other hand, determining the precise cost of a program run which aborts with an error would be quite complex. While such a program has clearly consumed a certain amount of resources until that point, there are difficulties in determining the total cost. For one, possible cleanup costs would have to be considered (although these costs could also be considered internal to the runtime environment, and not part of the code being analyzed). More importantly however, the exit-on-error behavior is different from the usual side-effect-free eager execution (since it skips part of the program), and would therefore require special handling. Therefore, we do not attempt to calculate this cost either.

Interpretation of boolean values

As mentioned previously in Section 2.1, for the purpose of boolean checks (such as in `if` conditions and the `not` function), Lisp considers any value other than `nil` to be true [McC65]. However, expressing this in cost relations is problematic due to the involvement of types. Since we use the same measure for integers and symbols and do not have any way of indicating the type of a function argument, we cannot distinguish between them. Specifically, it is not possible to definitively exclude the possibility of a variable being an integer. This can be done for the length and size measures (they are 0 if the value is not a list), but there is no such special value for integers, since the valid range of the integer measure extends over the entire range of the integer used to represent it. Thus, bounds are potentially unsound if the distinction between the integer value of 0 and the symbol `nil` is critical for determining a bound (see Section 2.5.4).

Listing 8 shows a synthetic example of such a function: if the argument `x` is an integer, it counts down to -10. However, CoFloCo will assume that the loop already terminates when `x` reaches 0, since this is indistinguishable from `nil` according the cost relation for `not`:

$$\mathrm{eq}(\texttt{not}(a_{\mathrm{int}} \rightarrow r_{\mathrm{int}}), 1, [\,], [a_{\mathrm{int}} = 0, r_{\mathrm{int}} = 1])$$
$$\mathrm{eq}(\texttt{not}(a_{\mathrm{int}} \rightarrow r_{\mathrm{int}}), 1, [\,], [a_{\mathrm{int}} = 1, r_{\mathrm{int}} = 0])$$

Therefore, the recursion as modeled by the cost relation stops earlier than in the actual program, resulting in an unsound upper bound.

We consider such a case to be unlikely, as there are few reasonable situations where a value which could be either a number of `nil` is used in this way to control a recursive function. The above example was artificially constructed, and we did not find any similar cases in the benchmarks through cursory inspection. Furthermore, a solution for this

```
(defun int-bool-loop (x)
  (+ 1 (cond ((not x) 0)
             ((> x -10) (int-bool-loop (- x 1)))
             (t 0))))
```

Listing 8: A synthetic example where deriving a sound bound would require being able to distinguish between an integer with value zero (which is considered as true in the context of a condition) and `nil` (which is false). As a result, the bound generated by our analysis reflects a maximum of only x recursive calls, whereas in reality, up to $x + 9$ recursive calls may be made.

problem (based on adding the type of each value as an additional measure) is proposed in Section 7.1.4 (Page 79).

CHAPTER 4

CoFloCo

CoFloCo is a bound analysis system due to Flores-Montoya and Hähnle [FH14; Flo16]. It is originally based on COSTA [Alb+07]. CoFloCo solves a set of cost relations (which encode a program) in order to obtain cost bounds.

4.1 Algorithm

This section provides a very brief overview of the process by which CoFloCo calculates bounds from cost relations. For a more in-depth description, see Flores [Flo16], which this description is based on.

```
(defun listlen (x)
  (if (consp x)
    (+ 1 (listlen (cdr x)))
    0))
```

$$\begin{aligned}
&\mathrm{eq}(\texttt{if1}(x_{\mathrm{len}} \to r_{\mathrm{int}}), 1, [\texttt{consp}(x_{\mathrm{len}} \to c_{\mathrm{int}}),\\
&\quad \texttt{cdr}(x_{\mathrm{len}} \to y_{\mathrm{len}}), \texttt{listlen}(y_{\mathrm{len}} \to rs_{\mathrm{int}}),\\
&\quad \texttt{+}(1, rs_{\mathrm{int}} \to r_{\mathrm{int}})], [c_{\mathrm{int}} = 1])\\
&\mathrm{eq}(\texttt{if1}(x_{\mathrm{len}} \to 0), 1, [\texttt{consp}(x_{\mathrm{len}} \to c_{\mathrm{int}})], [c_{\mathrm{int}} = 0])\\
&\mathrm{eq}(\texttt{listlen}(x_{\mathrm{len}} \to r_{\mathrm{int}}), 1, [\texttt{if1}(x_ell \to r_{\mathrm{int}})], [\,])
\end{aligned}$$

Listing 9: The `listlen` function which was previously discussed as an example in Chapter 3, together with its cost relations.

The algorithm starts with a set of cost relations, shown here for the example function `listlen` (Listing 9). First, the cost relations are preprocessed, such that the cost relation for `if1` is merged into that for `listlen`, resulting in the two cost equations ln_1 and ln_2, corresponding to the two branches (in the following, `listlen` is abbreviated as `ln`):

$$\begin{aligned}
ln_1 :\ &\mathrm{eq}(\texttt{ln}(x_{\mathrm{len}} \to r_{\mathrm{int}}), 2, [\texttt{consp}(x_{\mathrm{len}} \to c_{\mathrm{int}}), \texttt{cdr}(x_{\mathrm{len}} \to y_{\mathrm{len}}), \texttt{ln}(y_{\mathrm{len}} \to rs_{\mathrm{int}}),\\
&\quad \texttt{+}(1, rs_{\mathrm{int}} \to r_{\mathrm{int}})], [c_{\mathrm{int}} = 1])\\
ln_2 :\ &\mathrm{eq}(\texttt{ln}(x_{\mathrm{len}} \to r_{\mathrm{int}}), 2, [\texttt{consp}(x_{\mathrm{len}} \to c_{\mathrm{int}})], [c_{\mathrm{int}} = 0])
\end{aligned}$$

Next, a refinement step is performed, which orders cost equations into so-called *chains*. A chain consists of a sequence of *phases* representing a possible execution of a cost relation. A phase, in turn, consists of either "one or more recursive [cost equations] executed a positive number of times" ($(ce_1 \vee \ldots \vee ce_n)^+$ for a sequence of cost equations ce_i), or "a single (non-recursive) CE executed once" [Flo16] (ce). For example, $[ln_1^+ ln_2]$ is the chain corresponding to an invocation of `listlen` with a non-empty list, where the recursive branch is executed at least once (represented by the phase ln_1^+), and the non-recursive branch is executed once (phase ln_2) at the end.

During the refinement, infeasible chains (such as non-terminating chains which are known to be impossible) are removed. For example, the chain $[ln_1^+]$ is pruned, since starting with ln_1^+ provably results in the terminating chain $[ln_1^+ ln_2]$.

The remaining chains for `listlen` are thus:

- $[ln_1^+ ln_2]$: One or more iterations of ln_1, followed by a final execution of ln_2. This corresponds to a call to `listlen` with a non-empty list.
- $[ln_2]$: A single execution of ln_2, corresponding to applying `listlen` to an empty list.

The fact that no non-terminating chains remain also proves that `listlen` terminates for any input.

In the next step, *cost structures* are generated from the cost equations, chains and phases. A cost structure is a triple $\langle E, IC, FC(\vec{x})\rangle$ consisting of three parts:

- A linear expression E over *intermediate variables*, describing the cost of the cost structure.
- A set of *non-final* constraints IC over the intermediate variables.
- A set of *final* constraints $FC(\vec{x})$ which link the intermediate variables to the variables $\vec{x} = \vec{a} \cup \vec{v}$ of the cost equation(s) (see Section 2.2).

The final constraints $FC(\vec{x})$ are linear, whereas the non-final ones in IC may involve the multiplication of two intermediate variables. Through combinations of cost structures, "we can express complex polynomial expressions" [Flo16]. Cost structures are determined incrementally and compositionally from cost equations, then phases and finally chains.

The cost structure for chain $[ln_1^+ ln_2]$ of `listlen` is constructed in the following steps:

- cost equation ln_2: $\langle 2+1, \{iv_1 = 0\}, \{iv_1 = x_{\text{len}}\}\rangle$ – note that the constraint from `consp` was included. The phase ln_2 is identical.
- cost equation ln_1: $\langle 2+3, \{iv_1 > 0\}, \{iv_1 = x_{\text{len}}\}\rangle$

- phase ln_1^+: $\langle(2+3)*iv_2, \emptyset, \{iv_2 = |x_{\text{len}}|\}\rangle$ – the number of iterations $iv_2 = |x_{\text{len}}|$ is inferred from the constraints $iv_1 > 0$ and $iv_1 = x_{\text{len}}$ for ln_1.
- chain $[ln_1^+ ln_2]$: $\langle(2+3)*iv_2 + (2+1), \emptyset, \{iv_2 = |x_{\text{len}}|\}\rangle$

In the interest of simplicity, the cost of calling the basic functions (such as `consp`) has been included directly in the cost expressions E, as in $2+3$ for ln_1, where 2 is the cost of the cost equation itself and 3 the cost of the calls to `consp`, `cdr` and `+`.

The core idea behind the approach is to represent the bound analysis as an optimization problem over the cost structures. The upper and lower bounds can then be calculated as closed-form symbolic expressions by maximizing respectively minimizing the cost expression, such that the constraints are fulfilled. “This is done by incrementally substituting intermediate variables in E for their upper/lower bounds defined in the constraints until E does not contain any intermediate variable.” [Flo16] For `listlen`, the bound $5*x_{\text{len}}+3$ can thus be calculated.

The derivation of cost structures from cost equations, phases and chains includes a transformation of the constraints to linear form. This step requires the addition of new intermediate variables and constraints, and the application of various strategies such as inductive sum, basic product and min-max. By solving only linear problems, CoFloCo can thus calculate non-linear polynomial bound expressions.

4.2 Bound expressions

The bounds produced by CoFloCo are linear combinations of polynomials. Additionally, they may include certain functions, which provide only basic arithmetic and are not recursive. Currently, only two such functions are used:

max returns the maximum integer value among a list of values – in the context of bounds, this is the maximum cost among a set of possibilities.
nat is the coercion to $\mathbb{N}$, i.e., $\text{nat}(x)$, equivalent to $\max(x, 0)$.

The definition of bound correctness is provided in Section 2.4.

In the following, a few examples of bounds are listed:

- 1 : the constant cost of a basic function, such as `car`.
- $5*x_{\text{len}}+3$: a linear bound for the `listlen` function in Listing 2. As expected, it is linear in terms of the length of the list length. Note that nat is not needed, since additional constraints enforce that the length is ≥ 0.
- $\max([5*x_{\text{len}}, 1, \text{nat}(x_{\text{int}})*6+1])+6$: `type-dependent` in Listing 3, where the maximum expression represents the fact that the type of x is not known.

- $\max([9, \mathrm{nat}(x_{\mathrm{len}} - 1) * 9 + \mathrm{nat}(x_{\mathrm{len}} - 1) * \mathrm{nat}(x_{\mathrm{len}} - 1) * 5 + 9]) + 3$: a quadratic bound for `deg`$_2$`-length`(x) according to Listing 14.

4.3 Limitations

Although the analysis performed by CoFloCo is quite powerful, there are also certain limitations.

Multiplication CoFloCo can obtain only polynomial bounds, meaning that logarithmic or exponential bounds cannot be calculated, at least not directly (Section 7.1.2 proposes a partial solution). Constraints in cost equations must be linear, i.e., of the generalized form $c_1x + c_2 \leq 0$ (where c_1, c_2 are constants and x is a variable). Linear combinations of multiple variables are also permitted, as in $x_1 + c_2 * x_2 - x_3 \leq 0$. Multiplication of two variables and division of the form variable/variable or constant/variable, however, are not supported. It is not possible to calculate a bound (or even prove termination) for functions where such an operation modifies a measure which is relevant for termination. Listing 10 shows a simple function for which our approach cannot prove the otherwise trivially observable termination, since it depends on the result of the multiplication of two variables (or in this case, the multiplication of a variable with itself).

```
(defun mult-test (x)
  (if (and (> x 1) (< x 100))
    (mult-test (* x x))
  x))
```

Listing 10: A function where termination depends on the result of a multiplication of two variables.

Multiplication of a variable by a constant, however, is fully supported, even where the bound value depends on the multiplication's result. For an example of such a function, see Listing 11. However, there is a minor complication due to the fact that multiplication is represented as a call to the basic function `*`. The cost relation of this function is defined as

$$\mathrm{eq}(\texttt{*}(a_{\mathrm{int}}, b_{\mathrm{int}}, c_{\mathrm{int}}), 1, [\,], [c_{\mathrm{int}} = a_{\mathrm{int}} * b_{\mathrm{int}}])$$

It contains the constraint specifying the actual arithmetic multiplication ($c_{\mathrm{int}} = a_{\mathrm{int}} * b_{\mathrm{int}}$), meaning that in a call `*`$(x_{\mathrm{int}}, 2, y_{\mathrm{int}})$, the integer measure of the constant 2 would still be represented by a variable in the cost equation for `*`. Such cases are detected during the translation to cost equations, and in addition to the call to the multiplication function, the corresponding multiplication is added directly to the caller's cost equation as a constraint $y_{\mathrm{int}} = x_{\mathrm{int}} * 2$. Since this multiplication now involves only one variable, CoFloCo can determine a bound for the function.

It would also be sufficient, and in fact more versatile, to perform a complete constraint propagation on the cost equations. However, this would add additional complexity (since cost equations would have to be duplicated to include the propagated constants), and was therefore postponed for possible later work.

```
(defun exp2-limit (x)
  (if (and (> x 0) (< x 500))
    (+ 1 (exp2-limit (* x 2)))
    1))
```

Listing 11: Termination depends on the result of a multiplication of a variable and a constant. The bound $\max([1, \operatorname{nat}(-x_{\text{int}} + 500) * 7 + 1]) + 4$ is successfully calculated. However, this bound is asymptotically imprecise. The actual cost is logarithmic in terms of x_{int}, but logarithmic bounds cannot be calculated by CoFloCo.

While it is possible to calculate bounds for certain functions involving multiplication, these bounds are always polynomial. Whereas the precise bound for a recursion depending on linear multiplication (i.e., multiplication of a constant and a variable) should be logarithmic, CoFloCo will calculate a linear bound, if any. For example, the calculated bound for `exp2-limit` in Listing 11 is linear in terms of x_{int}, whereas the actual cost is logarithmic.

Certain other functions, such as the modulus (remainder) also cannot be precisely defined due to the lack of support for non-constant multiplication, division etc. in cost equations. More complex operations such as bit-shift are also unsupported.

It is worth noting that integers in Lisp, cost relations and CoFloCo are all mathematical integers, meaning that they are not limited in range. Therefore, integer overflows are not a concern.

Precision-performance trade-off In general, resource analysis involves a tradeoff between performance/running time on one side and high precision/completeness on the other. Both aspects are important – not only do long computation times reduce the feasibility of an analysis, especially for interactive use, but they also imply that fewer bounds can be calculated within a given time span. CoFloCo aims for high precision, accepting slower calculating speeds and potential non-termination as tradeoffs.

4.4 Mutual recursion

Mutual recursion is recursion involving multiple recursive functions. Rather than only a single function calling itself recursively, the recursion proceeds through a cycle of at least two mutually recursive functions, where each calls the other. A (somewhat artificial) example of mutual recursion involving two functions `even` and `odd` is shown in Listing 12.

```
(defun even (x)
  (if (= x 0)
    t
    (odd (- x 1))))
```

```
(defun odd (x)
  (if (= x 0)
    nil
    (even (- x 1))))
```

Listing 12: Two mutually recursive functions which determine whether a natural number is even or odd respectively by successively calling each other until zero is reached. This is intended merely as an illustrative example – clearly, there are much better ways of implementing this functionality in practice.

In this case, both of the functions are possible entry points into the mutual recursion cycle.

In CoFloCo, mutual recursion is analyzed as proposed by Albert et al. [Alb+11]. For each of the functions involved in a mutual recursion, the other function calls for one cycle are embedded or inlined into it by means of partial evaluation. Thus, the mutual recursion is converted to several simple recursive functions, each of which again represents an entry point. The result is equivalent to the original mutual recursion both functionally and with respect to cost. The sole limitation of this technique is that the combination of mutual and multiple recursion (i.e., several mutually recursive calls within one function) cannot be processed.

4.5 Amortized analysis

In many cases, there is a relation between different loops in the program (or recursive functions when analyzing functional programs), such that the cost of an inner recursion is amortized (averaged or distributed) over the iterations of the outer recursion. As a result, the combined maximum cost is far lower than the product of the individual cost maxima. An example of such a function is given in Listing 13, adapted from a very similar example by Tarjan [Tar85] In `amt`, depending on the values in the list `m`, either a new value is pushed to the stack `st`, or all values are removed. When called from `amortized`, such that `st` is initially empty, the total number of iterations in all calls to `popall` is at most equal to the length of `m`.

In such a case, merely multiplying the worst-case costs or iteration counts of the recursive functions would lead to a sound but (possibly asymptotically) inaccurate bound. A sound upper bound for `amortized` in the example would be the multiplication of the maximum cost of `popall` (which is proportional to the length of `st`) by the number of times `amt` is called. However, this would clearly be highly inaccurate. In the example, the asymptotic cost of `amortized` would be $O({m_{\text{len}}}^2)$ when calculated using simple multiplication of the maximum costs, whereas in reality, it is only $O(2 * m_{\text{len}})$.

Amortized analysis methods, as described by Tarjan [Tar85], handle such cases more precisely, since they respect the aforementioned relations. CoFloCo performs amortized

```
(defun popall (x)
  (if (endp x)
    nil
    (popall (cdr x))))

(defun amt (m st)
  (cond
    ((endp m) st)
    ((car m) (amt (cdr m) (popall st)))
    (t (amt (cdr m) (cons 42 st)))))

(defun amortized (m)
  (amt m nil))
```

Listing 13: A function with amortized complexity (example adapted from Tarjan [Tar85]).

bound analysis, and obtains a bound for `amortized` which is linear in terms of the length of `m`. It is important to note that amortized analysis is distinct from average-case analysis – in the former, the amortized bound is still a valid upper bound for any combination of inputs. The only disadvantage of amortized analysis is that it is more complex than non-amortized analysis, which merely requires obtaining the maximum cost of each loop/recursion individually.

CHAPTER 5

Experiments

We performed several experiments in order to evaluate both the approach and the implementation of our analysis. Notably, there are two interesting and relevant properties: the quality of results (how many bounds can be obtained and how precise they are) and the time which is spent on calculating them. Furthermore, the experiments were intended to reveal interesting cases which our analysis cannot solve yet, and to help in identifying patterns in programs which cause such failures.

5.1 Experiment setup

The experiments were run on a machine with an Intel Pentium E6700 (2 cores with one thread each, 3.2 GHz) and 14 GiB of RAM, running Debian 7.11. The CoFloCo version used was commit `f74635c`, running on SWI-Prolog 6.6.6 with PPL 1.2. ACL2 version 7.1 running on CCL 1.11 was used as an external theorem prover.

For the main analysis, a time limit of 300 seconds[1] and memory limit of 4 GiB[2,3] per input file were used (note that one file may include multiple entry points, which are all processed in the same invocation of CoFloCo). If either value exceeds the respective limit, the process is terminated immediately. However, results which have been obtained before the time or memory limit was reached are counted as successful results.

[1]Cumulative running time (utime) of the process and all its subprocesses. This excludes external factors such as file access delays, which are not relevant for the analysis.

[2]Measured as the sum of the resident set sizes (RSS) reported for the child processes.

[3]Both limits were checked 10 times per second. The time consumed by the checks is not included in the time limit or the statistics.

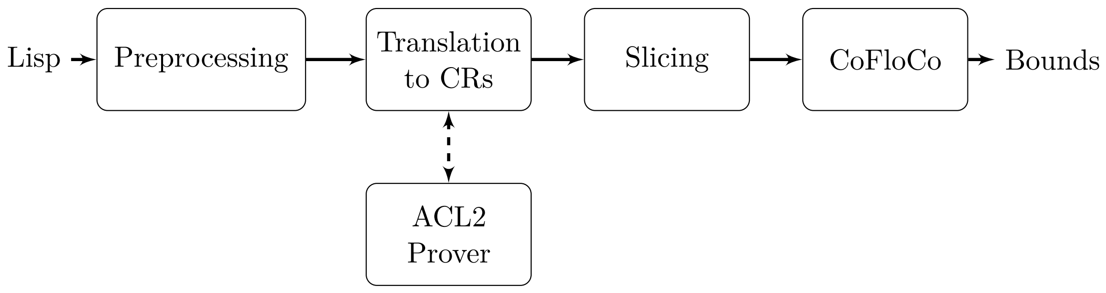

Figure 5.1: The structure of the analysis workflow, starting with the Lisp input and ultimately generating bounds.

Figure 5.1 shows a high-level overview of the process flow of the analysis. Starting with the original Lisp code, a preprocessing step (described in Section 3.1) transforms it to the simplified input language. Next, cost relations are generated (see Chapter 2 and 3), optionally with added constraints for multiply recursive functions, whose applicability is determined using the ACL2 prover. Next, the generated cost relations are simplified by slicing. Finally, CoFloCo calculates bounds from the cost relations.

We analyzed all "books" (Lisp files, see Section 3.1) in ACL2 version 7.1, in total 5,615 files. Of these, 5,147 files containing 20,431 local function definitions were successfully preprocessed. There was no time or memory limit on the preprocessing step, which takes considerable time since, for technical reasons, the CCG termination prover must be run as well (the result is not used). Some parts of the benchmark (468 out of the 5,615 Lisp files) are excluded from the analysis, as they either require special setups to process in ACL2, or take extremely long. Since these exclusions are purely technical and not caused by or otherwise directly related to our work, they are not included in any of the following statistics.

Generating cost relations from the simplified Lisp input took 86 minutes[4]. Thus, the translation step is not a bottleneck, though it takes longer if more elaborate processing, such as proving assumptions (see Section 5.3 for details and results), is performed. Of the successfully preprocessed functions, the translation succeeded for 5,111 files. Notably, only 1,934, or 38% of these files actually contain any entry points (corresponding to function definitions in the original Lisp books, excluding definitions which are included from other books). In total, the translation produced 19,491 unique entry points (i.e., functions for which bounds should be generated). The remainder of this chapter refers only to those files which contain at least one entry point.

We then ran CoFloCo on the resulting cost relations, with debugging output (`-debug`) and statistics (`-stats`) enabled. While this adds some additional computational and I/O load (and thus slightly increases the running time), it provides more detailed information on the solving process. Furthermore, we used the incremental output mode

[4]This value appears to be quite variable, presumably due to the large proportion of time spent on reading Lisp files from and writing cost relations to disk.

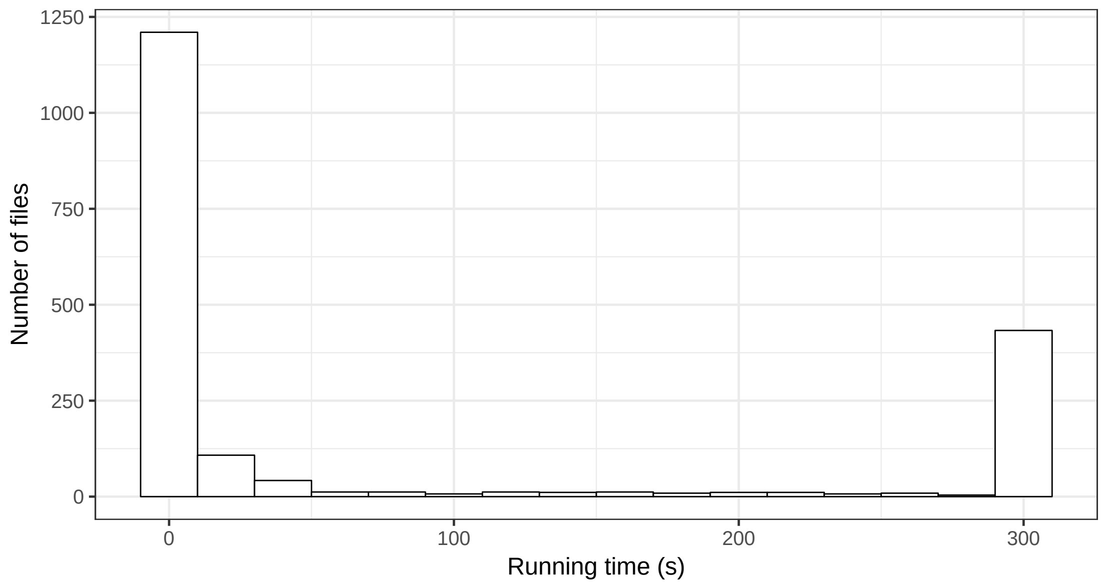

Figure 5.2: Histogram of running times of bound calculations (per file containing at least one entry point), excluding those for which no results could be calculated due to running out of memory (*oom*) or general errors and incompleteness issues (*err*). The rightmost bar (> 300 s) represents timeouts (but note that results may still have been obtained before the timeout).

(`-incremental`), which causes CoFloCo to print each bound as soon as it is calculated. This reduces the distortion of results caused by all functions within a file being analyzed in one step (see Section 5.4 for details). The exact invocation of CoFloCo was

```
    cofloco -i $ces_file -v 3 -assume_sequential -compress_chains 2 \
            -compute_lbs no -stats -debug -incremental
```

The informational and error outputs of CoFloCo are written to files, from which results (including the bounds) are then extracted.

5.2 Results

	const	n	n^2	n^3	n^4	$\sum$ ok	inf	timeout	oom	err	$\sum$ fail	total
#	5096	4081	1074	112	9	10372	2285	5466	771	597	9119	19491
%	26.1	20.9	5.5	0.6	0	53.2	11.7	28	4	3.1	46.8	100

Table 5.1: Number of functions (entry points) per result/complexity group and percentage of total, with sub-totals for succeeded and failed bound computations.

Primarily, the experiment was a success in the sense that an analysis of such a large benchmark was actually performed. Our system successfully calculated bounds for over 10,000 functions from code which was written for real-world use, rather than synthetic examples.

Table 5.1 shows the number of functions for each complexity degree, as well as failures. The first row lists absolute numbers, the second the percentages relative to the total sum. Subtotals for successful bound calculations (i.e., those where a bound was determined) and failed ones are also provided. *inf* counts cases where calculation finished, but no bound could be determined. *timeout* and *oom* denote functions for which the calculation timed out and ran out of RAM, respectively, as defined by the limits given in Section 5.1. *err* denotes certain types of failures related to incompleteness of the cost relation specification (also meaning that no bound can be derived), as well as any failures which do not fall into any of the previous category, such as instances for which CoFloCo returned only an error message. Note again that *timeout*, *oom* and *err* are counted per function – if computation on a file stops or fails at some point, all bounds calculated so far are counted as successful results.

Concerning the running time, the positive aspect is that calculation finished very quickly for a large number of files, as seen in Fig. 5.2. It is important to note that these times are per file – individual functions can often be analyzed in fractions of a second, fast enough to make the analysis suitable for interactive use in principle.

The results also demonstrate the advantages of building on top of an existing bound analysis system (CoFloCo). With a relatively simple frontend for a language which CoFloCo itself was not explicitly built for, it is possible to achieve reasonably good results. Furthermore, it can be concluded that the very simple approach of using three fixed measures is sufficient for analyzing a large number of functions, though it might affect the precision of the bounds.

The results clearly show that among the bounds which our tool can calculate, the majority is either constant or has a very low degree, whereas the number of bounds decreases sharply for cubic and further complexities. It is not clear whether this is due to the complexity of such examples causing the analysis to fail, or due to their rarity in the benchmark. However, based on cursory observations of the benchmark functions, the latter seems likely.

There is no fundamental limit to the degree of bounds our analysis can calculate, and the bound degree by itself does not appear to have any significant effect on the running time of the analysis. We performed an experiment with arbitrarily nested functions as shown in Listing 14. Functions nested 10 levels deep (with a resulting bound of degree 10) do not appear to take significantly longer to analyze than `deg1-length` alone.

Analyzing individual results

A selection of *infinite* results were manually analyzed. This is the most interesting category of results, where calculation terminated within the time limit, but no bound

```
(defun deg1-length (x)
  (if (endp x)
    0
    (+ 1 (deg1-length (cdr x)))))

(defun deg_i-length (x)
  (if (endp x)
    0
    (+ (deg_{i-1}-length (cdr x)) (deg_i-length (cdr x)))))
```

Listing 14: Functions with a given fixed degree of complexity. For each list element, the function with the next-lower degree iterates over the rest of the list, resulting in a cost of degree n for $\texttt{deg}_n\texttt{-length}(x)$.

undefined function	2
non-representable basic function	7
underconstrained basic function	6
insufficient measure expressiveness	5
(unclear)	2

Table 5.2: Likely causes of non-completeness of the analysis (result *inf*, i.e., analysis finished correctly, but without a bound being found) among 20 randomly selected results. There may be more than one cause per sample.

could be found. As opposed to the other failures, this possibly indicates an incomplete aspect of the analysis rather than merely a performance issue. Certain patterns can be discovered, but in many cases, it is still hard to determine the exact cause of the problem.

A set of 20 samples was randomly drawn from the set of functions with *infinite* result and inspected. The samples were grouped by (suspected) cause of the non-completeness into the categories shown in Table 5.2 (note that there may be more than one cause per sample).

As these results clearly show, there is a no single cause for incompleteness. The issue of non-representable basic functions would be hard to address, since these include actions such as (simulated) I/O access, which cannot be usefully represented outside of an actual Lisp environment. “Underconstrained” basic functions are cases where the analysis cannot guarantee that a basic function will return the value which it returns in Lisp, and therefore cannot guarantee termination. This is often related to types, for which a solution is proposed in Section 7.1.4. Another frequent issue is the limited expressiveness of the measures, mainly the inability to describe the (integer) values within a list. This issue together with a possible solution is described in detail in Section 7.1.1.

Unfortunately, this analysis is time-consuming to perform manually, and essentially impossible to automate – we considered automated matching for specific patterns, but this would likely not have been helpful, since the most interesting functions tend to not follow common patterns.

An example of such an interesting function is `msort`[5], which implements a merge sort function which acts on a list of values of arbitrary type, and for which no bound can be calculated due to (among other issues) the fact that the analysis cannot determine the size of the merge step's result (this is addressed in Section 5.3).

Nonetheless, manual investigation is still the most effective way to identify interesting functions. In order to discover the causes of performance issues, functions which caused the analysis to time out would also have to be manually analyzed. As mentioned above, it is also not clear whether functions which would have bounds of a high degree simply do not occur, or whether our implementation fails to calculate them due to exhausting the time or memory limit. Thus, it would be interesting to run the benchmark again, with much more generous time and memory limits, in order to see which new bounds can be discovered. Unfortunately, a full run already takes a very long time (over 42 hours of pure analysis time, excluding overhead), and faster hardware was not available for this work.

5.3 Proofs for multiple recursion relations

In an additional experiment, proof obligations were generated as described in Section 3.2.6 and submitted to an ACL2 prover process running in parallel to the translation step. A time limit of 10 seconds was set for each individual proof. In total, 145,623 proof obligations were submitted to ACL2, of which 23,328 were successfully verified. The proof attempts took approximately 179 minutes in total, with a mean time of 0.121 seconds but a median of only 0.01 seconds. Files for which at least one proof succeeded (meaning that constraints were added to cost equations) were then re-analyzed using CoFloCo.

	const	n	n^2	n^3	n^4	$\sum$ ok	inf	timeout	oom	err	$\sum$ fail
#	30	190	110	14	1	345	469	736	98	344	1647
%	1.5	9.5	5.5	0.7	0.1	17.3	23.5	36.9	4.9	17.3	82.7
#	56	210	99	13	1	379	470	764	98	281	1613
%	2.8	10.5	5	0.7	0.1	19	23.6	38.4	4.9	14.1	81

Table 5.3: Comparison of results for functions which involve at least one multiply recursive function. Results for an analysis without (top) and with (bottom) multiple recursion proofs are shown.

Table 5.3 shows the results with these constraints, compared to the previous run without them, for all functions whose call tree involves at least one multiply recursive function.

[5]contained in `sorting/msort.lisp`

As this comparison shows, the improvements gained by adding constraints to multiply recursive function calls are only marginal. This is likely in part due to the relative rareness of multiply recursive functions which are at the same time simple enough for the ACL2 proofs to succeed, yet were not already covered by the `car/cdr` solution described in Section 3.2.5. In addition, CoFloCo can already handle multiple recursion on its own in many cases. This is evidenced by the fact that even without the additional constraints, bounds can be found for 223 multiply recursive functions (and for 345 functions involving at least one multiply recursive call).

Furthermore, the improvement is in *err* rather than *inf* results, as might be expected. The reason for this is that, as described above, certain sorts of incomplete specification cause the CoFloCo analysis to fail outright. It is therefore hard to tell how many of these results are triggering this issue (and directly benefit from multiple recursion proofs), compared to how many are unrelated cases which are simply located in the same file.

The increase in the number of *timeout* results is also notable. Whereas *inf* and *err* (in this case) denote a failure of the analysis due to incompleteness of the cost relation abstraction, *timeout* results could possibly be solved if more time was available.

However, multiple recursion is often not the only problem, and solving it reveals other limitations of the analysis. For example, it is still not possible to obtain a bound for the `msort` function discussed in Section 5.2. Here, the proof succeeds, establishing that the sum of the lengths of the divided lists is not greater than the length of the input argument. However, in conjunction with the type-dependent merge function, the bound analysis fails (whereas it succeeds for the modified integer-only version of `msort` shown in Listing 7 in Section 3.2.6).

5.4 Limitations

Although the results are quite promising, the evaluation also revealed certain limitations. While several issues were addressed by the author of CoFloCo based on continuous iterations of these evaluations and successive improvements, some of the limitations remain.

Running time The most noticeable problem is that, despite performance improvements to CoFloCo which achieved considerable improvements in results, the analysis still exceeds the time limit for a large percentage of functions (see column *timeout* in Table 5.1). As Fig. 5.2 clearly shows, there is a pronounced division between files which can be analyzed very quickly (< 20 s), and another large number which time out, with few in between. This result is simultaneously encouraging and dissatisfying: on one hand, it means that many results can be obtained within near-interactive time, which would be advantageous if the analysis was integrated into an IDE, for example. On the other hand, the large number of calculations which did not finish even within 6 minutes (which would likely be too long for repeated use while programming anyway) is a significant issue.

In part, this large number of timeouts is a misrepresentation: since all functions in a file are analyzed at once, a single function which takes very long and thus exceeds the time limit causes all following functions within the same file to be counted as timed out as well. Given that timeouts are often caused by just a single particularly complex functions among many others in the same file, a certain distortion of the results is to be expected. Furthermore, since the maximum time per file is always the same, the time available per function decreases as the number of entry points increases. Simply scaling the timeout by the number of entry points contained in the file is possible, but does not resolve the issue of the analysis taking a long time on specific functions.

A separate time budget and corresponding time limit for each function would therefore have been desirable, but this would be very difficult or impossible to implement in the context of CoFloCo's current architecture. On the other hand, while simply running a separate analysis for each entry function would be easy, this would not be a reasonable representation of the overall cost of the analysis: since functions usually call other functions within the same file, these would be repeatedly analyzed.

The sharp division between files which are analyzed very quickly and those for which analysis times out, as observed in Fig. 5.2, is also interesting. The obvious question concerning the computations which timed out is whether a result could eventually be obtained for them, given sufficient time (meaning that there are a large number of cases which take a very long time to analyze), or whether the bound computation is truly stuck. Since the bound calculations in CoFloCo are potentially non-terminating, it is not possible to answer this question in all cases.

Memory usage Besides time, another issue was the analysis running out of RAM on certain instances (see the *oom* column in Table 5.1) despite the generous memory limit. The distribution of the peak memory consumptions (per file) is shown in Fig. 5.3.

However, it is not yet clear for all such cases why the excessive memory consumption occurs. In some cases, very high memory usage might simply be a result of high function complexity, which means that CoFloCo has to analyze a large number of chains (see Section 4.1). Even this, however, is not a satisfactory explanation for the memory consumption behavior observed for certain inputs, where the memory usage simply rises at a very fast and linear rate until the limit is reached. Additional investigation is certainly required here – in fact, it is likely that this behavior is due to a bug in the implementation rather than a more fundamental issue of the analysis. In addition, the above point on timeouts causing all remaining functions to fail applies here as well, meaning that again the actual problematic functions are likely far fewer than those counted in the *oom* column in Table 5.1.

Correctness & precision Notably, the correctness of the results was not verified manually (except for a few functions). Doing so for the entire benchmark would be extremely tedious and effectively impossible.

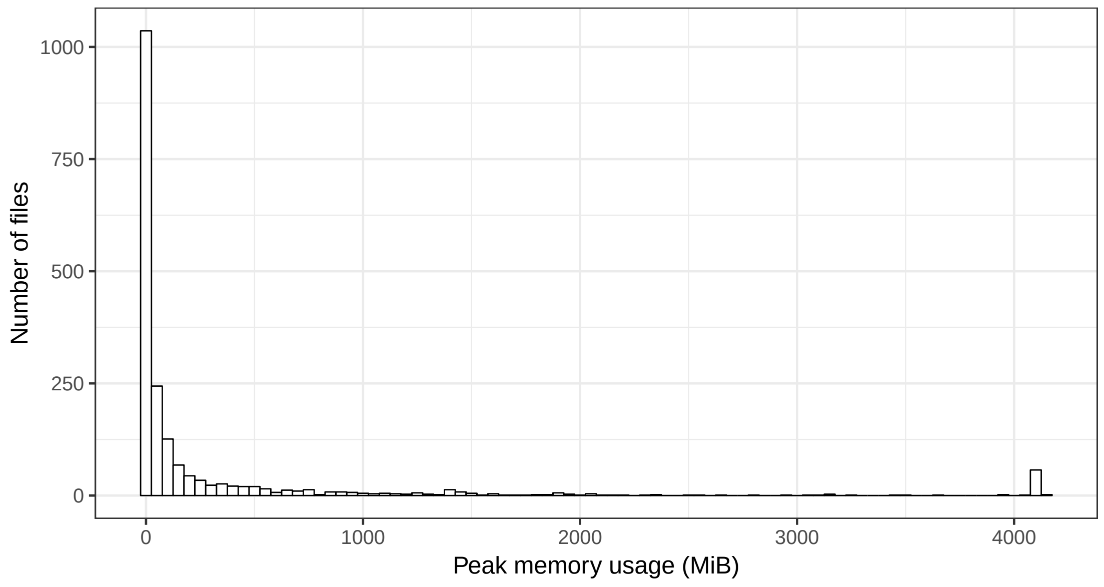

Figure 5.3: Histogram of memory watermarks (i.e., peak memory consumption) per file containing at least one entry point, excluding those where an error occurred (*err*). The rightmost bar (> 4096 MiB) represents out-of-memory cases.

We also did not attempt to determine the precision of the bounds, beyond manually checking whether the choice between length and size was correct for a small selection of bounds. Even asymptotic correctness is impossible to determine automatically, since doing so would essentially amount to solving the resource analysis problem again. Another option would be testing, that is, actually running the code while measuring the number of execution steps. The functions could be executed with sample inputs, with accuracy then being determined by inserting the input values into the bound expression, calculating the resulting numeric value and comparing it to the measured result. Such tests might be useful for measuring the distribution of precision, that is, whether a bound is inaccurate for only a few cases, or more generally. However, this method, like any test-based one, is necessarily incomplete, meaning that corner cases might not be detected. Generating test inputs which cover those corner cases as far as possible, as well as give a good general representation of the bound's precision, would likely be difficult.

Representativity of calculated bounds The distribution of the results appears to be somewhat skewed towards "trivial" bounds (such as constant-cost functions, or ones which consist only of a simple list iteration). However, this is not surprising: for one, the functions themselves are distributed in this way, meaning that highly complex functions are relatively rare. Furthermore, there appears to be a quite sharp distinction between "easy" functions on one hand, and very hard ones (involving complex control flows, associative arrays, etc.) on the other. The latter can often not be analyzed successfully, meaning that bounds for them (which would presumably also be complex) also appear less frequently in the results.

5.4.1 Comparability of results

Making useful comparisons between different bound analysis tools is problematic, as their accepted input formats and supported concepts differ widely (the same has been observed for termination analysis as well [Vro07]). They are either only designed to work on a very limited custom language or on pure integer programs, or specific existing programming languages (e.g., OCaml for RAML). Frequently, only subsets of the respective language are supported, meaning that it may not be possible to analyze "real-world" programs (as is the case with RAML [HDW17], for example).

In fact, we considered automatically converting benchmarks either from Lisp to OCaml or in the other direction, but the paradigms (most notably the type system) are too different to do this effectively, which would have made it an extensive project of its own. Converting from Lisp to OCaml is especially problematic because RAML fundamentally depends on type structure. The other direction is also not straightforward, because static typing does not exist in Lisp, which would not only require a complex conversion of type-based matching, but also put our implementation at a disadvantage, as it cannot rely on strict type rules.

Comparing the accuracy of generated bounds is a further issue. Usually, comparisons focus only on asymptotic cost, which is of course the most important characteristic. However, it would also be interesting to compare the actual bounds produced by different tools, in order to judge the precision of bounds with the same asymptotic cost. Since bounds differ both in format and structure, this is not as straightforward as comparing the asymptotic cost. Also, it would be necessary to compare functions and decide, given arbitrary inputs, which one is more precise. A relevant proposal was made by Albert et al. [Alb+15], which takes advantage of the specific properties of cost bounds.

CHAPTER 6

Discussion of related work

In the following, some closely related systems are discussed in more detail, and compared to our approach in specific aspects. Furthermore, specific tasks (average-cost analysis in Section 6.4, and analyzing other programming paradigms in Section 6.5) which can be analyzed by certain other systems, but not by our current analysis, are highlighted.

The basic concept of this work as well as related ones is to transform a potentially complex program into an abstract structure which is more amenable to analysis. Recurrence relations are one such abstraction, which is used for expressing recursion. Based on recurrence relations, the master theorem (see, e.g., [Cor+01]) provides a method for calculating the (asymptotic) number of times a function is called recursively, expressed in terms of the initial argument. However, this simple method is not sufficient in many cases. Most importantly, multivariate analysis is not possible, meaning that analysis is limited to functions with a single argument. Furthermore, in real-world programs, the hardest problem often lies in identifying such relations in the first place, that is, reducing a complex function, which might itself call other functions, to a suitable recurrence relation. As a result, more sophisticated analysis approaches had to be developed.

6.1 Metric

Wegbreit's Metric [Weg75], a system for resource analysis of simple Lisp programs, was introduced as early as 1975. It uses difference equations to express recursion, solved via "one or more of [...] direct summation, pattern matching, elimination of variables, best-case/worst-case analysis, and differentiation of generating functions" [Weg75]. In order to find the difference equations, recursive functions are executed symbolically, while excluding irrelevant arguments. In fact, this approach in Metric quite similar to that used for COSTA, a much more recent system [Mon+11], which is the predecessor of CoFloCo. Notably, the approach used for Metric also includes probabilistic analysis, where probabilities can be supplied for non-deterministic conditions. These probabilities will be considered in the analysis.

It appears that Wegbreit encountered the same issue in finding suitable abstractions for analyzing programs as we did: “What is not well understood is how to synthesize an abstraction from the program when the correct one is not already known by the system” [Weg75]. This issue is still quite relevant, see also Section 7.1.

While Metric was certainly a pioneering approach, it does not appear to have been compared to other approaches, certainly also owing to its very early appearance, when little other work was available for comparison. Wegbreit [Weg75] hints towards an implementation, but does not give any measurements or an indication that any larger-scale methodical testing was performed.

6.2 RAML

RAML [HS15; HDW17; Hof16] is a resource analysis system for a functional input language also called RAML (“a resource-aware version of the functional programming language OCaml.” [HDW17]). The system reuses the parser and type inference engine of the existing Inria OCaml compiler. RAML’s analysis is based on the structure of types (specifically recursive types), and thus closely dependent on the OCaml type system. In other words, the analysis depends entirely on syntactic properties of the program. Therefore, “RAML fails if the resource usage can only be bounded by a measure that depends on a semantic property of the program or a measure that depends on the difference of the sizes of two data structures” [HDW17].

The key concept of the analysis is the assignment of a *potential* to each data structure by a potential function. The potential must be sufficient to account (or “pay”) for the resources consumed by subsequent operations on this data structure [HS15], such that the total potential is never less than zero. This system also supports amortized analysis, since the potential which is consumed directly represents the cost. The system generates a system of potential requirements (i.e., potential required for operations), specifically a linear constraint system. This constraint system can be efficiently solved using linear programming: the initial potential, i.e., the potential which is required at the entry point of the program, is minimized while fulfilling all constraints. “It then follows that the initial potential function describes an upper bound on the resource consumption of the program.” [HS15] The analysis can handle general recursive types, and therefore works for both standard types (such as lists) and arbitrary user-defined ones.

As opposed to previous potential-based approaches, RAML can also calculate non-linear (polynomial) bounds, using only a linear constraint system [HAH11]. The potential functions are polynomial functions, referred to as *resource polynomials*, from which bounds are derived by solving a linear constraint system over the coefficients. A maximum degree for the resource polynomials must be provided, thus also limiting the maximum degree of bounds which can be determined. If a function has a bound polynomial of higher degree than this limit, RAML will not find a bound. This restriction is necessary because the number of base polynomials within a resource polynomial must be finite: "However, we currently allow the user only to select a maximal degree of the bounds and then track all indices that correspond to polynomials of the same or a smaller degree." [HDW17]. The approach used in CoFloCo, on the other hand, does not require a maximum degree to be specified, since polynomial bounds are assembled "on the fly" based on linear constraints (see Chapter 4).

RAML supports multiple resource metrics: currently evaluation steps (which are quite similar to the metric used in this work), heap space usage and "ticks" (which count how often a counting annotation is reached) can be calculated. RAML can calculate both upper and lower bounds for these metrics.

Furthermore, RAML includes limited support for higher-order functions: "the higher-order case is reduced to the first-order case if the higher-order arguments are available" [HDW17]. On the other hand, unknown higher-order arguments are considered to have no cost [HDW17]. In the latter case, the bound will of course not be sound, but will still reflect the termination behavior and resource consumption of the higher-order function itself. However, due to this lack of complete support for higher-order functions, as well as integers and other constructs such as `if`-conditions, RAML is not yet suitable for analyzing real-world OCaml programs.

Unlike CoFloCo, RAML currently cannot calculate bounds over integers. Instead, it depends on special data structures representing natural numbers [HS15]). This is problematic for analyzing realistic programs, which often combine basic types (such as integers) and complex ones, rather than just artificial examples.

In Section 3.2.5, we described how we generate required assertions for determining whether a multiply recursive function has polynomial complexity. This is done by establishing whether the combined size of the arguments to the recursive calls is within the size of the input. In comparison, RAML avoids this issue entirely by detecting type-based "syntactic" branching in `match` expressions. These expressions perform pattern matching (recognizing specific structural type instances), extract their members and conditionally execute code for the matching entry. Examples of patterns are `x::xs` (the head and tail of a list) or `Tree (a,b)` (an internal node of a binary tree). The code which is executed when a pattern matches then has access to the members of the structure, e.g., `x` and `xs` respectively `a` and `b`. In fact, RAML uses `match` blocks to detect structural program properties in general, including termination conditions, which means it does not have to analyze semantic properties. Since the decomposition implies that the members of the structure are non-overlapping, this method is quite powerful and in fact able to determine bounds for certain multiple recursive functions. Unfortunately, no equivalent to `match` exists in Lisp, therefore data structure accesses and type-dependent behavior at the semantic level, which can be far more complex.

As explained in Section 2.7, our analysis uses maximum expressions to express bounds for functions whose behavior (and thus cost) depends on the type of its argument (which cannot be statically determined). While this specific scenario can of course not occur in programming languages with a strong static type system, such as RAML/OCaml, it is quite similar to the different options of a `match` expression. RAML handles this situation by multiplying the appropriate parts of the bound by a variable which is defined as 1 if the argument matches the respective `match` branch, and 0 otherwise. This approach is actually superior than using a maximum expression, as it is immediately clear how the bound is calculated, rather than merely selecting the larger value. However, such a method is again not suitable for Lisp, as instead of `match`, type-checking conditions consist of regular program code. Unlike the clean structure of `match` conditions, these semantic conditions are more complex to analyze, and cannot be directly used for measures or bounds.

6.3 CCG

The same benchmark used for our experiments has also previously been used to evaluate the implementation of CCG (Calling Context Graph)-based termination analysis (Manolios & Vroon [MV06; Vro07]) which is integrated in ACL2. Termination analysis attempts to prove termination, i.e., determine whether or not a given program halts for all input parameters. Of course, there may again be “unknown” results due to the undecidability of the halting problem. Nonetheless, CCG can prove termination for over 98% of functions in the evaluation presented by Vroon [Vro07]. However, considering that bound analysis is a much harder problem than termination analysis, such success rates cannot be realistically expected for our work. Nonetheless, the CCG approach to termination analysis provides inspiration for resource analysis as well.

6.3.1 Measures

In comparison to this approach of using several measures for one value, CCG termination analysis uses only a single measure, `acl2-count`[1], based on the concept of ordinal numbers [Vro07]. It is applied to integers and `cons` structures as well as rational and complex numbers and strings. For `cons` pairs, this measure is equal to our *size* measure. When proving termination, this is not a disadvantage, since the only relevant property is whether this measure increases or decreases until eventual termination – precision is therefore not a concern. For bound analysis, however, such a measure would be insufficiently precise, as it is necessary to determine *how often* such a decrease or increase happens, and how large it is. A simple function such as `listlen` in Listing 2 demonstrates this. Clearly, only the length of the list is relevant for the bound of `listlen`. However, with a single measure such as `acl2-count`, the bound would be over the number of `cons` pairs in all of x (i.e., identical to our *size* measure). Furthermore, it would not be possible to symbolically express that the bound is the maximum among different choices – whereas `acl2-count` implicitly determines the maximum, our analysis can

[1]`http://www.cs.utexas.edu/users/moore/acl2/v7-0/manual/index.html?topic=ACL2____ACL2-COUNT`

produce bounds containing explicit maximum expressions over different measures, such as $\max(x_{\text{int}}, x_{\text{len}}, 0)$.

6.4 Average-cost analysis

Currently, our analysis returns only the worst-case upper bound (based on the same cost relations, CoFloCo could also perform lower bound analysis, but we did not explore this in detail). While the worst case is important for many applications, such as for determining the maximum stack usage of a program, or deadlines in real-time systems, it is less useful when comparing different algorithms or implementations. Here, the average (or mean/expected) cost is often more meaningful, as the worst case might be unlikely to occur and is not critical, whereas the average case determines important results such as the total running time of a program. A well-known example is quicksort compared to merge sort – the former has quadratic worst-case complexity, whereas the latter has an asymptotically much lower worst-case complexity of $n \log n$. However, the average complexity of quicksort is also $n \log n$. In practice, the choice between quicksort and merge sort depends on other factors, such as the speed of data access.

The average cost depends on the probability of conditions, which control what parts of the program are executed, being met. Listing 15 shows a function which is much simpler than quicksort, but whose cost also depends on the probability of a condition, namely whether list elements are equal to a given input value `x`. Clearly, the probability of this property being fulfilled affects the cost of `search-val`, as the recursion stops as soon as the check succeeds. This probability in turn depends on the input data.

```
(defun search-val (x lst)
  (cond ((endp lst) nil)
        ((= (car lst) x) lst)
        (t (search-val x (cdr lst)))))
```

Listing 15: A function which searches for the first occurrence of a given element `x` in a list `lst` and, if found, returns the remainder of the list (including `x`). The cost of this function (more specifically, the number of recursive calls) depends on whether the equality check succeeds or not, which is specific to the argument `x` and cannot be determined if `x` is unknown.

To see how the average-case cost could be derived, consider the example in Listing 15 again. Here, the probability of the test succeeding (i.e., a list element being equal to `x`) could be expressed as, e.g., $P(\texttt{car}(\mathit{lst}) = x)$. For the sake of simplicity, we assume for now that this probability is the same for all list elements. Consequently, the average cost is proportional to $\sum_{n=1}^{x_{\text{len}}}(1 - P(\texttt{car}(\mathit{lst}) = x))^n$, since the recursion stops as soon as the test succeeds. $P(\texttt{car}(\mathit{lst}) = x)$ is thus both a parameter of the cost expression and a measure of the argument (i.e., it expresses a specific property of the argument, namely the probability of a list element being equal to `x`), same as the list length x_{len}, and must be determined in order to calculate the actual cost. Considering the cost expression, it becomes clear why average-cost analysis can be important – even for low probabilities $P(\texttt{car}(\mathit{lst}) = x)$, the result is considerably less than x_{len} (which corresponds to the worst case where $P(\texttt{car}(\mathit{lst}) = x) = 0$).

In general, statistical analysis is a topic which does not appear to be frequently considered in more recent literature. RAML can generate bounds which include parameters such as the fraction of a specific member of a union type within a list. This method permits a sort of average-case analysis on structural properties of the input, but cannot be directly extended to semantic ones, and cannot calculate bounds which are asymptotically lower than the worst case. A more extensive approach for average-case analysis was proposed by Wegbreit as part of Metric [Weg75], where the "probabilities of unanalyzable tests appear as parameters". It combines multiple properties of a cost measure, specifically its minimum, maximum (i.e., upper bound), mean and variance. Schellekens [Sch10] presents an interesting approach which employs compositional average-case analysis, applied to a custom language. It is based on the concept of *random bags*, "which are used to represent data distribution" [Sch10]. Schellekens' work also includes an application of the method to quicksort, for which an average-case cost of $2(n + 1)\ln(n)$ is obtained.

Fundamentally, producing both concise and precise average-cost bounds for more complex programs is challenging. In the examples presented by Wegbreit [Weg75], even very simple functions result in complex expressions for the mean and especially for the variance. Again, accepting a certain loss of precision might be unavoidable. Given the running time issues (frequent timeouts) even in our current model (which calculates only upper bounds), the computational effort could be prohibitive. Furthermore, obtaining truly relevant information might actually require more advanced statistical information depending on complex properties of the input, which might not be feasible to determine automatically. Statistical dependences between the conditions would also have to be analyzed [Weg75]. In any case, sufficient statistical information might simply not be available or not meaningful – for the aforementioned quicksort example, detailed information on the ordering of input values would be necessary. Even the very simple `search-val` function requires detailed knowledge on which data is passed to it – while this is trivial in cases where the function is called with a constant input, obtaining such information in the context of a complex program is much harder.

6.5 Other programming paradigms

Beyond Lisp, there are a wide variety of functional programming languages with different paradigms or additional features. In related work, systems are described which are capable of analyzing some of those languages. Our current analysis could be extended in various ways in order to cover some of these areas as well.

6.5.1 Higher-order functions

Higher-order functions accept one or more arguments which are themselves functions, here referred to as “function-arguments” for clarity. Support for higher-order functions would be very important for analyzing application programs in most functional languages, beyond ACL2 models. In that context, higher-order functions are typically considered a core feature, improving modularity and structure [Hug89], and are therefore frequently used.

As an example, in a functional programming language, functions such as `list-factorials` in Listing 18 (Section 7.1) would normally be implemented using a higher-order `map` function, thus replacing specialized iterating functions with one generalized higher-order one. Currently, this is not possible in our limited subset of Lisp, which does not allow for higher-order functions.

```
(defun higher-order (f x)
    (funcall f (* x 2)))
```

Listing 16: A very simple higher-order function which applies the function-argument f to another argument x multiplied by 2. Whereas the cost of `higher-order` itself is constant and can be easily determined, the total cost of a call to `higher-order` depends on the cost of `f`, which is not known.

For determining the cost of specific calls to higher-order functions, a relatively simple approach is sufficient. Here, the cost of *known* function-arguments (in the context of a specific call with a known argument) is factored directly into the bound. Take, for example, the higher-order function defined in Listing 16. The cost of a call to `higher-order` with a concrete function-argument could be directly determined, as in `(higher-order #'g 5))` given `(defun g (x) (* x x))`. This call returns $(2 * 5)^2$, where `#’g` is the function `g` as a value. A sound cost bound for this function call could be easily determined, as the function-argument `#’g` passed to `higher-order` is known, and therefore its cost can be calculated. Conceptually, such a call is equivalent to inserting the code of the function-argument for each use of `f` in `higher-order`, and then calculating a cost bound for the resulting first-order function using the regular method described in this work. While this method is not currently supported in our analysis, it could be added directly to the frontend, without even requiring any modifications to CoFloCo.

The cost of the higher-order function itself (rather than of a specific call), however, cannot be determined using this method, nor could it be directly represented within the current bound format. Here, the cost of unknown function-arguments could simply be ignored, as is done in the current implementation of RAML [HDW17]. For `higher-order` (Listing 16) itself, only the cost of the multiplication operation performed by the function would be considered. While this approach simplifies the analysis, the resulting bounds are of course no longer sound with respect to the total cost.

Another, more powerful and most importantly sound approach would be to parameterize the cost bound of the function-argument as a symbolic expression of its inputs, rather than ignoring it. Of course, in restricted cases as described above, where the function-argument is known, the cost could still be directly determined. Ideally, such a solution should be implemented directly in CoFloCo rather than in the Lisp-specific frontend, such that higher-order functions in arbitrary input languages (given suitable frontends) could be analyzed. For the example in Listing 16, the bound would contain an expression similar to

$$\mathit{boundOf}(f, (x_{\text{int}}, x_{\text{len}}, x_{\text{size}}))$$

such that the final bound for a call with a specific f could then be obtained by inserting the bound calculated for f in terms of the input measures $x_{\text{int}}, x_{\text{len}}, x_{\text{size}}$. For a call such as `(higher-order #'factorial n)` (where n is a list of integers, and `factorial` is defined as in Listing 3), the bound $\text{nat}(n_{\text{int}}) * 5 + 5$ would be obtained. Another advantage of such an approach would be that it could tolerate undefined functions in general, by parametrizing their costs as well.

However, such a compositional approach would make amortized analysis impossible, as only the maximum cost for each invocation of the function-argument f would be considered. While we have not investigated this topic in depth yet, it is not immediately clear how, if at all, a bound could concisely express the cost of a higher-order function including an accurate representation of automatically determined amortization effects.

6.5.2 Lazy evaluation

Lisp is generally evaluated eagerly, except for the branches of `if` blocks. However, other functional languages have different evaluation strategies: Haskell, for example, uses lazy evaluation (specifically call-by-need argument passing). Thus, the cost of a function depends on whether each function application actually needs to be evaluated. It also becomes possible to write terminating functions which take streams (i.e., infinite lists) as input, as long as only a finite number of list elements are actually evaluated. In comparison, such a program would never terminate in an eagerly evaluated functional language, as it would attempt to generate the entire (infinite) list before calling the function.

Lazily evaluated programs are generally much harder to reason about from a resource analysis perspective, since the simple static evaluation order of eagerly evaluated programs is replaced with a dynamic, "demand-driven" one [Sim+12]. It is especially difficult to analyze space (heap or stack) usage, due to the non-strict evaluation order and data sharing among others [GS99].

There appear to be relatively few approaches in this area so far. Simões et al. [Sim+12] present an approach for amortized resource analysis of lazy higher-order functional languages, with an application to heap usage. The implementation processes a Haskell-like input language, using type inference to "[capture] the costs of unevaluated expressions in type annotations and by amortizing the payment of these costs" [Sim+12]. However, it is limited to programs with linear cost, and a large-scale evaluation was not performed. Vasconcelos et al. [Vas+15] expand on this work in order to analyze co-recursion on infinite streams. The effect of deallocation (e.g., through garbage collection) is not modeled, meaning that only cumulative allocations (rather than resident size) are analyzed. The system also has similar limitations as the previous one by Simões et al., being able to derive only linear bounds.

6.5.3 Parallel programming

Due to the absence of side effects, functional programs are generally easier to parallelize than imperative programs – even automatic parallelization is possible to some degree (see, e.g., Hogen et al. [HKL92]). Imperative parallel programs can have complex interaction patterns between threads due to shared variables and memory, as well as explicit synchronization, which can be highly challenging to analyze. For purely functional programs, on the other hand, no such sharing is possible, making them much easier to reason about [Roe91].

The impact which the parallelization of (parts of) a program has on its cost depends on the cost model underlying the analysis, and more broadly on which of the program's properties are being investigated. For example, if resource analysis is being performed in order to determine actual ("wall clock") running time in the context of a real-time system, the correct analysis of concurrent execution is important for obtaining a precise bound. This is even more critical for resource metrics which actually increase due to parallel execution, such as the total stack size.

ACL2 has experimental support for parallel execution[2], for which certain primitives (such as parallel `let` and function calls with parallel evaluation of the arguments) are provided. Although this does not appear to be frequently used in the benchmarks, it could nonetheless form a basis for expanding the existing analysis to parallel programming.

[2]`http://www.cs.utexas.edu/users/moore/acl2/v7-0/manual/index.html?topic=ACL2___ _PARALLEL-PROGRAMMING`

CHAPTER 7

Possible directions for future work

As the results of the evaluation (see Section 5.2) show, the analysis yields good results, but there is still room for improvements. On this basis, we discuss a number of possible improvements, which would address issues encountered in the course of this work.

7.1 Extended measures

The current fixed set of measures implies certain limitations of the power and accuracy of the analysis. We propose several approaches for adding specific new measures which help to avoid some of these limitations, most notably the dynamic addition of likely relevant measures, based on type inference.

7.1.1 Dynamic measures using type inference

Limitations of fixed measures

Generally, the precision of the analysis depends on the choice of measures [AGG13]. The main disadvantage of using only a limited set of predefined measures is the inability to calculate precise costs for more complex patterns, such as iteration over the lists contained in a list. Here, the length measure is no longer sufficient (as it does not describe nested lists), whereas the size is potentially imprecise (as we argue next). An example of a function with such a pattern is shown in Listing 17.

Such problems occur most frequently where a bound depends on the value of the head (`car`) of a list, as in the example in Listing 17. If the cost depends only on the length or size of the list element, we obtain a sound bound proportional to the size measure of the list. Considering the function `list-lengths` again, the size is indeed an accurate representation of the true cost, assuming that `x` is a list of lists of atoms (e.g., integers), such as `'((1 2) (3 4 5) (6))`. However, this becomes inaccurate if `x` is a more deeply

```
(defun list-lengths (x)
  (if (consp x)
    (cons (listlen (car x)) (list-lengths (cdr x)))
    nil))
```

Listing 17: Nested iteration over a list of lists: the function determines the length of each sublist and returns the resulting list of lengths. Our analysis derives the bound $4 * x_{\text{len}} + 5 * x_{\text{size}} + 3$, i.e., the bound references not only the length, but also the size of x. However, a more precise bound would be the sum of the lengths of the sublists plus the cost of the top-level iteration. `listlen` is defined in Listing 2 on Page 18.

nested structure (a list of lists of lists etc.), for example `'((1 2) (3 (4 5 (6 7) 8)) (9))`. In such cases, a bound proportional to the size is a significant overapproximation, as in reality, the cost depends only on the first two nesting levels. In the example, the size of the sub-sublist `'(4 5 (6 7) 8)` counts towards the total size of the argument, even though it is irrelevant for the cost of the iteration in `list-lengths`.

For lists of integers on the other hand, where the complexity of a function which is called on each element of the list depends on the integer value of that element, the current approach fails entirely. An example is shown in Listing 18, where a correct bound would depend on the value of integers contained within a list. As only the three predefined measures defined in Section 2.3 are available, it is not possible to express this cost as a combination of the measures for the argument `x`. Consequently, no bound can be calculated for `list-factorials`.

```
(defun list-factorials (x)
  (if (consp x)
    (cons (factorial (car x)) (list-factorials (cdr x)))
    nil))
```

Listing 18: A function which calculates the factorial for each value in the input list. `factorial` is defined in Listing 3. The cost of this function depends on both the length of the list and the values of its element, specifically the sum of the cost of `factorial` for each element, plus the cost of the iteration.

Determining measures suitable for calculating precise bounds becomes even harder with “semantic” patterns which include multiple recursive calls, such as alternately traversing the left and right branch in a tree. However, such functions are presumably very rare.

For the examples shown here, it is possible to add measures for the corresponding iteration patterns (such as iteration over each sublist of a list) directly to the set of predefined measures, under the assumption that these patterns occur somewhat frequently. However, this clearly works only for a limited number of such patterns, whereas iterations could in principle be nested to an arbitrary depth. For example, Listing 19 shows a function which calls the `list-lengths` function defined above (Listing 17) on each list element. In other words, it extends the iteration by an additional level.

```
(defun nested-list-lengths (x)
  (if (consp x)
    (cons (list-lengths (car x)) (nested-list-lengths (cdr x)))
    nil))
```

Listing 19: Doubly nested iteration which determines the lengths of the sublists of lists contained within a list. `list-lengths` is defined in Listing 17.

In fact, similar deep nesting occurs in many real-world programs utilizing complex data structures. Typically, "higher-level" functions call other functions acting on successively smaller nested parts of the initial argument(s). Furthermore, as explained in Section 3.2.1, the computational effort required to analyze a program rises quickly with the number of measures involved. Therefore, the large number of measures added by such an approach might improve precision, but cause scalability issues, and the high proportion of "useless" ones needlessly slows down the analysis.

Solution

In order to avoid these limitations, measures which might be relevant or helpful for the analysis should be identified and added dynamically, rather than relying on a constant set of predefined measures. Measures could be constructed based on the structure and type information of the program, as in Albert et al. [AGG13]. The core concept here is type inference, i.e., automatically determining possible types for function arguments and variables in an a-priori untyped program. The approach presented by Albert et al. is focused on recursive types, which are quite similar to lists in Lisp (which consist of nested `cons` pairs). Given a dynamically typed Lisp program, such an approach may result in multiple candidate measures, which however does not pose any difficulties for CoFloCo. In this case, scalability is still improved, as only relevant measures are added.

Consider the example `list-factorials` in Listing 18 again, which iterates over the elements of a list, where each element has a processing cost depending on its integer value. Since `factorial` requires its input to be an integer (which follows from its use of the integer operations `>` and `*`, c.f. the definition in Listing 3 on Page 30), it can also be inferred that `list-factorials` requires a list of integers. This leads to the assumption that the maximum of the integer values in the list might be a suitable measure.

Using this newly generated measure, which maps a list to the maximum integer value among its elements, the behavior of the function could be correctly specified by cost relations. This measure is used in the same way as the usual list measures, such as length. In cost relations, the integer measure of each element in the list can then be constrained to at most the maximum value among all list elements. This results in a sound overapproximation of the total cost, as each element is assumed to be maximal.

The cost equation for `car`, originally

$$\text{eq}(\texttt{car}(x_{\text{size}} \to y_{\text{size}}), 1, [\,], [y_{\text{size}} \leq x_{\text{size}} - 1])$$

is modified by adding a measure $\text{listmax}_{\text{int}}(x)$ for the maximum integer value in the list `x`, as well as the corresponding constraint $y_{\text{int}} \leq \text{listmax}_{\text{int}}(x)$:

$$\text{eq}(\texttt{car}(x_{\text{size}}, \text{listmax}_{\text{int}}(x) \to y_{\text{int}}, y_{\text{size}}), 1, [\,], [y_{\text{int}} \leq \text{listmax}_{\text{int}}(x), y_{\text{size}} \leq x_{\text{size}} - 1])$$

CoFloCo could then calculate the final bound as proportional to $x_{\text{len}} * \text{listmax}_{\text{int}}(x)$, i.e., the product of the list length and the maximum value. This bound can also be obtained by RAML.

Likewise, for the nested list iteration in `list-lengths` (Listing 17), type inference can determine that the argument must be a list of lists of some element type. From the fact that no type constraints can be inferred for this type, it follows that the corresponding elements of the sublists are not used in `list-lengths` and therefore do not affect the function's cost. Consequently, these elements are irrelevant for calculating the bound, and no measures need to be generated for them. As a result, the analysis would generate the measures x_{len} (the length of the argument `x`) and $\text{listmax}_{\text{len}}(x)$ (the maximum sublist length), with the cost being proportional to their product. An analogous measure for the sub-sublists in `nested-list-lengths` (Listing 19) could be $\text{listlistmax}_{\text{len}}(x)$, and so on for any other finite nesting depth.

An even better (more precise) measure would be an expression such as "sum of the integer values of the list elements", written, e.g., $\text{listsum}_{\text{int}}(x)$. `list-factorials` would then have the following cost relations:

$$\begin{aligned}
&\text{eq}(\mathit{if}_n(x_{\text{len}}, \text{listsum}_{\text{int}}(x) \to \dots), 1,\\
&\qquad [\texttt{factorial}(y_{\text{int}} \to r_{\text{int}}), \texttt{list-factorial}(z_{\text{len}}, \text{listsum}_{\text{int}}(z) \to w_{\text{int}}), \dots],\\
&\qquad [x_{\text{len}} > 0, \text{listsum}_{\text{int}}(x) = y_{\text{int}} + \text{listsum}_{\text{int}}(z)])\\
&\text{eq}(\mathit{if}_n(x_{\text{len}}, \text{listsum}_{\text{int}}(x) \to \dots), 1, [\,], [x_{\text{len}} = 0])\\
&\text{eq}(\texttt{list-factorials}(x_{\text{len}}, \text{listsum}_{\text{int}}(x) \to \dots), 1, [\mathit{if}_n(x_{\text{len}}, \text{listsum}_{\text{int}}(x) \to \dots], [\,])
\end{aligned}$$

As the precise bound for `factorial` is $\text{nat}(n_{\text{int}}) * 5 + 3$ and the iteration itself has some cost c_{iter}, the bound $c_{iter} + \text{listsum}_{\text{int}}(x) * 5 + x_{\text{len}} * 3$ for `list-factorials` could be calculated based on these cost relations.

Of course, the same would also be possible for the elements of a tree, as well as any other kind of structure, e.g., lists of lists. However, this approach is only suitable if the cost of the called function is linear in terms of each list element.

Nonetheless, even more complex (and more precise) measures are possible, following a similar structure. This approach is not limited to integer list elements and could represent arbitrary expressions for the elements, resulting in a sum expression such as $\text{listsum-fns}_{\text{int}}(x, f(y))$ where x is the list being iterated over, $f(y)$ is a function over each integer element y of the list, and the value of the measure is the sum of $f(y)$ applied to each element. The cost of the iteration itself must again be added separately.

Such complex measures would however only be feasible if they were generated/selected by another sophisticated analysis pass. Generally, as the complexity of the functions being analyzed increases, it becomes harder to determine which measures are suitable (or most useful) for representing the function's cost. Furthermore, a higher number of measures would adversely affect scalability again. In any case, the usefulness of a bound containing such a sum expression depends on the use case.

7.1.2 Logarithmic bounds

In addition to polynomial bounds, the analysis could be extended to also support logarithmic bounds in certain cases. This would be possible within the existing linear constraint-based system by treating the logarithm as a regular measure which changes linearly between recursive calls. For example, a function which divides its argument `x` by 2 in each recursive call could have an additional measure $\log_2 x_{\text{int}}$ (i.e., the base-2 logarithm of the integer measure x_{int}), which decreases by 1 with each call. Listing 20 shows an example of such a function together with the corresponding cost relations, including a logarithmic measure.

```
(defun lg2 (x)
  (if (> x 1)
    (+ 1 (lg2 (/ x 2)))
    0))
```

$$\begin{aligned}
&\mathrm{eq}(\texttt{/}(\log_2 x_{\text{int}}, y_{\text{int}} \to \log_2 r_{\text{int}}), 1, [\,],\\
&\quad [y_{\text{int}} = 2, \log_2 r_{\text{int}} = \log_2 x_{\text{int}} - 1])\\
&\mathrm{eq}(\texttt{lg2}(\log_2 x_{\text{int}} \to r_{\text{int}}), 2,\\
&\quad [\texttt{>}(\log_2 x_{\text{int}}, 0, c_{\text{int}}), \texttt{/}(\log_2 x_{\text{int}}, 2, \log_2 z_{\text{int}}),\\
&\quad \texttt{lg2}(\log_2 z_{\text{int}}, rn_{\text{int}}), \texttt{+}(rn_{\text{int}}, 1, r_{\text{int}})], [c_{\text{int}} = 1])\\
&\mathrm{eq}(\texttt{lg2}(\log_2 x_{\text{int}} \to 0), 2, [\texttt{>}(\log_2 x_{\text{int}}, 0, c_{\text{int}})], [c_{\text{int}} = 0])
\end{aligned}$$

Listing 20: A function which calculates the base-2 logarithm of an integer (rounded up) by successively dividing the argument by 2. It is immediately obvious that the number of recursive calls is likewise defined by the base-2 logarithm of the argument x, i.e., $\log_2 x$. Thus, by adding a measure $\log_2 x_{\text{int}}$ for the logarithm of the integer measure of `x` (as seen in the cost relation to the right), a bound could be formulated as a linear expression proportional to $\log_2 x_{\text{int}}$. The first cost equation shows the modified form of the division function, specialized to the case where the divisor is 2: division by 2 decrements the base-2 logarithm by 1.

An important restriction for this approach is that the sole change to the argument must be a linear increase or decrease of its logarithm. This means that programs containing recursive calls such as $f(x/2 + 1)$ could not be successfully analyzed using this method. Listing 21 shows a function `lg2-plus1` where this is the case, although it is otherwise identical to `lg2`.

```
(defun lg2-plus1 (x)
  (if (> x 2)
    (+ 1 (lg2-plus1 (+ (/ x 2) 1)))
    0))
```

Listing 21: Compared to `lg2` in Listing 20, the minor modification of adding 1 to the argument of the recursive call in `lg2-plus1` means that this logarithmic measure is no longer applicable, because the logarithm of `x`, i.e., the measure $\log_2 x_{\text{int}}$, is not simply decremented or incremented by a constant between calls.

A logarithmic measure could also be useful for analyzing divide-and-conquer based algorithms such as merge sort (see Listing 7 in Section 3.2.6, Page 38). This function has a worst-case complexity of $O(n \log n)$, whereas our current approach derives a bound with complexity $O(n^2)$. The partition step (splitting into elements with even respectively odd position in the list) poses a complication, as it may result in one branch being called with an argument which is slightly longer than half the input: for an input list of odd length x_{len}, one call will receive a list of length $\lceil x_{\text{len}}/2 \rceil$. Nonetheless, the total cost is still as expected.

```
(defun bin-tree-search (tr x)
  (cond ((null tr) nil)
        ((< x (first tr)) (bin-tree-search (second tr) x))
        ((> x (first tr)) (bin-tree-search (third tr) x))
        (t x)))  ; element found
```

Listing 22: Search function for a binary search tree containing integers. It is assumed that the tree is well-formed, with each node consisting of a three-element list (*e*, *left*, *right*). Therefore, the bound should be proportional to the maximum number of steps which need to be taken in order to find the item (or determine that it does not exist), which in turn is equal to the depth of the tree.

Additionally, it would be possible to add measures which are inherently logarithmic in some sense. Consider, for example, the time it takes to find an element in a binary search tree, using a simple search function as shown in Listing 22. For this function, an interesting candidate measure is tree depth (also called “height” [HDW17], and presented by Campbell [Cam09] with an application to memory analysis). An identical addition, as a consequence of encountering similar problems with specific predefined measures, was proposed by Wegbreit: “car-length [and] max-length (maximum path along any combination of `car` or `cdr` options in `cons` pairs)” [Weg75]. This depth measure would in fact be the logarithm of our current *size* measure, assuming a fully populated binary tree, and would be defined as follows:

$$depth(x) = \begin{cases} 1 + \max(depth(a), depth(b)) & \text{if } x = a \texttt{ . } b \\ 0 & \text{otherwise} \end{cases}$$

Such a measure would be helpful for certain other analysis tasks as well, such as determining the maximum stack size for depth-first search of a tree.

7.1.3 Manual annotations

For some functions, automatically identifying the measures and properties necessary for cost analysis is not possible or feasible, or at least very hard. Handling such complex cases automatically might still be possible in some circumstances, but would require a large number of heuristics and/or spending significant computational effort on attempting to analyze them.

Instead, it would likely be more useful to allow the user to add manual annotations which specify an expression to use as a measure. For example, unless CCG is enabled, ACL2 depends on such annotations, which specify hints used for proving termination.

Specifying implicit or external knowledge would also be important in some cases, such as for graph algorithms where termination depends on the graph being acyclic. This could also be done through manual annotations – in fact, ACL2 already provides extensive annotation mechanisms, e.g., in the form of *guard* annotations[1], which could be used for resource analysis as well. In a similar way, arbitrary user-specified lemmas could be taken into account to support various analysis tasks, such as proofs for multiple recursion (see Section 3.2.6).

7.1.4 Encoding type information

While Lisp is not statically typed, types often determine the behavior of functions. An inability to distinguish between types may lead to unsound results in certain rare cases, such as the confusion of integer and symbol values (see Section 3.3). The example of `int-bool-loop` in Listing 23 shows an instance of this problem (however, note that this is a synthetic example, which is not likely to occur in reality). The underlying cause of the error is that type information is lost during the conversion to cost relations, and consequently, it is no longer possible to distinguish between the integer value 0 and the symbol `nil`. Although `not` should return `t` only if the argument is `nil`, according to the semantics described in the cost relation for `not`, it would erroneously also return `t` for 0, and thus abort the loop early:

$$\mathrm{eq}(\texttt{not}(a_{\mathrm{int}} \rightarrow r_{\mathrm{int}}), 1, [\,], [a_{\mathrm{int}} = 0, r_{\mathrm{int}} = 1])$$
$$\mathrm{eq}(\texttt{not}(a_{\mathrm{int}} \rightarrow r_{\mathrm{int}}), 1, [\,], [a_{\mathrm{int}} \neq 0, r_{\mathrm{int}} = 0])$$

[1] `http://www.cs.utexas.edu/users/moore/acl2/v7-0/manual/index.html?topic=ACL2____GUARD`

Note that in this case, `not` (similar to `if`) is correctly defined only for symbol arguments `t` and `nil` (unlike in Common Lisp, where applying `not` to any value other than `nil` returns `nil`).

```
(defun int-bool-loop (x)
  (+ 1 (cond ((not x) 0)
             ((> x -10) (int-bool-loop (- x 1)))
             (t 0))))
```

Listing 23: A synthetic example where deriving a sound bound depends on the ability to distinguish between an integer with value zero (which is considered as true in the context of a condition) and `nil` (which is false). As a result, the bound generated by our analysis reflects a maximum of only x recursive calls, whereas in reality, up to $x + 9$ recursive calls may be made. In the `cond` block, the final condition `t` is the default or "otherwise" option. This example is repeated from Listing 8.

One way of resolving the issue would be to add a separate measure which explicitly encodes the type of the value. Specific integer values would correspond to specific types (such as integer, symbol and `cons` pair, and possibly other types which are currently not supported, but could then be added more easily). In this way, the type information is preserved and expressed as part of the cost relations. The analysis can then determine a choice of possible integer values corresponding to possible types – for example, the input of `length` must be either a non-empty list or `nil`, and the output is an integer. Thus, it becomes possible to distinguish between different types of values of a function argument, which cannot be reliably distinguished based solely on the existing integer and size/length measures.

The definition of the basic function `not` could be modified to specify that the result is false (i.e., `nil`, with the corresponding integer value 0) only if the value is of type symbol. Simplified cost relations are given below, where the first cost equation concerns the case where the argument is `nil`, the second where it is a symbol other than `nil`, and the third where it is a value of any other type. x_{type} is the type-indicating measure, and $\mathrm{t}_{\texttt{symbol}}$ is an integer constant which specifies that the value described by the type measure is a symbol:

$$\begin{aligned}
&\mathrm{eq}(\texttt{not}(x_{\text{int}}, x_{\text{type}} \to 1, \mathrm{t}_{\texttt{symbol}}), 1, [\,], [x_{\text{type}} = \mathrm{t}_{\texttt{symbol}}, x_{\text{int}} = 0])\\
&\mathrm{eq}(\texttt{not}(x_{\text{int}}, x_{\text{type}} \to 0, \mathrm{t}_{\texttt{symbol}}), 1, [\,], [x_{\text{type}} = \mathrm{t}_{\texttt{symbol}}, x_{\text{int}} > 0])\\
&\mathrm{eq}(\texttt{not}(x_{\text{int}}, x_{\text{type}} \to 0, \mathrm{t}_{\texttt{symbol}}), 1, [\,], [x_{\text{type}} \neq \mathrm{t}_{\texttt{symbol}}])
\end{aligned}$$

The first cost equation handles the case where x is the symbol *nil* (which has the integer value 0 by convention), meaning that the result is `t` (with integer value 1). The second cost equation applies for symbols other than `nil` (note that the integer values assigned to symbols are always positive, hence the condition > 0), where the result is `nil`. The final cost equation handles all cases where x is not a symbol. The behavior of this revised cost relation corresponds precisely to that of `not` in Common Lisp and ACL2.

For `int-bool-loop` in Listing 23, the analysis would now determine that `x` might be either an integer or `nil`, with the important difference being that the definition of `not` now reliably distinguishes between the two cases. Since the analysis has to assume the worst case (in this case, that the condition which would cause an early exit from the recursion is not met), the resulting bound would be sound.

Type information encoded in the cost relations, such as possible types of arguments and type-specific behavior, could be used for other analysis steps as well. For example, type-inference based techniques like the one described in Section 7.1.1 could rely on such specifications and would thus be able to analyze any program expressed as cost relations, rather than being specific to the Lisp frontend.

7.2 Bound simplification

As explained in Section 3.2.2, many bound expressions are too complicated to be intuitively understood, and the corresponding degree of precision might not actually be required, depending on the use case. Currently, only basic arithmetic simplifications (such as grouping of constant additions), which do not affect the result, are applied to bounds. While this could still be extended to some degree, it cannot reduce inherent complexity. Some loss of precision stemming from simplifications of the bound expression (such as removal of constant terms or grouping of all multiplications with the asymptotically most significant term) might therefore need to be accepted.

Similarly, the simplistic approach to supporting higher-order functions described in Section 6.5.1, where the (unknown) function argument is simply ignored, might in fact be desirable for certain applications. It is often more interesting to determine how often the function argument of a higher-order function is executed, whereas the detailed cost is not important.

7.3 Comparisons with other systems

As described in Section 5.4.1, experimental comparisons of our implementation to other resource analysis systems would be interesting. However, as previously explained in Section 5.4.1, such comparisons involve various challenges, as the equivalence of the semantics and cost models must be ensured in order to obtain comparable bounds. As discussed in previous sections, the space of functional programming is highly diverse, with quite different paradigms (e.g., statically vs. dynamically typed, eager vs. lazy execution). A meaningful experimental comparison would necessitate either the design and implementation of either appropriate frontends or conversion methods, or implementing various existing approaches for a single target language. For example, the implementation of a frontend for OCaml, but using the same or similar approaches as described in this work, would be interesting for comparing our analysis to RAML. Since our analysis does not require types, but can optionally use type information, such an extension would be relatively simple in principle. However, OCaml is considerably more complicated than

Lisp, which is one of the reasons why we chose to focus on the latter at first (the fact that no “simplified” benchmarks similar to the ACL2 models are available for OCaml is another).

CHAPTER 8

Conclusion

We adapted CoFloCo, an existing resource analysis system, in order to process Lisp input, and applied it to process a large benchmark suite containing over 19491 functions. The analysis is completely automatic, does not require manual annotations or other user interaction, is typically fast where it can obtain a result, and can calculate bounds for a considerable fraction of the input. The results are promising, with bounds being obtained for more than 53% of the functions in the benchmark. However, these results also clearly show that additional work is still required, especially concerning performance (running time).

Judging from both these results and comparisons to related work, it appears that resource analysis in general has advanced quite far. The challenges generally lie in identifying useful abstractions, and their quality appears to be the most important factor – according to the literature, this is common among related approaches as well. The most significant challenges likely lie in applying this work to real-world use cases involving very large and complex programs. There are also some remaining points, such as comprehensive analysis of generalized higher-order functions, which are yet to be completely solved.

A notable consideration which has been paid little attention so far is the intended application or use of the obtained bounds, which influences the desired characteristics of the analysis. Merely producing (precise) bounds is not sufficient for all applications – if information should be provided to the user, the utility of complex bound expressions is doubtful, as they are hard to understand intuitively. Therefore, it might be necessary to simplify these bounds, or to analyze only higher-level parts of the program while ignoring irrelevant details. In order to make resource analysis directly useful for programmers, integration into existing toolchains (such as integrated development environments) would also be desirable.

List of Figures

List of Tables

List of Listings

Bibliography

[AGG13] Elvira Albert, Samir Genaim, and Raúl Gutiérrez. “A Transformational Approach to Resource Analysis with Typed-Norms”. In: *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation.* Ed. by Gopal Gupta and Ricardo Peña. Springer. 2013, pp. 38–53. DOI: 10.1007/978-3-319-14125-1_3.

[Alb+07] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. “COSTA: Design and implementation of a cost and termination analyzer for Java bytecode”. In: *Proceedings of the 6th International Symposium on Formal Methods for Components and Objects.* Ed. by Andrew D. Gordon. Springer. 2007, pp. 113–132.

[Alb+11] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. “Closed-form upper bounds in static cost analysis”. In: *Journal of Automated Reasoning* 46.2 (2011), pp. 161–203. DOI: 10.1007/s10817-010-9174-1.

[Alb+15] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. “A practical comparator of cost functions and its applications”. In: *Science of Computer Programming* 111 (2015), pp. 483–504.

[Cam09] Brian Campbell. “Amortised memory analysis using the depth of data structures”. In: *Proceedings of the 18th European Symposium on Programming.* Ed. by Giuseppe Castagna. Springer. 2009, pp. 190–204.

[Cor+01] Thomas H Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT Press Cambridge, 2001.

[FH14] Antonio Flores-Montoya and Reiner Hähnle. “Resource analysis of complex programs with cost equations”. In: *Proceedings of the 12th Asian Symposium on Programming Languages and Systems.* Ed. by Jacques Garrigue. Springer International Publishing. 2014, pp. 275–295. DOI: 10.1007/978-3-319-12736-1_15.

[Flo16] Antonio Flores-Montoya. “Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations”. In: *Proceedings of the 21st Inter-*

national Symposium on Formal Methods (FM'16). Ed. by John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou. 2016. DOI: `10.1007/978-3-319-48989-6_16`.

[Gre+15] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. "Static analysis of energy consumption for LLVM IR programs". In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. ACM. 2015, pp. 12–21.

[GS99] Jörgen Gustavsson and David Sands. "A foundation for space-safe transformations of call-by-need programs". In: *Electronic Notes in Theoretical Computer Science* 26 (1999), pp. 69–86.

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Multivariate amortized resource analysis". In: *ACM SIGPLAN Notices*. Vol. 46. 1. ACM. 2011, pp. 357–370.

[HDW17] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. "Towards Automatic Resource Bound Analysis for OCaml". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*. ACM, 2017, pp. 359–373. DOI: `10.1145/3009837.3009842`.

[HKL92] Guido Hogen, Andrea Kindler, and Rita Loogen. "Automatic parallelization of lazy functional programs". In: *Proceedings of the 4th European Symposium on Programming*. Ed. by Bernd Krieg-Brückner. Springer. 1992, pp. 254–268. DOI: `10.1007/3-540-55253-7_15`.

[Hof16] Jan Hoffmann. *Resource Aware ML*. `http://raml.co`. Accessed: 2016-09-05. 2016.

[HS15] Jan Hoffmann and Zhong Shao. "Type-based amortized resource analysis with integers and arrays". In: *Journal of Functional Programming* 25 (2015), e17.

[Hug89] John Hughes. "Why functional programming matters". In: *The Computer Journal* 32.2 (1989), pp. 98–107.

[Hun+10] Warren A Hunt Jr, Sol Swords, Jared Davis, and Anna Slobodova. "Use of formal verification at Centaur Technology". In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Ed. by David S. Hardin. Springer, 2010, pp. 65–88.

[KM16] Matt Kaufmann and J Strother Moore. *ACL2 Homepage*. `http://www.cs.utexas.edu/users/moore/acl2/`. Accessed: 2016-08-30. 2016.

[KM97] Matt Kaufmann and J. Strother Moore. "An industrial strength theorem prover for a logic based on Common Lisp". In: *IEEE Transactions on Software Engineering* 23.4 (1997), pp. 203–213.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[McC65] John McCarthy. *LISP 1.5 programmer's manual.* MIT Press, 1965.

[McC79] John McCarthy. *The implementation of Lisp.* `http://www-formal.stanford.edu/jmc/history/lisp/node3.html`. Accessed: 2016-10-27. 1979.

[MML97] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. "Static timing analysis of embedded software". In: *Proceedings of the 34th annual Design Automation Conference.* ACM. 1997, pp. 147–152.

[Mon+11] Manuel Montenegro, Olha Shkaravska, Marko Van Eekelen, and Ricardo Peña. "Interpolation-based height analysis for improving a recurrence solver". In: *Proceedings of the 2nd International Workshop on Foundational and Practical Aspects of Resource Analysis.* Ed. by Ricardo Peña, Marko van Eekelen, and Olha Shkaravska. Springer. 2011, pp. 36–53.

[MV06] Panagiotis Manolios and Daron Vroon. "Termination analysis with calling context graphs". In: *Proceedings of the 18th International Conference on Computer Aided Verification.* Ed. by Thomas Ball and Robert B. Jones. Springer. 2006, pp. 401–414. DOI: `10.1007/11817963_36`.

[Roe91] Paul Roe. "Parallel programming using functional languages". PhD thesis. University of Glasgow, 1991.

[Rus+05] David Russinoff, Matt Kaufmann, Eric Smith, and Robert Sumners. "Formal verification of floating-point RTL at AMD using the ACL2 theorem prover". In: *Proceedings of the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation* (2005).

[SC95] David B Skillicorn and Wentong Cai. "A cost calculus for parallel functional programming". In: *Journal of Parallel and Distributed Computing* 28.1 (1995), pp. 65–83.

[Sch10] Michel P Schellekens. "MOQA; unlocking the potential of compositional static average-case analysis". In: *The Journal of Logic and Algebraic Programming* 79.1 (2010), pp. 61–83. DOI: `10.1016/j.jlap.2009.02.006`.

[Sim+12] Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. "Automatic amortised analysis of dynamic memory allocation for lazy functional programs". In: *ACM SIGPLAN Notices*. Vol. 47. 9. ACM. 2012, pp. 165–176.

[Ste90] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

[SZV14] Moritz Sinn, Florian Zuleger, and Helmut Veith. "A simple and scalable static analysis for bound analysis and amortized complexity analysis". In: *Proceedings of the 26th International Conference on Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Springer. 2014, pp. 745–761. DOI: `10.1007/978-3-319-08867-9_50`.

[SZV15] Moritz Sinn, Florian Zuleger, and Helmut Veith. "Difference constraints: an adequate abstraction for complexity analysis of imperative programs". In: *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc. 2015, pp. 144–151.

[Tar85] Robert Endre Tarjan. "Amortized computational complexity". In: *SIAM Journal on Algebraic Discrete Methods* 6.2 (1985), pp. 306–318.

[Tur36] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58 (1936), pp. 345–363.

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*. First Edition. Manning Publications Co., 2014.

[Vas+15] Pedro Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. "Type-based allocation analysis for co-recursion in lazy functional languages". In: *Proceedings of the 24th European Symposium on Programming Languages and Systems*. Springer. 2015, pp. 787–811. DOI: `10.1007/978-3-662-46669-8_32`.

[Vro07] Daron Vroon. "Automatically proving the termination of functional programs". PhD thesis. Georgia Institute of Technology, 2007.

[Weg75] Ben Wegbreit. "Mechanical program analysis". In: *Communications of the ACM* 18.9 (1975), pp. 528–539.