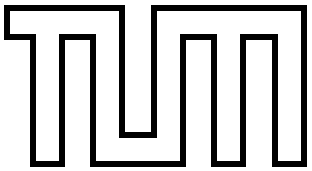


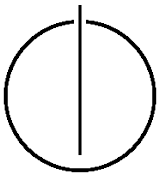
FAKULTÄT FÜR INFORMATIK

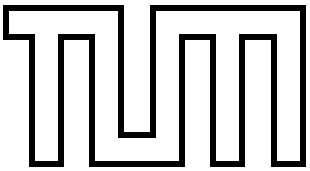
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Formally Verified Complexity Analysis of a Functional Language

Simon Wimmer





FAKULTÄT FÜR INFORMATIK

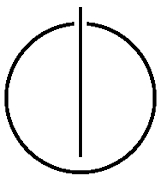
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Formally Verified Complexity Analysis of a Functional Language

Formal verifizierte Komplexitätsanalyse einer funktionalen Sprache

Author:	Simon Wimmer
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	Dipl.-Inf. (Univ.) Lars Noschinski
Submission date:	July 15, 2014



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, July 15, 2014 Simon Wimmer

Abstract

This thesis describes a framework for the formally verified runtime complexity analysis of functional programs with the help of Isabelle/HOL. A simple, deeply embedded ML-style programming language **PMλ** is defined together with a corresponding small-step operational semantics. Additionally, a complexity measure for general terminating **PMλ**-programs is defined. In order to enable employment of the framework for practical complexity analyses, fundamental properties of the language, its semantics, and the complexity measure are shown. Among these properties, the most significant one is that programs with inputs of bounded size can only exhibit a finite number of execution paths. Finally, it is demonstrated how the framework can be set to use for the worst-case complexity analysis of three sample programs – a program for list concatenation and two versions of list reversal. The body of this work is preceded by a brief discussion of relevant Isabelle preliminaries and concludes with a presentation of relevant previous work, which is compared to the approach described here, together with a discussion of possible extensions to the framework.

Diese Arbeit beschreibt ein Rahmenwerk für die formal verifizierte Laufzeit-Komplexitätsanalyse von funktionalen Programmen mit der Hilfe von Isabelle/HOL. Eine einfache, tief eingebette ML-artige Programmiersprache wird zusammen mit einer zugehörigen kleinschrittigen operationellen Semantik definiert. Zusätzlich wird ein Komplexitätsmaß für allgemeine terminierende **PMλ**-Programme definiert. Um die Benutzung des Rahmenwerks für praktische Komplexitätsanalysen zu ermöglichen, werden grundlegende Eigenschaften der Sprache, ihrer Semantik und des Komplexitätsmaßes aufgezeigt. Unter diesen Eigenschaften ist die wichtigste, dass Programme mit Eingaben von beschränkter Größe nur endlich viele mögliche Ausführungspfade aufweisen können. Abschließend wird demonstriert wie das Rahmenwerk für die Worst-Case-Komplexitätsanalyse von drei Beispielprogrammen – einem Programm für die Konkatenation von Listen und zwei Versionen der Listenumkehrung – eingesetzt werden kann. Dem Hauptteil der Arbeit geht eine knappe Diskussion von relevantem Vorwissen über Isabelle vorraus. Die Arbeit wird mit einem Überblick über relevante vorangegange Arbeiten, die mit der hier beschriebenen Vorgehensweise verglichen werden, und einer Diskussion von möglichen Erweiterungen des Rahmenwerks abgeschlossen.

Contents

Abstract **v**

I. Introduction and Theory **1**

1. Introduction **3**

2. Isabelle Prelimnaries **5**

II. Complexity Analysis with Isabelle/HOL **9**

3. A Minimalist Functional Language with Semantics **11**
3.1. A Minimalist Functional Language 11
3.2. A Small-Step Semantics 12
3.3. Important Properties of the Small-Step Semantics 16
3.4. A Big-Step Semantics 17
3.5. Language Extensions 19

4. Runtime Complexity of PMλ-Programs **21**
4.1. A Complexity Definition 21
4.2. Related Properties of the $n$small-step relation 24
4.3. Properties of the Complexity Definition 25

5. Finite Reductions for Fixed Input Sizes **27**
5.1. Proof Idea 27
5.2. Formalization Basics 28
5.3. Reduction Equivalence 30
5.4. Reduction Equivalence Class Representatives 32
5.5. Putting it Together 34

6. Complexity Analysis by Example **37**
6.1. Lists in **PMλ** 37
6.2. Complexity Analysis for *rev* 38
6.3. Complexity Analysis for *concat* 44
6.4. Complexity Analysis for *reverse* 46

III. Discussion and Conclusion **53**

7. Discussion and Related Work **55**
7.1. Comparison With Other Language and Semantics Definitions 55
7.2. Complexity Analysis for Functional Languages 57
7.3. Formalizations for Computational Theory 59

8. Conclusion **61**
8.1. Future Work 62

Bibliography **65**

Part I.

Introduction and Theory

1. Introduction

As software invades every sector of life, we also rely increasingly on software in high-security applications, e.g. in the banking business, or in life critical applications such as medical devices or control systems of vehicles. Surgical robots or self-driving cars soon will not be a future utopia anymore. These applications all share the common property that they are zero fault tolerant. Consequently, we want to have absolute confidence that the employed software functions correctly. The industrial standard for acquiring this confidence – mere testing – is not sufficient to reach absolute confidence in the correctness of the software. Only rigorous mathematical proof of the correctness of the software can reach this goal. However, a proof developed and checked by a human is just as error-prone as software developed by a human. Thus one also wants to ensure the correctness of the proof by checking it mechanically. This task can be tackled with the help of so-called *proof assistants*. They allow a human user to construct machine-checked proofs in a continuous dialog with the system, which supports the user with functionality for automatic proving. Today's proof assistants, such as Coq [35], Isabelle [30] or Nuprl [5], have evolved to a level where they can be used to verify the correctness of complex real-world software. Important milestones include the verification of an operating system microkernel [15], the verification of a C compiler [20] and a correctness proof for Java's bytecode verifier [14].

In the classical study of program properties, besides the verification of program correctness, a second type of analysis is predominant – the drive to predict a program's runtime for inputs of certain size. This property of programs is commonly referred to as a program's *runtime* complexity[1]. Although the field has been broadly investigated by research in theoretical computer science, little effort has been made to formalize the achieved results with the help of proof assistants. However, such a formalization would not only be desirable for romantics in the research community who are interested in treating both fields equally, but could also have a practical impact: in emerging security applications, such as smart cards, environmental considerations have to be taken into account, as a possible attacker can have physical access to the system. The computational speed of a system is a possible weakness that could be exploited by such an attacker. Thus the interest in building up absolute confidence in the operational speed of computer systems, by analyzing the underlying programs' algorithmic complexity in a formal way, is also bound to rise.

The purpose of this work is to make the first steps to bring the capability for such an analysis to the Isabelle proof assistant. For a formal verification of software correctness with the help of Isabelle, programs are written in Isabelle/HOL (c.f. Chapter 2), which, informally speaking, is a combination of logics and programming. These Isabelle/HOL programs can then be exported into programs for mainstream functional programming languages such as Standard ML or Haskell. For our interests, the problem with this approach is that programs are actually just Isabelle/HOL terms, which makes it impossible

[1] Throughout the remainder of the thesis, the term complexity will also refer to a program's runtime complexity.

to reason about their structure. Therefore, we cannot assign them any notion of program execution forbidding us to reason about their runtime complexity.

Consequently, in order to be able to analyze the complexity of functional programs in Isabelle/HOL, we have to use a *deep embedding* of a functional language. That is, we have to define a functional language, together with a semantics describing the language's meaning, in terms of Isabelle/HOL constructs. We can then assign this language a notion of execution, used to define a complexity measure for programs in the language. With the help of this, we will be able to state and prove assertions about the algorithmic complexity of programs in Isabelle/HOL. The long-term goal would be to show the equivalence of programs in this language to the usual Isabelle/HOL programs. This would allow us to verify the correctness of programs together with exporting their code the usual way, while the programs' complexity could be analyzed using the deeply embedded language. However, this point will remain untouched in this work.

Following Knuth [17], four characteristics are relevant for the analysis of a program's complexity – an upper and a lower bound for the program runtime and its mean and deviation. We choose to concentrate only on the analysis of upper bounds, i.e. *worst-case* complexity, as this is the most intensively studied figure in algorithms research. However, the author claims that the approach of this work is equally applicable to the analysis of the other quantities.

We give a brief description of the technical steps we will take to construct a framework for complexity analysis with Isabelle. The approach we take is to first define a compact functional language **PMλ** syntactically using Isabelle/HOL datatypes. Next, we assign a semantic meaning to this language by means of an inductive Isabelle/HOL relation that defines an operational small-step semantics for the language. This small-step semantics is then used to define algorithmic complexity for general terminating programs.

The outline of this work is as follows. Chapter 2 briefly discusses some aspects of Isabelle that are relevant for the remaining part of the work. Chapter 3 defines our language together with its semantics and states some important properties of the latter; Chapter 4 will define the complexity measure for this language. Due to the way our complexity measure is defined, in order to be able to reason about the complexity of programs, we will have to show that there are only finitely many execution paths a program can take for inputs of finite size. A proof of this property is the subject of Chapter 5, completing our framework for complexity analysis. Chapter 6 demonstrates how the framework could be used to analyze the complexity of three concrete sample programs – two versions of list reversal and a program for list concatenation. Chapter 7 discusses some previous related work and how it compares to this work. Chapter 8 concludes with a number of final thoughts and hints at possible future extensions to the framework.

2. Isabelle Prelimnaries

Isabelle [26] is a generic proof assistant in the sense that it in principal supports arbitrary user-defined logics by providing primarily a *meta-logic* (an intuitionistic fragment of higher-order logic) in which different *object-logics* can be encoded. However, for most applications the user does not provide his own object-logic, but rather works on one of the predefined object-logics Isabelle offers. Out of these, Isabelle/HOL, a formalization for common higher-order logic, is probably the one that is most frequently used. Past applications include formalizations of different aspects of mathematics and the aforementioned verification of a whole operating-system kernel [15]. Isabelle/HOL allows us to write down formalized mathematics in a way that is very similar to what we are used to from the usual, more informal mathematics. Isabelle/HOL is also the object logic on which we will base this work, and we will refer to it as the abbreviated term HOL from now on.

Isabelle is a so-called LCF-style prover[1]. This means that it possesses a small *trusted* inference kernel which has to certify any theorem before it is regarded as proven. More precisely, the kernel provides a number of elementary functions for manipulating goal states which can be combined to build more powerful proof tactics. The important characteristic of this approach is that if the trusted kernel is implemented correctly and if Isabelle's meta logic is sound, then any theorem provable in Isabelle is also correct. (Precisely, for an *object-logic* the soundness of the logic's encoding is also necessary for soundness of the whole system.) Isabelle provides a number of automatic proof methods, including first-order proof methods, tableau provers, simplification tactics and classical reasoners. As all the proof steps they produce have to be certified by the kernel first, their implementation is not relevant for soundness.

We want to discuss Isabelle's *fastforce* method more specifically as it will be crucial for the complexity analysis in Chapter 6. On the one hand, Isabelle provides a number of automatic proof methods that are based on so-called *classical reasoning*; these are proof tactics that reason via natural deduction in combination with backtracking or other search strategies. A more general version of natural deduction is the *sequent calculus* which is also supported by Isabelle. There are also a number of classical reasoners in Isabelle that are based on this more powerful reasoning method. On the other hand, Isabelle also provides *simplification* methods which rely heavily on term rewriting (in the style of rewriting terms by using equations from left to right). The *fastforce* method combines the best of both worlds; it interweaves classical reasoning and simplification. We will later make use of the power of this combination to automatically calculate certain reduction sequences.

There are some notational specialties arising from the nature of Isabelle's meta-logic, of which we want to point out one aspect that will appear frequently in the notation of lemmas and theorems in the remainder of this work. All propositions in Isabelle can be written in a natural deduction style format as *Hereditary Harrop Formula*. This way, we can

[1] The abbreviation LCF stands for Logic of Computable Functions [9].

write a conclusion Q following from premises P_1 through P_n, where both the conclusion and its premises can contain the universally quantified variable x, as:

$$\bigwedge x.\ [\![P_1; P_2; \ldots; P_n]\!] \Longrightarrow Q$$

This notation is used for lemma or theorem statements as well as to represent the internal goal state, where subgoals of the current proof state are the premises. An empty set of subgoals ($n = 0$) represents a successful proof attempt. We will also use this notation to present lemmas and theorems throughout the text.

Originally, theorems in Isabelle were proved by applying a sequence of proof tactics (in so-called *tactic-style* proof scripts) to refine an initial goal into subgoals until all subgoals are proved, constituting a successful proof attempt. The disadvantage of this approach is that theses scripts do not really convey any meaning to the user; they can only be understood and maintained if the intermediate proof states acquired by the applications of tactics are reinspected by the user. This makes these proofs hard to maintain and makes it difficult to present the knowledge captured in them to a broader audience. Therefore, more recent developments have added the proof language *Isar* [38], which allows us to write down proofs in a structured manner as so-called Isar proof scripts or structured proofs. Isar tries to mimic the way natural language proof texts are written. Although Isar proofs scripts for Isabelle propositions are usually more verbose than their corresponding tactic-style versions, structured proofs have the advantage of being more comprehensible and maintainable. Moreover, they allow us to use a free mixture of forward and backward reasoning in a way that is more convenient than that for tactic-style proof scripts. Most of the Isabelle formalization in this work makes use of structured proofs, and in Chapter 6 we will present some actual Isar proof scripts for the complexity analysis of example programs within our framework.

We conclude this chapter with a brief discussion of a few aspects of Isabelle/HOL that are relevant for this work. Extensive documentation on Isabelle/HOL (as well as on Isar) can be found on the Isabelle webpage[2].

Types. The type system of Isabelle is based on the simply-typed lambda calculus [4]. However, HOL supports also more advanced concepts like that of *type classes*, popularized by the Haskell programming language. We will now explain some aspects of HOL's type system that are relevant for this work. There are a number of elementary datatypes such as the type *bool* for boolean values or the type *nat* for Church numerals. Type variables such as *'a* allow the definition of polymorphic datatypes such as the built-in type for modeling erroneous or missing values, *'a option*, and the type for pairs, $'a \times 'b$. The function type with argument type *'a* and result type *'b* is written as $'a \Rightarrow 'b$. The function arrow associates to the right; for instance, $'a \Rightarrow 'b \Rightarrow 'c$ is the same as $'a \Rightarrow ('b \Rightarrow 'c)$. As is familiar from mainstream functional programming languages, types are usually automatically inferred. The notation $t::'a$ can be used to explicitly annotate the term t with the type *'a*. HOL supports a mechanism for defining (polymorphic) inductive datatypes via the datatype keyword. An important feature of this mechanism is that it also allows the definition of mutually recursive datatypes. For instance, a tree consisting of alternating layers of nodes labeled with *A* and *B* could be defined in the following manner:

[2] http://isabelle.in.tum.de/

datatype
'a ATree = *ATip*
| *ANode ('a BTree) 'a ('a BTree)*
and
'a BTree = *BTip*
| *BNode ('a ATree) 'a ('a ATree)*

Functions. Isabelle allows us to define *well-defined* (partial or total) functions via multiple methods, including primitive recursion over datatypes and wellfounded recursion. Only definitions via the fun keyword are relevant for this work. The fun keyword allows us to define a total function by a set of recursive equations using arbitrary pattern matching. Isabelle tries to automatically proof the properties regarding pattern matching that are necessary for wellfoundedness and termination. This is usually sufficient for simple function definitions, including those that are found in this work. With this method, multiple mutually recursive functions for mutually recursive datatypes can also be defined using the and keyword in the definition via fun. The functions *swapa* and *swapb*, which can swap the labeling of our trees, are an instance of this.

fun *swapa* :: *'a ATree* ⇒ *'a BTree*
and *swapb* :: *'a BTree* ⇒ *'a ATree*
where
swapa ATip = *BTip* |
swapa (*ANode l x r*) = *BNode* (*swapb l*) *x* (*swapb r*) |
swapb BTip = *ATip* |
swapb (*BNode l x r*) = *ANode* (*swapa l*) *x* (*swapa r*)

Inductive Predicates and Sets. Isabelle features inductive definitions for predicates and sets via a set of rules. They specify the *least* predicate or set that only contains elements adhering to these rules. They are defined via the inductive and inductive_set keywords, respectively. These definitions also exist in the flavors of mutually inductive predicates or sets, using the and keyword to define multiple predicates or sets at the same time. Conversely, these definitions specify the *greatest* predicate or set that is consistent with the given rules. To illustrate this, we say that a non-trivial tree is leaf balanced if the children of each of its branches are either both leafs or both nodes. This property is described by the mutually inductive predicates *lba* and *lbb*:

inductive *lba* :: *'a ATree* ⇒ *bool*
and *lbb* :: *'a BTree* ⇒ *bool*
where
ABase: *lba* (*ANode BTip x BTip*) |
ACombined: ⟦*lbb l*; *lbb r*⟧ ⟹ *lba* (*ANode l x r*) |
BBase: *lbb* (*BNode ATip x ATip*) |
BCombined: ⟦*lba l*; *lba r*⟧ ⟹ *lbb* (*BNode l x r*)

We will see examples of ordinary inductive definitions of sets and predicates in the next chapter. A further example of a mutually inductive predicate can be found in Chapter 5.

All inductive definitions produce an induction rule from their defining rules which can be employed in a *rule induction*. The induction rules for *lba* and *lbb* are given as follows:

$$\begin{array}{l}
[\![lba\ x1.0;\\
\quad \bigwedge x.\ P1.0\ (ANode\ BTip\ x\ BTip);\\
\quad \bigwedge l\ r\ x.\ [\![lbb\ l;\ P2.0\ l;\ lbb\ r;\ P2.0\ r]\!] \Longrightarrow P1.0\ (ANode\ l\ x\ r);\\
\quad \bigwedge x.\ P2.0\ (BNode\ ATip\ x\ ATip);\\
\quad \bigwedge l\ r\ x.\ [\![lba\ l;\ P1.0\ l;\ lba\ r;\ P1.0\ r]\!] \Longrightarrow P2.0\ (BNode\ l\ x\ r)]\!]\\
\Longrightarrow P1.0\ x1.0
\end{array}$$

$$\begin{array}{l}
[\![lbb\ x2.0;\\
\quad \bigwedge x.\ P1.0\ (ANode\ BTip\ x\ BTip);\\
\quad \bigwedge l\ r\ x.\ [\![lbb\ l;\ P2.0\ l;\ lbb\ r;\ P2.0\ r]\!] \Longrightarrow P1.0\ (ANode\ l\ x\ r);\\
\quad \bigwedge x.\ P2.0\ (BNode\ ATip\ x\ ATip);\\
\quad \bigwedge l\ r\ x.\ [\![lba\ l;\ P1.0\ l;\ lba\ r;\ P1.0\ r]\!] \Longrightarrow P2.0\ (BNode\ l\ x\ r)]\!]\\
\Longrightarrow P2.0\ x2.0
\end{array}$$

Rules for inductive definitions can be named (as can be observed at the definition of *lba* and *lbb*), and their names can be used to refer to the appropriate induction cases in structured proofs that employ a rule induction. In this work we will also use these names in the presentation of informal proofs of this kind. To conclude this chapter we note that the HOL formalizations of all concepts introduced and results proved in the remainder of this work can be found in the abstract, which is delivered in electronic form together with this thesis.

Part II.

Complexity Analysis with Isabelle/HOL

3. A Minimalist Functional Language with Semantics

The first step to build our framework for complexity analysis is to define a language, **PMλ**[1], syntactically together with a corresponding semantics that assigns a meaning to the language. For this purpose we closely follow a set of lecture slides by Xavier Leroy for a course at Paris Diderot University [21]. The first section of this chapter will give the syntactic definition of **PMλ** and will discuss its main constructs. In the next section this will be complemented with a definition of the semantics for **PMλ** that will be primarily used in this work. Section 3.3 discusses some important properties of this semantics. As a slight digression from our main concern, Section 3.4 gives an alternative semantics and proves its equivalence to the first version. Finally, Section 3.5 presents some simple extensions to **PMλ** that can be achieved by adding mere syntax sugar but add to the practical usefulness of the language.

3.1. A Minimalist Functional Language

Before we define the abstract syntax of **PMλ**, we shall describe its basic features. Our language is fairly minimal in the sense that it supports very few concepts of functional languages that make it just expressive enough to accomplish Turing-completeness. The basic constructs we need for this are unnamed functions, function applications and a fixed-point combinator that allows us to define recursive functions. The only feature we support that does not belong to the category of mere basics is pattern matching. We add this capability to enable us to write programs in a style that is familiar to programmers who are used to real word functional languages, such as ML or Haskell, for which pattern matching is central. **PMλ** does not provide elementary datatypes (such as integers, floats or characters) but rather has a simple mechanism for constructing nested data values that can be pattern matched against. Moreover, **PMλ** does not internalize a type system in order to keep the semantics and the definition as simple as possible, making the whole language easier to work with. We define the abstract syntax of **PMλ** through the mutually recursive HOL datatypes *expr* and *PatMat*, representing regular language expressions and patterns, respectively.

Definition 3.1.
datatype
expr = *Fn string expr*
| *V string*
| *Constr string (expr list)*

[1] **PMλ**– Pattern-matched λ-calculus.

| *Match expr PatMat*
| *App expr expr*
| *Fix string expr*

and

PatMat = *Pat string (string list) expr*
| *Or PatMat PatMat*

To be able to write programs in a more readable format we add concrete Isabelle infix syntax for the constructors of *expr* and *PatMat*. This way, we can write the term *Fix f e* as $\mu\, f\ .\ e$, for instance. The remaining syntax abbreviations are displayed in Figure 3.1. Throughout the text, a, b, c, e, f and t (as well as e', e'' etc.) will commonly refer to expressions of type *expr*, while p, ps and q will refer to patterns of type *PatMat*. We give a short characterization for each of the language constructs:

- *FN* $x \Rightarrow e$ is an unnamed function with an argument of name x and function body e.
- *V* x is a free variable with name x used to reference the name of a recursive function or the argument of an unnamed function.
- $S \gg xs$ constructs a composite datatype from a constructor name S and a list of arguments xs.
- *MATCH e WITH p* matches an expression e against a pattern p.
- $a \,\$\, b$ denotes the application of expression a to expression b.
- $\mu\, f\ .\ e$ is a recursive function that can reference itself by the name f in the function body e. This is an internalized version of the fixed-point combinator that is known from the λ-calculus.
- $S \gg xs \Rightarrow e$ defines a pattern to bind a constructor value's arguments with constructor name S in-order to the names given by xs in the body expression e.
- $p \parallel ps$ combines a pattern p with a number of patterns ps that can be matched against an expression e if e does not match p. This operator is right-associative and patterns in **PMλ** are always matched from left to right. Patterns are expected to be nested only on the right-hand side.

At the end of this section we note that our language does not distinguish between program – or, say, function definitions – and expressions for the sake of minimality. Hence, we mix these terms from this point on; specifically when we use the term program, we in fact mean the term expression.

3.2. A Small-Step Semantics

Now that we have defined our language in a syntactic way, we want to define what programs in this language *mean* by defining a corresponding semantics. For this purpose many approaches have been investigated by researchers in the field of programming languages. Important approaches include operational semantics that try to capture the notion

$$
\begin{array}{ll}
FN\ x \Rightarrow e & \equiv\ Fn\ x\ e \\
S \gg xs & \equiv\ Constr\ S\ xs \\
MATCH\ e\ WITH\ p & \equiv\ Match\ e\ p \\
a\ \$\ b & \equiv\ App\ a\ b \\
\mu f\ .\ e & \equiv\ Fix\ f\ e \\
S \gg xs \Rightarrow e & \equiv\ Pat\ S\ xs\ e \\
p \parallel q & \equiv\ Or\ p\ q
\end{array}
$$

Figure 3.1.: Syntax abbreviations for the *expr* and *PatMat* datatypes

of how a program executes, denotational semantics that describe the meaning of a program with mathematical structures, and collecting semantics that describe the set of states a program flow can have reached at any point in the program execution[2]. We choose an operational semantics because, for the analysis of the runtime complexity of programs, we are concerned with the execution flow of programs rather than a program's mathematical notion or its possible execution states.

For operational semantics one can distinguish between the so-called big-step (or natural) and small-step semantics. The former is a relation between an initial program (potentially with an initial state) and its final execution result. The relation tells us how the execution of a program works step-by-step, but the relation looks as if the initial program would directly take just one big step to its execution result. A small-step semantics, on the other hand, is a relation between a start expression and a successor expression that can be reached by taking one atomic execution step from the start expression. Both types can potentially carry a corresponding state. Here, we primarily work with a small-step operational semantics since it quite directly yields a notion of the runtime complexity of programs, as we will see in Chapter 4. We nevertheless also present a big-step semantics for our language and prove its equivalence to the small-step semantics to provide some evidence that our intuition about program execution, which is captured in the small-step semantics, is the right one.

Before we give a definition of our small-step semantics, we describe some basic properties that it will assign to our programming language. Firstly, the reduction strategy has a right-to-left order, which means that we always have to reduce the right-hand side of an application to a value before we can reduce the left-hand side of the application. Secondly, we enforce a call-by-value execution scheme: the argument of a function must be fully reduced to a value before β-reduction can take place. Thirdly, functions in **PMλ** are first order, i.e. they are not first-class citizens in the sense that they cannot be passed as an argument to another function. This last property poses a major limitation to the expressiveness of **PMλ** compared to real-world functional languages but will facilitate the complexity analysis. Of course, the simple programs analyzed in this work are not affected by this limitation.

Precisely, the small-step semantics is defined via an HOL inductive set *small_step* that contains pairs of expressions together with a predicate *small_step_rel* that describes which pairs of expressions belong to the set *small_step*. The intended meaning of *small_step_rel a*

[2]For a more detailed explanation of these different styles of semantics refer to [25].

b is that expression a transforms into expression b in one execution step. We use Isabelle syntax sugar to abbreviate *small_step a b* as $a \to b$.

Definition 3.2.
inductive-set
small_step :: $(expr * expr)$ *set*
and
small_step_rel :: $[expr, expr] \Rightarrow bool$ (**infix** $\to$ 55)
where
$X \to Y \equiv (X, Y) \in small\_step$ |

Beta: $is\_cval\ v \Longrightarrow (FN\ x \Rightarrow c)\ \$\ v \to subst\ c\ x\ v$ |
RecBind: $(FN\ f \Rightarrow c)\ \$\ (\mu\, f.\ a) \to subst\ c\ f\ (\mu\, f.\ a)$ |
RightRed: $d \to d' \Longrightarrow c\ \$\ d \to c\ \$\ d'$ |
LeftRed: $[\![is\_cval\ v; c \to c']\!] \Longrightarrow c\ \$\ v \to c'\ \$\ v$ |

Rec: $is\_cval\ v \Longrightarrow (\mu\, f.\ FN\ x \Rightarrow c)\ \$\ v \to subst\ (subst\ c\ x\ v)\ f\ (\mu\, f.\ FN\ x \Rightarrow c)$ |

MatchTrueS: $[\![S = S'; is\_cval\ (S \gg cs); length\ xs = length\ cs]\!] \Longrightarrow$
$MATCH\ S \gg cs\ WITH\ S' \gg xs \Rightarrow c \to fold\ (\lambda\ (x, d)\ c.\ subst\ c\ x\ d)\ (zip\ xs\ cs)\ c$ |
MatchTrueM: $[\![S = S'; is\_cval\ (S \gg cs); length\ xs = length\ cs]\!] \Longrightarrow$
$MATCH\ S \gg cs\ WITH\ S' \gg xs \Rightarrow c \parallel ps \to fold\ (\lambda\ (x, d)\ c.\ subst\ c\ x\ d)\ (zip\ xs\ cs)\ c$ |
MatchRed: $c \to c' \Longrightarrow MATCH\ c\ WITH\ p \to MATCH\ c'\ WITH\ p$ |
MatchFalse: $[\![S \neq S'; is\_cval\ (S \gg cs)]\!] \Longrightarrow$
$MATCH\ S \gg cs\ WITH\ S' \gg xs \Rightarrow c \parallel ps \to MATCH\ S \gg cs\ WITH\ ps$ |

ConstrLeftRed: $[\![c \to c'; is\_cval\ (S \gg vs)]\!] \Longrightarrow S \gg c{\cdot}vs \to S \gg c'{\cdot}vs$ |
ConstrRightRed: $S \gg cs \to S \gg cs' \Longrightarrow S \gg c{\cdot}cs \to S \gg c{\cdot}cs'$

The definition utilizes another inductively defined predicate, *is_cval*, which describes the set of valid *constructor values*, that is, the set of expressions that are constructed only from nested applications of the abstract syntax constructor *Constr*. The names v and v' will continuously refer to constructor values throughout the text.

Definition 3.3.
inductive *is_cval* :: $expr \Rightarrow bool$
where
Constr: $(\forall\, x \in set\ xs.\ is\_cval\ x) \Longrightarrow is\_cval\ (Constr\ s\ xs)$

Furthermore, the definition of the predicate *small_step_rel* employs a function *subst* that performs capture avoiding substitution, where *subst t x t'* replaces the term $V\ x$ in term t with the term t' at free occurrences of x, i.e. at places where x is not bound by an enclosing anonymous function, pattern or fixed-point operator. As the abstract syntax of our language is defined through a mutually recursive HOL datatype, the HOL definition of capture avoiding substitution employs two mutually recursive functions, *subst* for the type *expr* and *subst_pat* for the type *PatMat* – a pattern frequently encountered in the remaining part of this work.

Definition 3.4.
fun *subst*::*expr* ⇒ *string* ⇒ *expr* ⇒ *expr*
and *subst_pat*::*PatMat* ⇒ *string* ⇒ *expr* ⇒ *PatMat* **where**

subst (*V x*) *y t* = (*if x* = *y then t else V x*) |
subst (*FN x* ⇒ *a*) *y t* = *FN x* ⇒ (*if x* = *y then a else subst a y t*) |
subst (*μ x*. *a*) *y t* = *μ x*. (*if x* = *y then a else subst a y t*) |
subst (*S* ≫ *cs*) *y t* = *S* ≫ *map* (*λu*. *subst u y t*) *cs* |
subst (*MATCH e WITH ps*) *y t* = *MATCH subst e y t WITH subst_pat ps y t* |
subst (*a* $ *b*) *y t* = (*subst a y t*) $ (*subst b y t*) |

subst_pat (*S* ≫ *xs* ⇒ *e*) *y t* = *S* ≫ *xs* ⇒ (*if* (*y* ∈ *set xs*) *then e else subst e y t*) |
subst_pat (*p* || *ps*) *y t* = *subst_pat p y t* || *subst_pat ps y t*

We briefly explain the defining rules of the small-step semantics. Rule *Beta* describes usual β-reduction with the notable peculiarity that it enforces the first-order property of our language as well as the call-by-value execution scheme by demanding a constructor value for the right hand side of the function application. The role of rule *RecBind* may not be clear at first sight. It allow us to bind a recursive function defined via the fixed-point combinator to a name in some command, thus enabling us to work with a construct similar to the *letrec*-construct found in functional languages like Scheme or Caml. In fact, we can employ some syntax sugar to extend our language with a *letrec*-construct of our own:

LETREC f = *e IN b* ≡ (*FN f* ⇒ *b*) $ (*μ f* . *e*)

We note that if the rule *Beta* would not specifically enforce the first order property of our language, it would already subsume rule *RecBind*. Rules *RightRed* and *LeftRed* define the right-to-left execution order of our language for applications. These rules conclude the rule set that could describe an usual untyped λ-calculus with an outermost-first reduction, call-by-value execution scheme.

On top we add a fixed-point combinator that is internalized in the language for two reasons. On the one hand, the first order property of the language prohibits us to use a fixed-point combinator defined through unnamed functions, hence its internalization is necessary for Turing-completeness. On the other hand, it resembles the way one would rewrite the application of a value to a recursive function by hand, i.e. by substituting the value at free occurrences of some variable that is bound by the function, as well as substituting the recursive function's definition at places where it references itself. This behavior is captured by rule *Rec*.

The second rule set defines the behavior of the pattern matching mechanism. Rules *MatchTrueS* and *MatchTrueM* describe the cases of a successful pattern match for a single and multiple patterns to match against, respectively; rule *MatchFalse* handles the case of a pattern match against an inadequate pattern where more patterns are left to try. All of the rules require the matched value to be a fully reduced constructor value. A usual *left fold*[3] instantiated on the function for capture-avoiding substitution is used to bind the pattern's variables from left to right in the case of a successful pattern match. This means that in the

[3]The type of the *fold* function in HOL is (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a list* ⇒ *'b* ⇒ *'b*.

case, that one variable name appears multiple times in a pattern, it is bound to the value corresponding to its rightmost occurrence. This behavior is different from what would be expected from a real world functional language as those usually would not allow a valid program to contain multiple occurrences of one variable in the same pattern. The last two rules describe the reduction under the value constructor *Constr*, which, analogously to the reduction of applications, follows a right-to-left order.

We conclude this section with a discussion of some popular alternatives that could have been chosen for the definition of a small-step operational semantics and the reasons that induced us to step back from them in favor of the chosen approach. A first striking feature of our semantics is that it doesn't make use of an execution state or some execution context, but rather employs textual substitution. This is unnatural in the sense that a common compiler or interpreter could not rely on textual substitution to resemble basic behavior of β-reductions or recursive function applications if these operations are sought to be executed efficiently. A semantics using some execution environment that stores intermediate variable bindings – for recursive function variable names introduced through a *letrec*-construct as well as variables introduced through unnamed functions – would describe program execution more realistically. In such an approach free variable bindings from the environment could be substituted when they are needed, i.e. when an occurrence of a term of the form $V\ x$, where x is one of the names bound in the environment, is to be reduced. We nevertheless rely on textual substitution for it has some advantages, as initial experimentation has shown: it allows us to define an equivalent big-step semantics, yields a more compact rule set, and probably more closely models the intuition one has in in mind when thinking about program reductions.

Yet another alternative that is frequently used for the definition of operational programming language semantics, which we have decided not to use, are Felleisen Hieb-style reduction contexts [6]. These work with a so-called *fill-in-a-hole* grammar that defines evaluation contexts, which are terms where ordinary subterms are replaced by special ones, called *holes*, which define the place of the next reduction step. Owens [29] claims that such an approach could yield a more compact formalization, but that it would be less accessible for automatic reasoning with proof assistants in HOL. Specifically, we argue that a semantics only defined through an inductive relation should be more convenient to reason about with HOL proof assistants, as inductive relations are widely used in HOL formalizations. In contrast, reasoning about the hole filling process that appears along the way of a program evaluation would supposedly require more intensive mechanical proof work.

3.3. Important Properties of the Small-Step Semantics

In this section we want to discuss some properties of the small-step semantics that will prove to be useful for the complexity analysis of programs. All of the proofs mentioned in this section are automatic, using adequate Isabelle proof methods. We first note that the language fulfills the so-called property of weak reduction: we cannot reduce the body of an anonymous function before getting rid of the function abstraction by performing a β-reduction. This is proved by a simple case analysis on the rules constituting the *small_step*. Formally, one can show:

Lemma 3.5 (Weak reduction). $\nexists e'.\ FN\ x \Rightarrow e \rightarrow e'$

Moreover, using rule induction on the *is_cval* relation, we can prove that values never reduce under a small-step:

Lemma 3.6. *is_cval* $v \Longrightarrow \nexists e.\ v \to e$

Taking these two properties together one can prove – employing rule induction on the semantics' rules – the important property of determinism of the small-step semantics: if we can take a reduction step from an expression e to an expression e' and another reduction step from e to an expression e'', it follows that e' has to be equal to e''. Formally:

Lemma 3.7 (Determinism of the small-step semantics)**.**
$[\![e \to e'; e \to e'']\!] \Longrightarrow e'' = e'$

3.4. A Big-Step Semantics

As a slight digression from the main theme of this work, we present in addition to our small-step semantics also a big-step semantics and prove the equivalence of both. The big-step semantics is mainly a byproduct of the author's efforts to develop the small-step semantics. In order to find some proof that the small-step semantics captures the correct intuition, the easier comprehensible big-step semantics were defined and shown to be equivalent to the small-step version. Analogous to the definition of the small-step semantics, the big-step semantics is an inductively defined relation, *big_step*, that relates a start expression a with some result expression b, denoted as $a \Rightarrow b$. Its definition is given as follows:

Definition 3.8.
inductive
big_step :: *expr* ⇒ *expr* ⇒ *bool* (**infix** ⇒ 55)
where
Val: *is_cval v* ⟹ *v* ⇒ *v* |
Fn: (*FN x* ⇒ *c*) ⇒ (*FN x* ⇒ *c*) |
RecSelf: μ *f*. *c* ⇒ μ *f*. *c* |
Beta: ⟦*a* ⇒ *FN x* ⇒ *c*; *b* ⇒ *v'*; *is_cval v'*; *subst c x v'* ⇒ *v*; *is_val v*⟧
⟹ *a* $ *b* ⇒ *v* |
Rec: ⟦*a* ⇒ μ *f*. *FN x* ⇒ *c*; *b* ⇒ *v'*; *is_cval v'*; *subst* (*subst c x v'*) *f* (μ *f*. (*FN x* ⇒ *c*)) ⇒ *v*;
is_val v⟧ ⟹ *a* $ *b* ⇒ *v* |
RecBind: ⟦*b* ⇒ μ *f*. *d*; *subst c f* (μ *f*. *d*) ⇒ *v*; *is_val v*⟧
⟹ (*FN f* ⇒ *c*) $ *b* ⇒ *v* |
MatchTrueS: ⟦*S* = *S'*; *c* ⇒ *S* ≫ *vs*; *is_cval* (*S* ≫ *vs*); *length xs* = *length vs*;
fold (λ (*x*, *d*) *c*. *subst c x d*) (*zip xs vs*) *d* ⇒ *v*; *is_val v*⟧ ⟹
MATCH c WITH S' ≫ *xs* ⇒ *d* ⇒ *v* |
MatchTrueM: ⟦*S* = *S'*; *c* ⇒ *S* ≫ *vs*; *is_cval* (*S* ≫ *vs*); *length xs* = *length vs*;
fold (λ (*x*, *d*) *c*. *subst c x d*) (*zip xs vs*) *d* ⇒ *v*; *is_val v*⟧ ⟹
MATCH c WITH S' ≫ *xs* ⇒ *d* || *ps* ⇒ *v* |
MatchFalse: ⟦*S* ≠ *S'*; *c* ⇒ *S* ≫ *vs*; *is_cval* (*S* ≫ *vs*);
MATCH S ≫ *vs WITH Ps* ⇒ *v*; *is_val v*⟧ ⟹
MATCH c WITH S'≫ *xs* ⇒ *d* || *Ps* ⇒ *v* |

ConstrRed: $[\![c \Rightarrow v; \mathit{is\_val}\ v; S \gg cs \Rightarrow S \gg vs; \mathit{is\_cval}\ (S \gg vs)]\!]$
$\Longrightarrow S \gg (c{\cdot}cs) \Rightarrow S \gg (v{\cdot}vs)$

The definition of the big-step semantics makes use of a new predicate for expressions that describes those types of expressions that would commonly be referred to as *values* in programming language research: constructor values, function abstractions, and applications of the fixed-point combinator. Henceforth, we will call all expressions that fulfill the *is_val* predicate values.

Definition 3.9.
inductive *is_val* :: *expr* $\Rightarrow$ *bool*
where
is_val $(\mu f\ .\ c)$ |
is_val $(FN\ \_ \Rightarrow \_)$ |
is_cval $x \Longrightarrow$ *is_val* x

By comparison with the small-step semantics most of the rules should be self-explanatory. We want to point out that the first three rules entail that we can take a big-step from any value (as per the *is_val* predicate) to itself. This is crucial for the equivalence with the big-step semantics as we will see below. A second peculiarity is that we always require the reduction results of big-steps to be values.

Precisely, the presented big-step semantics cannot be shown to be equivalent to the small-step semantics for every reduction. As the big-step semantics only takes *big* steps to some value, we can only show that reductions of the big-step and the small-step semantics are equivalent if they both end in a value. That is, we want to show that we can take a big-step from an expression e to a value v if and only if we can take a sequence of small-steps to get from e to v. This simplifies the equivalence proof, but the equivalence should still hold if the requirement was weakened to require any reduction result to be an expression that cannot be reduced any further, a *normal form*[4]. In order to formalize the notion of a reduction sequence with respect to the small-step semantics, we apply the Kleene star on the small-step relation. The Kleene star is predefined in HOL (more precisely in the subpackage *IMP*) as follows:

Definition 3.10.
inductive
star :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
for *r* **where**
refl: *star r x x* |
step: *r x y* $\Longrightarrow$ *star r y z* $\Longrightarrow$ *star r x z*

Now we can denote that we can take some number of small-steps to get from the expression x to expression y as $x \rightarrow^* y$, using the following definition:

Definition 3.11.
abbreviation
small_steps :: *expr* $\Rightarrow$ *expr* $\Rightarrow$ *bool*
where $x \rightarrow* y \equiv$ *star small_step_rel x y*

[4]For a discussion of different reduction strategies and normal forms for the untyped λ-calculus, see Chapter 5 of Benjamin C Pierce's *TAPL* [31].

This allows us to state the final equivalence theorem, which we want to arrive at, in a formal way:

Theorem 3.12. *is_val v* $\Longrightarrow$ *e* $\Rightarrow$ *v* = *e* $\rightarrow*$ *v*

The direction from left to right does not need the reduction result to be a value and is easily shown by a rule induction over the defining rules of the big-step relation. For the direction from right to left we show a continuation lemma that states that if we can take a small-step from an expression e to an expression e' and if we can take a *big-step* from e' to some value v, then it also has to be possible to directly take a big-step from e to v.

Lemma 3.13. $[\![e \rightarrow e'; e' \Rightarrow v; \mathit{is\_val}\ v]\!] \Longrightarrow e \Rightarrow v$

This lemma can be proved by a rule induction on the small-step semantics using case analysis on the big-step semantics in some of the cases. We can lift this continuation lemma to a form where we assume that we can take a whole sequence of small-steps to get from e to e', employing a rule induction on the defining rules of the *star* relation.

Lemma 3.14. $[\![e \rightarrow e'; e' \Rightarrow v; \mathit{is\_val}\ v]\!] \Longrightarrow e \Rightarrow v$

With the knowledge that values can always be reduced to themselves via a big-step, the second direction of the equivalence theorem follows immediately.

Lemma 3.15. $[\![e \rightarrow* v; \mathit{is\_val}\ v]\!] \Longrightarrow e \Rightarrow v$

3.5. Language Extensions

We conclude this chapter by demonstrating how our language can be extended through the addition of mere syntax sugar with a number of basic language constructs that are essential to most (functional) programming languages. We already showed how to define a *letrec*-construct via a syntax abbreviation, and, just as well, we produce a *let*-syntax with an analogous construction that replaces the fixed-point combinator with an unnamed function:

LET x = e IN b $\equiv$ *(FN x* $\Rightarrow$ *b) $ e*

Furthermore, we define the truth values *true* and *false* as *''TRUE''* $\gg$ [] and *''FALSE''* $\gg$ [], respectively. We then can easily build an *if-then-else* construct via pattern matching:

IF b THEN c ELSE d $\equiv$ *MATCH b WITH ''TRUE''* $\gg$ [] $\Rightarrow$ *c* $\|$ *''FALSE''* $\gg$ [] $\Rightarrow$ *d*

Finally, we demonstrate how to construct tuples by showing how to encode pairs that then could be nested to arrive at n-ary tuples. We define pairs by means of the following syntax abbreviation:

$\langle a, b \rangle \equiv$ *''Pair''* $\gg$ $[a, b]$

The functions *Fst* and *Snd* that extract the first and second element from a pair, respectively, are easily defined via pattern matching.

Fst p $\equiv$ *MATCH p WITH ''Pair''* $\gg$ $[$*''a''*, *''b''*$]$ $\Rightarrow$ *V ''a''*

Snd p $\equiv$ *MATCH p WITH ''Pair''* $\gg$ $[$*''a''*, *''b''*$]$ $\Rightarrow$ *V ''b''*

4. Runtime Complexity of PMλ-Programs

In the last chapter, by defining an operational semantics for **PMλ**, we have laid the foundations for various types of mathematical analysis that could be performed on our language. However, to be able to reason about the runtime complexity of **PMλ**-programs in a formal way, we first have to define what we mean when we talk about complexity. Consequently, in this chapter we define a measure for the runtime complexity of programs in our language. We then present some important properties related to that complexity measure that we will use in later chapters for the complexity analysis of some sample programs.

4.1. A Complexity Definition

Before we can come up with a complexity definition for programs we need to build an intuition of the execution time of programs in our language. As we do not have a machine model such as a Turing machine, we cannot define "execution time" through the number of computational steps a machine needs to execute a program until the final result of the program has been computed. However, we can use a quite natural alternative: we simply say that the execution time of a program (i.e. an expression) is the number of reduction steps it takes in our small-step semantics to reduce the initial expression a *normal form*. (Recall that normal forms are expressions that cannot be reduced any further.)

We note that the phrase "*the* number of reduction steps" is not absolutely precise: for non-terminating programs we can never arrive at a normal form; therefore, for these, there is no such number. Nevertheless, for terminating programs there always is only one possible sequence of reductions from an initial program to a normal form, as the the small-step relation is deterministic. Thus, for terminating programs there is exactly one such number of reduction steps.

We argue that the number of reduction steps needed to arrive at a normal form is also a sensible execution time measure. Any interpreted or compiled execution of a program in our language has to simulate the basic reduction processes that are carried out by the small-step semantics. Through the usage of a runtime stack or similar construction, all of these operations can be performed with an overhead that – at least for practical considerations – can be assumed to be constant. Therefore, upper bounds for the runtime complexity of programs in our language, as given by the number of reduction steps in our small-step relation, should generally only differ by a constant factor from the runtime upper bound for realistic machine models. Of course, this statement is by no means precise for all programs or machine models, particularly in hindsight on compiler optimizations and similar peculiarities. Still, we can safely assume that this runtime measure is reasonable for most common analysis of the runtime complexity of programs.

In order to move these thoughts into the direction of a formalization, we need to formalize the notion of a reduction sequence. In contrast to Section 3.4, the Kleene star would

not be the appropriate choice here as we want to count the number of steps a reduction sequence consists of. Instead, we employ the power of the relation product of the small-step relation. The relation product over an Isabelle/HOL *inductive_set* and its power are already predefined, which is why we chose a little detour, defining the small-step relation as an *inductive_set* rather than directly as an inductive relation. The defining equations for the relation product operator OO, and its power operator ^^, together with their signatures are given as follows[1]:

Definition 4.1.
op OO :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'a \Rightarrow 'c \Rightarrow bool$
$[\![R\ a\ b; S\ b\ c]\!] \Longrightarrow (R\ OO\ S)\ a\ c$

*op ^^* :: $nat \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
$R\ \hat{}\hat{}\ 0 = HOL.eq$
$R\ \hat{}\hat{}\ (Suc\ n) = (R\ \hat{}\hat{}\ n)\ OO\ R$

Analogously to the concrete syntax introduced for one step of the small-step relation we write $a \rightarrow^n b$ for the nth power of the relation product of the small-step relation between a and b, henceforth referred to as the $n$small-step relation.

Definition 4.2.
abbreviation
small_step_n :: *expr* ⇒ *nat* ⇒ *expr* ⇒ *bool*
where $x \rightarrow^k y \equiv (small\_step\_rel\ \hat{}\hat{}\ k)\ x\ y$

Still, this observation about the execution time of programs does not alone already provide us with a complexity definition as common complexity analysis is not concerned with programs applied to concrete inputs, but rather with programs that are given a whole class of arguments, namely all arguments of a certain size. These arguments generally correlate to an infinite number of expressions in our language. Thus we first need to define a size measure for expressions in our language so that we can talk about programs with inputs of a certain size. To this end, we define an uniform size measure over our language's abstract syntax. We assign every abstract syntax constructor a size of one and add to it the sum of the sizes of its arguments. Two special cases occur: for the value constructor *Constr* we add the sum of the arguments in its argument list and for the pattern constructor *Pat* we take the length of the pattern's argument list and add one to it. The HOL definition of our size measure employs two mutually recursive functions, *size* for the type *expr* and *size_pat* for the type *PatMat*.

Definition 4.3.
fun *size* :: *expr* ⇒ *nat*
and *size_pat* :: *PatMat* ⇒ *nat*
where
size (*V* _) = *1*|
size (*Constr x xs*) = *listsum* (*map size xs*) + *1* |

[1]For the sake of brevity, the definition is slightly simplified here compared to the actual definition in Isabelle/HOL.

size (*MATCH a WITH b*) = *size a* + *size_pat b* + 1 |
size (*μ x . c*) = *size c* + 1 |
size (*FN x* ⇒ *c*) = *size c* + 1 |
size (*a* $ *b*) = *size a* + *size b* + 1 |
size_pat (*S* ≫ *xs* ⇒ *c*) = *length xs* + *size c* + 1 |
size_pat (*a* ‖ *b*) = *size_pat a* + *size_pat b* + 1

We introduce the common notation |*e*| to refer to *size e*, the size of expression *e*. [2] Before we can give our complexity definition, we first need one further tool: as we want to talk about programs with an arbitrary number of arguments, we define an HOL function *app_list* that applies any number of arguments, given as a list of expressions, to a program by recursively nesting applications around the program.

Definition 4.4.
fun *app_list* :: *expr* ⇒ *expr list* ⇒ *expr* **where**
f $$ *xs* = *foldl* (λ *e x*. *e* $ *x*) *f xs*

We add concrete Isabelle infix syntax, allowing us to write *app_list f ts* conveniently as *f* $$ *ts*. We call expressions that consist of a **PMλ**-function applied to its expected number of arguments *closed programs*.

Without further ado we give our complexity definition:

Definition 4.5.
definition
C :: *expr* ⇒ (*expr* ⇒ *bool*) *list* ⇒ *nat list* ⇒ *nat*
where
C f Ts ns =
Max ({*k*. ∃ *ts v*. *length ts* = *length ns* ∧ *length ts* = *length Ts*
∧ *valid_terms Ts ns ts*
∧ *f* $$ *ts* →^{k} *v* ∧ *is_cval v* } ∪ {0})

The term *C f Ts ns* is intended to be the worst-case execution time of the program *f* applied to a number of arguments of which the ith is an expression of the size given by the ith integer in *ns* and of the "type" described by the ith predicate in *Ts*. As a slight terminological lapse we will refer to these predicates as the arguments' types throughout the remaining part of this work. The requirements on the program inputs are captured in part by the predicate *valid_terms*, which is defined as follows:

Definition 4.6.
valid_terms Ts ns ts ≡ (∀ (*T*, *t*)∈*set* (*zip Ts ts*). *T t*) ∧ *map size_abbrev ts* = *ns*

The complexity is defined over the set of all possible numbers of reduction steps *k* for which we can find a list of reasonable inputs *ts* and a constructor value *v* such that the function *f* applied to inputs *ts* reduces to a constructor value *v* in *k* steps of the small-step relation. By "reasonable" we mean the property that the length of *ts* fits the length of *Ts* and *ns* and that *valid_terms Ts ns ts* holds. To this set we add zero to assign the default

[2] We do not introduce such an abbreviation for *size_pat* to avoid unnecessary syntax ambiguities as we would only seldom need it in the discussion that follows.

complexity zero to all parameters of C where the complexity would not exist otherwise. This would be the case, for instance, for non-terminating programs or combinations of input sizes and types where no values fulfilling the desired properties exist. With the requirement that the final result of all reductions has to be a constructor value rather than a general normal form, we restrict ourselves to error-free computations rather than solely terminating ones. We do this to keep the later complexity analysis and the breadth of this discussion compact. A generalization to general normal forms would be straightforward as we will note on occasions where this difference in the definition would be relevant. Type restrictions to the input terms are added for a similar reason – they spare us arguing that our programs show some default reduction behavior for inputs that are not of a valid type.

To conclude this section we note that two properties of a set on which the Isabelle/HOL *Max*-operator is applied, are necessary for reasoning about the Isabelle/HOL *Max* operator and therefore for reasoning about C. Firstly, the set has to be non-empty, which self-evidently is always the case here; secondly, the set has to be finite. A proof that this holds in a general case is the subject of Chapter 5.

4.2. Related Properties of the $n$small-step relation

As the power of the relation product of the small-step relation is central to the notion of execution time used here, its properties will naturally prove to be crucial for our later complexity analysis. Therefore, we portray some basic observations about the $n$small-step relation in this section. As we argued informally before, the determinism property of the small-step relation transfers to the power of its relation product, which is easily proved using an induction on the exponent of the relation product.

Lemma 4.7 (Determinism of the $n$small-step relation)**.** $[\![e \to^n e'; e \to^n e'']\!] \Longrightarrow e'' = e'$

Naturally, this implies that there is always only one "reduction path" from an initial expression to a normal form. Formally put, we can show the following properties:

Lemma 4.8 (Uniqueness of reductions ending in a normal form)**.**
$[\![c \to^k c'; c \to^l c''; \mathit{final}\ c'; \mathit{final}\ c'']\!] \Longrightarrow k = l$
$[\![c \to^k c'; c \to^l c''; \mathit{final}\ c'; \mathit{final}\ c'']\!] \Longrightarrow c' = c''$

Both properties follow easily from these more contrived lemmas:

Lemma 4.9.
$[\![e \to^l v; k < l]\!] \Longrightarrow \nexists v'.\ e \to^k v' \wedge \mathit{final}\ v'$
$[\![e \to^l v; \mathit{final}\ v; l < k]\!] \Longrightarrow \nexists v'.\ e \to^k v'$

Proof. As the proofs for both statements are completely analogous to each other for the greater part, we showcase only the second one. We assume that l reduces to some normal form v in l steps and for the sake of contradiction we also assume that e reduces to some value v' in k steps where $k > l$. From that we can find an expression e' and a natural number k' with $k = l + k'$ such that e reduces to e' in l steps, e' reduces to v in k' steps. From Lemma 4.7 it follows that e' equals v and thus is a normal form. But from $k > l$ we know $k' > 0$, which implies that we can take at least one small-step starting from e', yielding a contradiction to the fact that v is a normal form. □

Another useful property that follows directly from the determinism of the $n$small-step relation is a sort of continuation lemma. If it is possible to take n steps from expression x to arrive at some normal form v and if it is also possible to take m steps to arrive at some expression y starting from x, then it has to be possible to reduce y in $n - m$ steps to v. Formally:

Lemma 4.10 (Continuation of small-step reduction sequences)**.**
$[\![x \to^n v; x \to^m y; \mathit{final}\ v]\!] \Longrightarrow y \to^{n-m} v$

Finally, we observe that the properties of the language's reduction strategy transfer to the $n$small-step relation. For instance, we derive the following two lemmas that describe the reduction strategy for constructor values:

Lemma 4.11.
$S \gg cs \to^n S \gg vs \Longrightarrow S \gg c{\cdot}cs \to^n S \gg c{\cdot}vs$
$[\![c \to^k c'; \mathit{is\_cval}\ (S \gg vs)]\!] \Longrightarrow S \gg c{\cdot}vs \to^k S \gg c'{\cdot}vs$

4.3. Properties of the Complexity Definition

By means of the lemmas presented in the last section, we can prove some basic properties of our complexity definition that will come in handy for the later complexity analysis. To aid the following discussion, we first introduce two notational shorthands related to our complexity definition. We define as the C_{set} for parameters *f*, *Ts* and *ns* the set that the complexity is defined on for these parameters, minus the value zero[3]:

Definition 4.12.
$C\_set\ f\ Ts\ ns \equiv$
$\{k \mid \exists\, ts\ v.$
$\quad |ts| = |ns| \wedge$
$\quad |ts| = |Ts| \wedge \mathit{valid\_terms}\ Ts\ ns\ ts \wedge f\ \$\$\ ts \to^k v \wedge \mathit{is\_cval}\ v\}$

Furthermore, we say that a program *f* is reducible given types *Ts* and size bounds *ns* if its corresponding C_{set} is non-empty, i.e.:

Definition 4.13. $\mathit{reducible}\ f\ Ts\ ns \equiv C\_set\ f\ Ts\ ns \neq \emptyset$

From the complexity definition we immediately see that the complexity $C\ f\ Ts\ ns$ is contained in its corresponding C_{set} if its C_{set} is finite and $\mathit{reducible}\ f\ Ts\ ns$ holds:

Lemma 4.14. $[\![\mathit{finite}\ (C\_set\ f\ Ts\ ns); \mathit{reducible}\ f\ Ts\ ns]\!] \Longrightarrow C\ f\ Ts\ ns \in C\_set\ f\ Ts\ ns$

On the other hand, if $\mathit{reducible}\ f\ Ts\ ns$ does not hold, $C\ f\ Ts\ ns$ clearly is zero.

Lemma 4.15. $\neg\ \mathit{reducible}\ f\ Ts\ ns \Longrightarrow C\ f\ Ts\ ns = 0$

Additionally, $C\ f\ Ts\ ns$ is an upper bound for all values contained in a corresponding finite C_{set}:

[3] As a small notational slip we write C_set rather rather than C_{set} in the structured proofs. We will do the same for analogous notation appearing throughout this work.

Lemma 4.16 (Elements of a finite C_{set} are bounded by the corresponding C)**.**
$[\![\mathit{finite}\ (C\_\mathit{set}\ f\ Ts\ ns); n \in C\_\mathit{set}\ f\ Ts\ ns]\!] \Longrightarrow n \leq C\ f\ Ts\ ns$

Lastly, we observe that if any of the desired input sizes for C is zero, then $C\ f\ Ts\ ns$ is also zero:

Lemma 4.17. $[\![0 \in \mathit{set}\ ns; \mathit{finite}\ (C\_\mathit{set}\ f\ Ts\ ns); 0 < |ns|]\!] \Longrightarrow C\ f\ Ts\ ns = 0$

Proof. For our size definition, any expression in our language has a size greater than zero. Therefore, if any element of *ns* has size zero, then we cannot find predicates *Ts* and terms *ts* such that *valid_terms Ts ns ts* holds. Thus *C_set f Ts ns* is empty, which implies that *C f Ts ns* is zero. □

5. Finite Reductions for Fixed Input Sizes

In the last chapter we stated that *C_set f Ts ns* has to be finite to be able to reason about *C f Ts ns*, because we have defined *C* by means of the *Max* operator, which is only defined on finite and non-empty sets in HOL. The subject of this chapter is a result that will generally take away the burden of reasoning about this property in our later complexity analysis. We will show that *C_set f Ts ns* is finite if, for any type *T* in *Ts* and any expression *e*, the property *T e* implies that *e* is a constructor value. This follows easily from a more general result, which states that the number of possible reduction steps needed to arrive at *some* types of normal forms for any program, given only constructor values whose size is bounded by some number *n* as arguments, is finite. Formally, we will show:

Theorem 5.1.
finite $\{k \mid \exists\, ts\ v.\ |ts| = m \wedge e\ \$\$\ ts \rightarrow^k v \wedge is\_val\ v \wedge (\forall\, v{\in}set\ ts.\ is\_cval\ v \wedge |v| \leq n)\}$

The meaning of the phrase "*some* normal forms" is captured by the predicate *is_val* as introduced in Section 3.4. The important property of the relation is that all constructor values by means of the *is_cval* relation are also values by means of the *is_val* relation. The author claims that the results presented in this chapter could also be carried into a setting where we require general normal forms as reduction results rather than the specialized case of values. However, as our complexity definition is always demanding constructor values as reduction results anyway, this simplified version is sufficient for our purpose and therefore we can spare ourselves some tedious details in our proof that we would have to fiddle with if we worked in a setting with general normal forms.

5.1. Proof Idea

In order to keep the discussion of the proof clear, we will in fact proof a slightly less general version of Theorem 5.1 which is limited to programs with only one argument:

Lemma 5.2.
finite $\{k \mid \exists\, t\ v.\ e\ \$\ t \rightarrow^k v \wedge is\_val\ v \wedge is\_cval\ t \wedge |t| \leq n\}$

The proof for Theorem 5.1 is completely analogous to what we will present in the following and can be found in its formalized version in the appendix. The central idea of the proof is the observation that any program contains only a finite number of pattern matches. Therefore, for inputs of bounded size, there is only a finite number of ways in which an execution can branch off at pattern matches. Reductions cannot branch off at places where no pattern match occurs, thus we can get reduction sequences of different lengths only from pattern matches. Altogether this means that there can only be a finite number of lengths of reduction sequences for inputs of bounded size.

In order to formalize this proof idea, we will first introduce a notion of equivalence between expressions that show equivalent reduction behavior with respect to a set of pattern-matched constructor names in question (i.e. a set of constructor names ought to appear in the patterns of some program's pattern matches). We will then show a lemma that captures the idea that expressions that are regarded as equivalent in this sense really show equivalent reduction behavior, which will imply that equivalent expressions yield reduction sequences of equal length in particular. Thus, we will get a partition of our expressions into equivalence classes with regard to a set of pattern-matched constructor names where all elements in one equivalence class have reduction sequences of equal length.

Especially for any program, we will get a partition of all possible argument lists for the program w.r.t. the pattern-matched constructor names of the program. We will show that any of these equivalence classes have a common representative, which is obtained by modifying every input (i.e. expression in the argument list) in the following way: every constructor value whose head constructor name is not contained in the program's pattern-matched constructor names is replaced by a fresh dummy constructor value which remains the same for all renaming operations. As there are only finitely many pattern-matched constructor names in any program, there are also only finitely many such representatives of bounded size (as expressions never grow in size through the modifications described) and thus there are only finitely many equivalence classes w.r.t. the program's pattern-matched constructor names for inputs of bounded size. This means that there is only a finite number of possible lengths for all the reduction sequences of the program for inputs with bounded size, which constitutes the final result.

5.2. Formalization Basics

What follows in the remainder of this chapter is a description of how the formalization of the proof we just sketched is carried out. We begin by formalizing the notion of *reduction equivalence* of expressions. Reduction equivalence of expressions is defined by means of an inductive relation on pairs of expressions w.r.t. a set of strings that represents the set of pattern-matched constructor names in question. Once again, this relation consists of two mutually inductive predicates; *red_equiv* for the type *expr* and *red_equiv_p* for the type *PatMat*.

Definition 5.3.
inductive $red\_equiv :: string\ set \Rightarrow expr \Rightarrow expr \Rightarrow bool$
and $red\_equiv\_p :: string\ set \Rightarrow PatMat \Rightarrow PatMat \Rightarrow bool$
where
$red\_equiv\ A\ (V\ x)\ (V\ x)\ |$
$[\![is\_cval\ (S \gg xs);\ is\_cval\ (S' \gg xs');\ S \notin A;\ S' \notin A]\!] \Longrightarrow red\_equiv\ A\ (S \gg xs)\ (S' \gg xs')\ |$
$[\![length\ xs = length\ ys;\ \forall\,(x,y) \in set\ (zip\ xs\ ys).\ red\_equiv\ A\ x\ y]\!]$
$\Longrightarrow red\_equiv\ A\ (S \gg xs)\ (S \gg ys)\ |$
$red\_equiv\ A\ a\ b \Longrightarrow red\_equiv\ A\ (FN\ x \Rightarrow a)\ (FN\ x \Rightarrow b)\ |$
$red\_equiv\ A\ a\ b \Longrightarrow red\_equiv\ A\ (\mu\, f.\ a)\ (\mu\, f.\ b)\ |$
$[\![red\_equiv\ A\ a\ b;\ red\_equiv\ A\ e\ f]\!] \Longrightarrow red\_equiv\ A\ (a\ \$\ e)\ (b\ \$\ f)\ |$
$[\![red\_equiv\ A\ a\ b;\ red\_equiv\_p\ A\ p\ q]\!]$
$\Longrightarrow red\_equiv\ A\ (MATCH\ a\ WITH\ p)\ (MATCH\ b\ WITH\ q)\ |$

red_equiv A a b $\Longrightarrow$ *red_equiv_p A* $(S \gg xs \Rightarrow a)$ $(S \gg xs \Rightarrow b)$ |
$[\![$*red_equiv_p A p q*; *red_equiv_p A r s*$]\!] \Longrightarrow$ *red_equiv_p A* $(p \parallel r)$ $(q \parallel s)$

Most of the rules should be self-explanatory as they simply follow the syntactic structure of the expressions. Only the second rule is interesting; it encodes the idea that two constructor values are reduction equivalent w.r.t. a set A of pattern-matched constructor names if neither of them has a head constructor name that is contained in A, because then both can never match any pattern whose constructor is contained in A. We use the notation $a \approx_A b$ to denote that expression a is reduction equivalent to expression b w.r.t. set A. We state that this notion of reduction equivalence actually fulfills the defining properties of an equivalence relation – reflexivity, symmetry and transitivity. However, we will not need the latter to derive the results in the remainder of this chapter.

To aid the further course of the proof, we introduce the mutually recursive functions *cases* and *cases_p*, which collect all the constructor names of pattern matches that occur in an expression or a pattern, respectively.

Definition 5.4.
fun *cases* :: *expr* $\Rightarrow$ *string set*
and *cases_p* :: *PatMat* $\Rightarrow$ *string set* **where**
cases $(V \_) = \{\}$ |
cases $(a \$ b) =$ *cases a* $\cup$ *cases b* |
cases $(\mu \_ . e) =$ *cases e* |
cases $(MATCH\ x\ WITH\ p) =$ *cases x* $\cup$ *cases_p p* |
cases $(FN \_ \Rightarrow e) =$ *cases e* |
cases $(Constr \_ es) = (\bigcup e \in set\ es.\ cases\ e)$ |
cases_p $(p1 \parallel p2) =$ *cases_p p1* $\cup$ *cases_p p2* |
cases_p $(S \gg \_ \Rightarrow e) = \{S\} \cup$ *cases e*

Moreover, we use the mutually recursive functions *constrs* and *contrs_pat*, which collect all the constructor names appearing in applications of the value constructor *Constr*.

Definition 5.5.
fun *constrs* :: *expr* $\Rightarrow$ *string set*
and *constrs_pat* :: *PatMat* $\Rightarrow$ *string set* **where**
constrs $(V\ x) = \{\}$|
constrs $(Constr\ x\ cs) = (\bigcup x \in set\ cs.\ constrs\ x) \cup \{x\}$ |
constrs $(MATCH\ a\ WITH\ b) =$ *constrs a* $\cup$ *constrs_pat b* |
constrs $(\mu\ x\ .\ e) =$ *constrs e* |
constrs $(FN\ x \Rightarrow e) =$ *constrs e* |
constrs $(a \$ b) =$ *constrs a* $\cup$ *constrs b* |
constrs_pat $(S \gg xs \Rightarrow e) =$ *constrs e* |
constrs_pat $(a \parallel b) =$ *constrs_pat a* $\cup$ *constrs_pat b*

We observe some easy properties of the reduction equivalence relation. Firstly, by structural induction, it can be shown that constructor values are only reduction equivalent to other constructor values:

Lemma 5.6. $[\![v \approx_A v'$; *is_cval v*$]\!] \Longrightarrow$ *is_cval v'*

This can then be easily lifted to general values:

Lemma 5.7. $[\![a \approx_A b; \mathit{is\_val}\ a]\!] \Longrightarrow \mathit{is\_val}\ b$

Secondly, we note that reduction equivalence is invariant with regard to capture-avoiding substitution of free variables:

Lemma 5.8 (Reduction equivalence is invariant w.r.t. substitution)**.**
$[\![e \approx_A e'; v \approx_A v']\!] \Longrightarrow \mathit{subst}\ e\ x\ v \approx_A \mathit{subst}\ e'\ x\ v'$
$[\![\mathit{red\_equiv\_p}\ A\ p\ p'; v \approx_A v']\!] \Longrightarrow \mathit{red\_equiv\_p}\ A\ (\mathit{subst\_pat}\ p\ x\ v)\ (\mathit{subst\_pat}\ p'\ x\ v')$

The proof employs a rule induction on the induction rules for *subst* and *subst_pat* that are automatically generated by Isabelle and requires the further property that constructor values are not affected by substitution of free variables:

Lemma 5.9 (Constructor values are not affected by substitution)**.**
$\mathit{is\_cval}\ v \Longrightarrow \mathit{subst}\ v\ x\ t = v$

Naturally, this property transfers to a row of substitutions that are carried out in sequence by the application of a left fold as used by the rules for successful pattern matches of the small-step relation. This can be proved by an induction over the list structure:

Lemma 5.10.
$[\![a \approx_A b; \mathit{length}\ as = \mathit{length}\ bs; \forall\,(a, b) \in \mathit{set}\ (\mathit{zip}\ as\ bs).\ a \approx_A b]\!]$
$\Longrightarrow \mathit{fold}\ (\lambda(x, d)\ c.\ \mathit{subst}\ c\ x\ d)\ (\mathit{zip}\ xs\ as)\ a \approx_A \mathit{fold}\ (\lambda(x, d)\ e.\ \mathit{subst}\ e\ x\ d)\ (\mathit{zip}\ xs\ bs)\ b$

5.3. Reduction Equivalence

These properties already suffice as tools to prove that our notion of reduction equivalence is the right one in the sense that expressions we regard to be reduction equivalent really show equivalent behavior under reductions following the small-step semantics. In other words, reduction equivalence is invariant with respect to reductions under the small-step semantics. Formally, we want to show:

Lemma 5.11 (Reduction equivalence is invariant w.r.t. small-step semantics)**.**
$[\![e \to f; e \approx_A e'; \mathit{cases}\ e \subseteq A; \mathit{cases}\ e' \subseteq A]\!] \Longrightarrow \exists f'.\ e' \to f' \wedge f \approx_A f'$

The natural proof technique we want to use for the proof of this lemma is a rule induction on the small-step semantics. However, we cannot directly apply this proof method to the claimed property, for that would yield an induction hypothesis that is not strong enough in the cases that describe the reduction behavior of value constructor expressions. Instead, we prove the following stronger property:

Lemma 5.12.
$[\![e \to f; e \approx_A e'; \mathit{cases}\ e \subseteq A; \mathit{cases}\ e' \subseteq A]\!]$
$\Longrightarrow \exists f'.\ e' \to f' \wedge f \approx_A f'$
$\wedge\ (\forall S\ xs.\ e = S \gg xs$
$\longrightarrow (\exists xs'\ ys'.\ f = S \gg xs' \wedge f' = S \gg ys' \wedge \mathit{length}\ xs' = \mathit{length}\ ys'$
$\wedge\ (\forall\,(x, y) \in \mathit{set}\ (\mathit{zip}\ xs'\ ys').\ x \approx_A y)))$

Proof. The idea here is that from the premises *cases* $e \subseteq A$ and *cases* $e' \subseteq A$ it follows that if e and e' are constructor terms, they can only be reduction equivalent if they share the same head constructor name and all their arguments in corresponding positions are reduction equivalent. The statement we prove encodes that this property is invariant with respect to reductions under the small-step relation.

As announced, we use a rule induction over the small-step semantics. With lemmas 5.6 and 5.8 the cases for rules *Beta*, *Rec*, *RecBind*, *RightRed*, *LeftRed* and *MatchRed* follow directly.

Naturally, the cases *MatchTrueS* and *MatchTrueM* work analogously; therefore, we only showcase the argument for *MatchTrueS*. Let e be the expression *MATCH* $S \gg cs$ *WITH* $S \gg xs \Rightarrow c \approx_A e'$ for some S, cs and xs. We may assume that xs and cs are of equal length, $S \gg cs$ is a constructor value, e is equivalent to some e' and *cases* e, *cases* e' are subsets of A. From the rules of the reduction equivalence relation, we know immediately that some c and cs' with $c \approx_A c'$ and $S \gg cs \approx_A S \gg cs'$ have to exist such that e' equals *MATCH* $S \gg cs'$ *WITH* $S \gg xs \Rightarrow c'$. By a case analysis on the rules of the reduction equivalence relation, we can see that the arguments of $S \gg cs$ and $S \gg cs'$ in corresponding positions have to be pairwise reduction equivalent. This knowledge allows us to use lemma 5.10 to show *fold* $(\lambda(x, d)\ c.\ subst\ c\ x\ d)\ (zip\ xs\ cs)\ c \approx_A fold\ (\lambda(x, d)\ c.\ subst\ c\ x\ d)\ (zip\ xs\ cs')\ c'$. From lemma 5.6, $S \gg cs \approx_A S \gg cs'$ and the information that $S \gg cs$ is a constructor value, we can conclude that $S \gg cs'$ is also a constructor value. It follows that we can take a reduction step from e' to *fold* $(\lambda(x, d)\ c.\ subst\ c\ x\ d)\ (zip\ xs\ cs')\ c'$. Altogether we have shown the thesis.

For the case *MatchFalse*, we now let e be the expression *MATCH* $S \gg cs$ *WITH* $S' \gg xs \Rightarrow c \parallel ps$ with $S \neq S'$. The given premises are similar to the last case and in particular we will get $S \in A$ and that e' is equal to *MATCH* $S'' \gg cs'$ *WITH* $S' \gg xs \Rightarrow c' \parallel ps'$ for some S'', c' and ps' with $S \gg cs \approx_A S'' \gg cs'$. If we can prove that S'' is not equal to S', we know that the pattern match for e' is equally unsuccessful as for e and hence we immediately arrive at the thesis. From the premises we know $S' \in A$, so suppose $S'' \in A$. Together with $S \gg cs \approx_A S'' \gg cs'$, this implies $S = S''$ and thus we get $S \neq S''$ using the inequality $S \neq S'$.

The proofs for cases *ConstrLeftRed* and *ConstrRightRed* work in a similar fashion, but the latter is more complicated; therefore we conclude this proof by showcasing *ConstrRightRed*. Let e be the expression $S \gg c{\cdot}cs$. Our premises are that we can take a small-step from $S \gg cs$ to $S \gg cs'$, that e is reduction-equivalent to e', and that *cases* e and *cases* e' are subsets of A. Furthermore, we have an induction hypothesis corresponding to $S \gg cs$. Again, by a case analysis on the rules of the reduction equivalence relation, we can find some d and ds with $c \approx_A d$ and $S \gg cs \approx_A S \gg ds$ such that e' equals $S \gg d{\cdot}ds$. By the same argument, we know that arguments in corresponding places of $S \gg cs$ and $S \gg ds$ have to be reduction-equivalent. This knowledge allows us to discharge the induction hypothesis, which we designed specifically to be strong enough for this case: we can obtain a list ds' of the same length as ds, with $S \gg ds \rightarrow S \gg ds'$, such that corresponding arguments of $S \gg cs'$ and $S \gg ds'$ are reduction equivalent in turn. Together with the information that c and d are reduction equivalent, this directly yields the thesis.

□

Now Lemma 5.11 follows immediately and can be lifted to reduction sequences, which is easily proved by an induction over n.

Lemma 5.13 (Reduction equivalence is invariant w.r.t. small-step reduction sequences)**.**

$[\![e \to^n f; e \approx_A e'; \mathit{cases}\ e \subseteq A; \mathit{cases}\ e' \subseteq A]\!] \Longrightarrow \exists f'.\ e' \to^n f' \wedge f \approx_A f'$

5.4. Reduction Equivalence Class Representatives

The next step is now to build the right tools to be able to show that every equivalence class, as given by the reduction equivalence relation with respect to a program's pattern-matched constructor names, contains a common representative that we can obtain from other expressions in the same equivalence class by modifying them in such a manner that we replace "uninteresting" constructor subterms with some dummy constructor value. To this end, we first give the definition of two mutually recursive functions *subst_constr* and *subst_constr_p*, which carry out such a dummy constructor substitution on expressions and patterns, respectively. An invocation of the form *subst_constr e S S'* replaces any occurrence of a constructor value, which has the head constructor name *S*, with the dummy constructor $S' \gg []$. The functions' definitions follow the syntactic structure of expressions and patterns in a straightforward manner.

Definition 5.14.
fun *subst_constr* :: *expr* ⇒ *string* ⇒ *string* ⇒ *expr*
and *subst_constr_p* :: *PatMat* ⇒ *string* ⇒ *string* ⇒ *PatMat* **where**
subst_constr (*V x*) _ _ = *V x* |
subst_constr (*a* $ *b*) *c c'* = *subst_constr a c c'* $ *subst_constr b c c'* |
subst_constr (*μ x. e*) *c c'* = *μ x. subst_constr e c c'* |
subst_constr (*MATCH x WITH p*) *c c'*
= *MATCH subst_constr x c c' WITH subst_constr_p p c c'* |
subst_constr (*FN x ⇒ e*) *c c'* = *FN x ⇒ subst_constr e c c'* |
subst_constr (*Constr S es*) *c c'* =
(*if c = S ∧ is_cval* (*Constr S es*) *then Constr c'* []
else Constr S (*map* (*λe. subst_constr e c c'*) *es*)) |
subst_constr_p (*p1* || *p2*) *c c'* = (*subst_constr_p p1 c c'* || *subst_constr_p p2 c c'*) |
subst_constr_p (*S ≫ xs ⇒ e*) *c c'* = *S ≫ xs ⇒ subst_constr e c c'*

The important property of these functions is that reduction equivalence is invariant with respect to substitutions of "uninteresting" constructor names by means of these functions:

Lemma 5.15. $[\![a \approx_A b; S \notin A; S' \notin A]\!] \Longrightarrow a \approx_A \mathit{subst\_constr}\ b\ S\ S'$

Again, the proof is by induction according to the rules for *subst_constr* and *subst_constr_p*. Besides the case of the third rule, all of the cases follow easily from the following lemma, which is proved using an induction on the list structure:

Lemma 5.16. $\mathit{is\_cval}\ (S \gg xs) \Longrightarrow \mathit{is\_cval}\ (S \gg \mathit{map}\ (\lambda e.\ \mathit{subst\_constr}\ e\ S\ S')\ xs)$

The other case requires some fiddling with technicalities introduced through the usage of *zip* in the definition of the third rule of the equivalence relation. However, it does not contain any surprises and is therefore left out. We will also need a version of this lemma lifted to a sequence of reductions carried out by a left fold:

Lemma 5.17.

$[\![set\ ns \cap A = \emptyset; S' \notin A; a \approx_A b]\!] \Longrightarrow a \approx_A fold\ (\lambda S\ b.\ subst\_constr\ b\ S\ S')\ ns\ b$

Another easy property of the constructor substitution, which is automatically proved with the help of rule induction on the induction rules for *subst_constr* and *subst_constr_p*, is that the process does not change the subjected expression or pattern if the constructor name that ought to be replaced does not appear in the subjected expression or pattern:

Lemma 5.18.
$S \notin constrs\ e \Longrightarrow subst\_constr\ e\ S\ S' = e$
$S \notin constrs\_pat\ p \Longrightarrow subst\_constr\_p\ p\ S\ S' = p$

Moreover, by structural induction we prove that the substitution process does not change the pattern-matched constructor names of an expression and, again, we lift this into the context of a left fold:

Lemma 5.19.
$cases\ e = cases\ (subst\_constr\ e\ S\ S')$
$cases\ e = cases\ (fold\ (\lambda S\ e.\ subst\_constr\ e\ S\ S')\ nl\ e)$

In order to prove that the number of representatives we produce for the reduction equivalence classes is finite, we informally want to argue in the following manner: all of the representatives are constructed from a finite number of variable names, function names, and constructor names appearing in constructor and pattern-matching expressions. However, there can only be a finite number of expressions of bounded size that are constructed from such a finite set of *names*; thus the number of representatives for our equivalence classes also has to be finite. For a formal variant of this reasoning, which we will present in the next section, we introduce functions *names* and *names_p*, which collect the set of all names appearing in an expression or a pattern.

Definition 5.20.
fun *names* :: *expr* ⇒ *string set*
and *names_pat* :: *PatMat* ⇒ *string set* **where**
$names\ (V\ x) = \{x\}|$
$names\ (Constr\ x\ cs) = (\bigcup\ x \in set\ cs.\ names\ x) \cup \{x\}\ |$
$names\ (MATCH\ a\ WITH\ b) = names\ a \cup names\_pat\ b\ |$
$names\ (\mu\ x\ .\ c) = names\ c \cup \{x\}\ |$
$names\ (FN\ x \Rightarrow c) = names\ c \cup \{x\}\ |$
$names\ (a\ \$\ b) = names\ a \cup names\ b\ |$
$names\_pat\ (S \gg xs \Rightarrow c) = set\ xs \cup names\ c \cup \{S\}\ |$
$names\_pat\ (a \parallel b) = names\_pat\ a \cup names\_pat\ b$

This gives us the tools to state a lemma that describes the essence of the modifications applied in order to obtain a equivalence class's common representative from one of the class's elements:

Lemma 5.21.
$[\![set\ nl \cap constrs\ e = \{\}; is\_cval\ t]\!]$
$\Longrightarrow \exists t'.\ fold\ (\lambda S\ e.\ subst\_constr\ e\ S\ D)\ nl\ (e\ \$\ t) = e\ \$\ t'$
$\wedge\ names\ t' \subseteq names\ t - set\ nl \cup \{D\} \wedge is\_cval\ t' \wedge |t'| \leq |t|$

It tells us that if we replace in an application of some expression e to some constructor-value argument t, each constructor name from some list nl, for which none of its elements is a constructor name appearing in e, by some dummy constructor name D, we get a result of the following form:

- e is not affected by the process; therefore we arrive at an application of e to some constructor value t', which has the properties that,
- compared to t, all of the names in nl have been removed from the names of t', but the dummy constructor name D has probably been added, and
- the size of t' is lower or equal to the size of t.

This lemma follows from three more basic properties, which are lifted into the context of the left fold in the lemma, after an induction on the list structure of nl. Firstly, a lemma that describes how the *names* of a constructor value are affected by *subst_constr*, which is easily shown using an induction on the rules for the *is_cval* predicate.

Lemma 5.22. $is\_cval\ v \Longrightarrow names\ (subst\_constr\ v\ S\ S') \subseteq names\ v - \{S\} \cup \{S'\}$

Secondly, a lemma that states that expressions never grow in the process of substituting a dummy constructor, which is proven employing a structural induction on the mutually recursive datatypes involved.

Lemma 5.23 (Size monotonicity of constructor substitution)**.**
$|subst\_constr\ e\ S\ S'| \leq |e|$

And thirdly, a lemma that tells us that constructor substitution does not affect expressions and patterns, in which the targeted constructor name does not appear.

Lemma 5.24.
$S \notin constrs\ e \Longrightarrow subst\_constr\ e\ S\ S' = e$
$S \notin constrs\_pat\ p \Longrightarrow subst\_constr\_p\ p\ S\ S' = p$

The proof is by an induction on the two functions' induction rules. Together with Lemma 5.17 this will, in the proof of the final theorem, be sufficient for the reasoning about the construction process of common representatives for our reduction equivalence classes.

5.5. Putting it Together

The last part of the puzzle is to argue that the set of the equivalence classes' representatives is finite. As already mentioned, we will do this by arguing that a set of expressions of bounded size constructed from a finite number of names is finite. Formally, the main result here is:

Lemma 5.25.
$finite\ A \Longrightarrow finite\ \{e \mid |e| = m \wedge names\ e \subseteq A\} \wedge finite\ \{p \mid size\_pat\ p = m \wedge names\_pat\ p \subseteq A\}$

Proof. The proof is by a strong induction over m. The idea is to construct from the sets $\{e \mid |e| < m \wedge \mathit{names}\ e \subseteq A\}$ (henceforth named E) and $\{p \mid \mathit{size\_pat}\ p < m \wedge \mathit{names\_pat}\ p \subseteq A\}$ a number of *finite* sets – one for each constructor of the datatype *expr* – whose union covers $\{e \mid |e| = m \wedge \mathit{names}\ e \subseteq A\}$ and analogously for $\{p \mid \mathit{size\_pat}\ p = m \wedge \mathit{names\_pat}\ p \subseteq A\}$. For instance, one of the constructed sets would be

$$\{C \gg es \mid C \in A \wedge es \in \{es \mid |es| < m \wedge es \in \mathit{lists}\ E\}\}.$$

Here, *lists E* (predefined in HOL) is the infinite set of lists that only contains elements from E. Due to the way this set is constructed, the proof has to rely on the further property that a subset of *lists E*, which is restricted to lists of bounded size, is always finite – a property that is easily proved with the help of an induction over the size bound. The rest of the proof does not contain any more interesting details and is therefore skipped.

□

Now we are ready to assemble all the parts needed to prove Lemma 5.2.

Proof. Let S be the set $\{k \mid \exists\, t\ e'.\ e\ \$\ t \rightarrow^k e' \wedge \mathit{is\_val}\ e' \wedge \mathit{is\_cval}\ t \wedge |t| \leq n\}$. Additionally, let D be a dummy constructor name that is neither contained in *cases e* nor *constrs e* – we can always find such a D due to the fact that *cases e* and *constrs e* are finite sets. These facts can be easily proved by structural induction. Let N be the set *cases e* $\cup$ *constrs e* $\cup \{D\}$ and let R be the set $\{k \mid \exists\, t\ e'.\ e\ \$\ t \rightarrow^k e' \wedge \mathit{is\_val}\ e' \wedge \mathit{is\_cval}\ t \wedge |t| \leq n \wedge \mathit{names}\ t \subseteq N\}$. The latter is intended to be the set of all possible lengths of all possible reductions sequences for the representatives of the reduction equivalence classes with respect to *cases e*. As *cases e* and *constrs e* are finite, N is finite and therefore, we can argue that R is also finite; it is a direct consequence of Lemma 5.25 that the set $\{t \mid \mathit{is\_cval}\ t \wedge |t| \leq n \wedge \mathit{names}\ t \subseteq N\}$ is finite and with Lemma 4.8 (uniqueness of reduction sequences ending in a normal form) it follows that R is finite.

What is left to show is that S is a subset of R. So, suppose $k \in S$ for some arbitrary but fixed natural number k. We can find expressions t and e', such that:

- $e\ \$\ t \rightarrow^k e'$
- $\mathit{is\_val}\ e'$
- $\mathit{is\_cval}\ t$
- $|t| \leq n$

Let N' be the set *constrs* $t - (\mathit{cases}\ e \cup \mathit{constrs}\ e)$, i.e. the set of constructor names we intend to remove from t in the dummy constructor substitution process. Let nl' be a list that contains exactly the elements from N' and let t' be the term $\mathit{fold}\ (\lambda c\ e.\ \mathit{subst\_constr}\ e\ c\ D)\ nl\ (e\ \$\ t)$ – the term from which all constructor names in N' have been removed by substituting dummy constructor D. From Lemma 5.17, we know $e\ \$\ t \approx_{\mathit{cases}\ e} t'$. From Lemma 5.13 (reduction equivalence is invariant w.r.t. small-step reduction sequences) and $e\ \$\ t \rightarrow^k e'$ we then know that we can find a term f, such that $t' \rightarrow^k f$ and $e' \approx_{\mathit{cases}\ e} f$ hold. As e' is a value, the latter property tells us together with Lemma 5.7 that f is also a value. With the help of Lemma 5.21 and a bit of set arithmetic, we can find an expression t'' with the following properties:

- $t' = e \$ t''$
- *names* $t'' \subseteq N$
- *is_cval* t''
- $|t''| \leq |t|$

With $|t| \leq n$, we immediately have $|t''| \leq |t|$ and together with $t' \rightarrow^k f$ and the knowledge that f is a value, we can conclude that $k \in S$ holds. □

With the help of Theorem 5.1 we can easily arrive at our final result about the finiteness of C_{set}:

Corollary 5.26. $\forall P{\in}$*set Ts*. $\forall$ *v*. *P v* $\longrightarrow$ *is_cval v* $\Longrightarrow$ *finite* (*C_set f Ts ns*)

Proof. With Theorem 5.1 we can first conclude that the set

$$\{k \mid \exists\, ts\ v.\ |ts| = |ns| \wedge map\ size\ ts = ns \wedge e\ \$\$\ ts \rightarrow^k v \wedge is\_cval\ v \\ \wedge\ (\forall v{\in}set\ ts.\ is\_cval\ v)\}$$

is finite. By observing that *C_set f Ts ns* is a subset of this set, we arrive at the thesis. □

6. Complexity Analysis by Example

So far, we have built a framework for runtime complexity analysis of programs in our language. Based upon this framework, this chapter shows how to derive runtime complexity bounds for three concrete example programs in our language that all work on the list data structure: the concatenation of lists and two versions of list reversal – one naive variant based on list concatenation and one, more efficient, tail recursive variant.

6.1. Lists in PMλ

As all of our example programs are based on the list data structure, we first show how this structure can be implemented in our language and prove some helpful properties for the complexity analysis. We use the abbreviation

$$nil \equiv ''Nil'' \gg []$$

for the empty list and the abbreviation $x\#\#xs$ to construct a new list from a new list element x (the head of the new list) and a list xs (the tail of the new list), where $x\#\#xs$ is defined by means of

$$x \mathbin{\#\#} xs \equiv ''Cons'' \gg [x, xs].$$

In the remainder of this chapter the names xs, xs' and ys will refer to **PMλ** lists. We define an inductive relation *is_list* that describes all the expressions that we regard as lists.

Definition 6.1.
inductive *is_list*::*expr* $\Rightarrow$ *bool* **where**
Nil: *is_list nil* |
Cons: *is_cval x* $\Longrightarrow$ *is_list xs* $\Longrightarrow$ *is_list* (*x*##*xs*)

We observe that by this definition every list is a constructor value, which is easily proved using a rule induction on the *is_list* predicate. Furthermore, we show some easy properties of the reduction behavior of lists. If we can reduce a list whose head is already a normal form, then its tail also has to be reducible, which is proved easily using a case analysis on the small-step relation.

Lemma 6.2. $[\![x \mathbin{\#\#} xs \to c; \mathit{final}\ x]\!] \Longrightarrow \exists\, xs'.\ xs \to xs'$

With the help of this result and by using basic properties of the $n$small-step relation, which we proved earlier, one can show that this property transfers to any number of reduction steps.

Lemma 6.3. $[\![x \mathbin{\#\#} xs \to^k v; \mathit{final}\ x]\!] \Longrightarrow \exists\, xs'.\ xs \to^k xs'$

Proof. The proof employs an induction over k. The base case ($k = 0$) is trivial. For the induction step assume $x \#\# xs \rightarrow^{Suc\ n} v$ for some x, xs and a normal form v. Then, due to the determinism of the small-step semantics and the properties of the power of the relation product, we can find an xs' such that we can take a reduction step from xs to xs', and that $x\#\#xs'$ can be reduced to v in n steps. The latter property allows us to discharge the induction hypothesis, lending us an xs'' to which xs' can be reduced in n steps, which implies $xs \rightarrow^{Suc\ n} xs''$. From the elementary properties of the $n$small-step relation we then get $x \#\# xs \rightarrow^{Suc\ n} x \#\# xs''$, yielding the thesis. □

If we make the further assumption that the reduction result is a constructor value, we can show that the tail reduces to a constructor value as well.

Lemma 6.4. $[\![x \#\# xs \rightarrow^k v; \mathit{final}\ x; \mathit{is\_cval}\ v]\!] \Longrightarrow \exists xs'.\ xs \rightarrow^k xs' \wedge \mathit{is\_cval}\ xs'$

Proof. The proof is analogous to Lemma 6.3 using the extra argument in the last step that v is a constructor value and that v has to equal $x\#\#xs''$ due to determinism of the $n$small-step relation, which altogether implies that x and xs'' also must be constructor values. □

This concludes the discussion of the properties related to the *is_list* predicate.

6.2. Complexity Analysis for *rev*

As a first example we derive a linear runtime upper bound for a tail-recursive variant of list reversal. As it will turn out, the analysis for this version is easier than the naive, list concatenation-based version. For the proofs that appear in the remaining part of this chapter, we will also briefly mention the required Isabelle proof steps in order to demonstrate how the end user would have to use our framework. A possible definition for the tail-recursive variant of list reversal is the following:

Definition 6.5.
rev ≡
LETREC ''rev'' = FN ''acc'' ⇒ FN ''xs'' ⇒
MATCH V ''xs'' WITH
''Nil'' » [] ⇒ V ''acc'' ||
''x'' ## ''xs'' ⇒ V ''rev'' $ (V ''x'' ## V ''acc'') $ V ''xs''
IN FN ''xs'' ⇒ V ''rev'' $ nil $ V ''xs''

However, this definition is not very accessible to a formal complexity analysis. After taking two reduction steps that substitute *nil* and the argument to *rev* in the recursive function defined by the *letrec*-construct, the inner *rev* function, the remaining steps of the program evaluation will only involve pattern matching and recursive applications of the inner *rev* function. Still, in order to employ inductive arguments for the complexity analysis, we must use the recurring part of the computation, i.e. the inner *rev* function. Thus we assign this function the name *rev'* and primarily analyze *rev'*.

Definition 6.6.
rev' ≡

μ ''rev''. FN ''acc'' $\Rightarrow$ FN ''xs'' $\Rightarrow$
MATCH V ''xs'' WITH
''Nil'' $\gg$ [] $\Rightarrow$ V ''acc'' ||
''x'' ## ''xs'' $\Rightarrow$ V ''rev'' $ (V ''x'' ## V ''acc'') $ V ''xs''

At the end of this section we will show how to derive an upper bound of the runtime of *rev* from the upper bound of *rev'* in a simple manner.

From the pattern-matching portion of *rev'* it is obvious that *rev'* returns its first argument (the *accumulator*) if its second argument is the empty list. In Isabelle, this can be formally expressed using the following *schematic* lemma:

Lemma 6.7. *is_cval v* $\Longrightarrow$ *rev' $$ [v, nil]* $\rightarrow^{?k}$ *v*

Schematic lemmas can contain schematic variables, denoted by a preceding "?", for which solutions have to be calculated during the proof process. Here, after unfolding the definition of *rev'*, the necessary number of reduction steps (three) can be automatically calculated through an application of the *fastforce* method set up to use Lemma 5.9 (stating that constructor values are not affected by substitution) as an additional simplification rule, together with two basic introduction rules about the power of the relation product. We note that the concrete Isabelle proof makes use of a version of this lemma in which the nested applications of *Suc* are folded to numerals. The same holds for all other lemmas that originate from a schematic lemma and which are discussed below.

After describing the reduction behavior in the base case of *rev'*, we can employ the very same technique to prove the following schematic lemma that describes the reduction behavior of *rev'* in the case entailing a recursive application of *rev'*:

Lemma 6.8. $[\![$*is_cval a*; *is_cval x*; *is_cval xs*$]\!] \Longrightarrow$ *rev' $$ [a, x ## xs]* $\rightarrow^4$ *rev' $$ [x ## a, xs]*

We want to derive from these lemmas inequalities or equalities describing runtime upper bounds for both cases. Unfortunately, the process of deriving these (in)equalities is less automatic than the process of deriving statements about the reduction behavior of *rev'*. In order to tackle the base case, we first need to show that if a closed application of *rev'* with sensible arguments, the second of which is size one, can be reduced to a constructor value, then the second argument has to be the empty list.

Lemma 6.9.
$[\![$*rev' $$ [t1.0, t2.0]* $\rightarrow^k$ *v*; *is_cval t1.0*; $|$*t2.0*$| = 1$; *is_list t2.0*; *is_cval v*$]\!] \Longrightarrow$ *t2.0* $=$ *nil*

Proof. The proof of this lemma is based on the additional property that all constructor values of size one have the form $S \gg []$ for some constructor name S, which is proved with a bit of handiwork in Isabelle using classical contradiction and a case distinction on the inductive rules of the *is_cval* predicate. Let now S be some constructor name such that $t2$ equals $S \gg []$ and suppose $S \neq$ ''Nil'' (else $t2 = nil$ and the thesis follows immediately). Then, it can be shown that the following reduction can performed:

rev' $$ [t1, t2] $\rightarrow^3$ *MATCH S* $\gg$ [] *WITH ''x'' ## ''xs'' $\Rightarrow$ rev' $ (V ''x'' ## t1) $ V ''xs''*

This property is proved with the help of Lemma 5.9 by applying a combination of automatic Isabelle methods to overcome some technicalities (involving unfolding of numerals). An application of the *auto* proof method can prove that the result of this reduction is

a normal form, allowing us to employ the uniqueness of computation paths ending in a normal form (cf. Lemma 4.8) to derive that this expression has to be equal to v. Now, as v is assumed to be a constructor value and the reduction result is not, we have derived a contradiction. □

In order to keep the notation in the remaining discussion compact we abbreviate *C rev′ [is_cval, is_list]* as $C_{rev'}$ and *reducible rev′ [is_cval, is_list]* as $R_{rev'}$. We can now prove that the runtime of *rev′* in the base case is three:

Lemma 6.10. *R_rev′ [m, 1] ⟹ C_rev′ [m, 1] = 3*

The proof makes use of a new introduction rule for C:

Lemma 6.11.
⟦⋀*k ts v*.
⟦|*ts*| = |*ns*|; |*ts*| = |*Ts*|; ∀ (*T*, *t*)∈*set* (*zip Ts ts*). *T t*; *map size_abbrev ts* = *ns*;
f $$ *ts* →^{k} *v*; *is_cval v*⟧ ⟹ *k* ≤ *x*;
x ∈ *C_set f Ts ns*⟧ ⟹ *C f Ts ns* = *x*

Informally this lemma states that if x is an upper bound for the runtime of f on sensible input and if x belongs to the corresponding C_{set} for parameters f, Ts and ns, then x is the complexity for these parameters. The lemma can easily be derived from the following characteristic of the *Max*-operator:

⟦⋀*y*. *y* ∈ *A* ⟹ *y* ≤ *x*; *x* ∈ *A*⟧ ⟹ *Max A* = *x*

Now we are ready to prove Lemma 6.10.

Proof. Using Lemma 6.11, our first proof obligation is to show the upper bound property of the number three. From the assumption *R_rev′ [m, n]* we can assume that there is a constructor value t_1 of size m and a list t_2 of size one such that *rev′* applied to t_1 and t_2 reduces to a constructor value v in k steps. We have to show $k \leq 3$. With Lemma 6.9 we know that t_2 is the empty list and thus we have $k = 3$ using the uniqueness of computation paths ending in a normal form (cf. Lemma 4.8), Lemma 6.7 and the knowledge that constructor values are always normal forms.

Secondly, we have to prove that three is an element of *C_set rev′ [is_cval, is_list] [m, 1]*. This follows directly from Lemma 6.7 if we can find a constructor value of size m that we can give to *rev′* as its first argument. However, by assumption *R_rev′ [m, n]* we can always find such a constructor value. □

In order to prove the latter property, the concrete Isar proof script in Isabelle requires a bit of fiddling with the definition of C_{set}, mainly explicating the fact that HOL lists of length two consist of two elements. It can then be shown that three is contained in the right C_{set} with the help of Lemma 6.7 and an introduction rule for C_{set}-membership that just unwinds the definition of *valid_terms*:

Lemma 6.12.
⟦*f* $$ *ts* →^{k} *v*; |*ts*| = |*ns*|; |*ts*| = |*Ts*|; ∀ (*T*, *t*)∈*set* (*zip Ts ts*). *T t*;
map size_abbrev ts = *ns*; *is_cval v*⟧
⟹ *k* ∈ *C_set f Ts ns*

We can eliminate the assumption in Lemma 6.10 by relaxing the property to an inequality (and thus including those numbers m for which *R_rev'* $[m, n]$ does not hold):

Lemma 6.13. *C_rev'* $[m, 1] \leq 3$

Proof. The Isabelle proof is solved by the *auto* method using lemmas 6.10 and 4.15 after applying a case analysis on whether *R_rev'* $[m, n]$ holds. □

Before we derive a lemma describing a runtime upper bound for the recursion case, we lay the foundations for discharging properties about *C_rev'* $[m, n]$. With help of the *fastforce* method we can automatically prove from Corollary 5.26 (the main result of chapter 5) and the knowledge that lists are always constructor values that *C_set rev'* [*is_cval*, *is_list*] $[m, n]$ is finite.

Lemma 6.14. *finite* (*C_set rev'* [*is_cval*, *is_list*] $[m, n]$)

We now want to derive a sort of recursion equation for the complexity upper bound of *rev'* in the case that the second argument of *rev'* is greater than one:

Lemma 6.15.
$[\![1 < n; R\_rev'\ [m, n]]\!] \Longrightarrow \exists\, m'\, n'.\ n' < n \wedge C\_rev'\ [m, n] \leq 4 + C\_rev'\ [m', n'] \wedge R\_rev'\ [m', n']$

Proof. Let M be *C_rev'* $[m, n]$. From the assumptions and lemmas 4.14 (for a non-empty C_{set} the corresponding C is contained in the C_{set}) and 6.14 we can assume the existence of expressions *t1*, *t2* and v that adhere to the following properties:

- *is_cval t1*
- $|t1| = m$
- *is_list t2*
- $|t2| = n$
- *rev'* \$\$ *ts* $\rightarrow^M$ *v*
- *is_cval v*

From *is_list t2* we show the thesis by a case analysis on the *is_list* predicate. As we have postulated $n > 1$, the case of the empty list is trivial. Thus, we can assume a value x and a list *xs* where x is the head of *t2* and xs is the tail of *t2*. With the help of Lemma 6.8 we then have *rev'* \$\$ [*t1*, *t2*] $\rightarrow^4$ *rev'* \$\$ [*x* ## *t1*, *xs*]. From Lemma 4.10 (continuation of small-step reduction sequences) and the knowledge that as v is a constructor value it is also a normal form, we can then show that the last reduction result can be reduced to v in $M - 4$ steps. With the additional knowledge we have about x, xs, *t1* and v, it is obvious that $M - 4$ has to be contained in *C_set rev'* [*is_cval*, *is_list*] [|*x* ## *t1*|, |*xs*|] allowing us to derive the inequality *C_rev'* $[m, n] \leq 4 +$ *C_rev'* [|*x* ## *t1*|, |*xs*|] by means of Lemma 4.16 (elements of a finite C_{set} are bounded by its corresponding C) and the finiteness of the C_{set} for *rev'*. In the same way, we can also conclude *R_rev'* [|*x* ## *t1*|, |*xs*|]. It is obvious that $|xs| < n$, which together with the last two properties constitutes the thesis. □

Contrast this informal proof with the Isar proof script given in Figure 6.1. The overall proof structure is very similar to the structure of the informal proof, although some arguments necessitate more detailed reasoning. The process of discharging the reducibility assumption breaks down into three Isar proof steps, because the assumption first has to be discharged to obtain a list ts of length two, which then allows us to obtain its two list elements and to transfer the properties from ts to its elements. The following lemma, named *list_length_2_D* in the Isar proof script, states that any list of length two consists of two elements:

Lemma 6.16. $|xs| = 2 \Longrightarrow \exists\, a\ b.\ xs = [a, b]$

We note that this argument can be carried over completely analogously to programs with any number of arguments, provided that first a lemma analogous to Lemma 6.16 is proven for the correct list size. The only further step that is not in a natural analogy to the informal reasoning is the argument that $M - 4$ is contained in *C_set rev'* [*is_cval*, *is_list*] [|*x ## t1*|, |*xs*|]. We first need to explicitly apply Lemma 6.12 as an introduction rule and then issue Isabelle to solve the subgoal that demands reducibility by assumption, before we can let the *auto* method take care of the rest. However, this combination of proof methods can simply be duplicated for a similar proof step in other complexity analyses.

Analogous to the approach for the base case, we now eliminate the reducibility assumption by weakening the statement to contain an inequality.

Lemma 6.17. $1 < n \Longrightarrow \exists\, m'\, n'.\ n' < n \wedge C\_rev'\,[m, n] \leq 4 + C\_rev'\,[m', n']$

Again, after a case analysis on whether *R_rev'* [*m*, *n*] holds, the proof obligations can be solved by applications of automatic methods using lemmas 6.15 and 6.13. Now we are ready to resolve the recurrence to obtain an upper bound for *C_rev'* [*m*, *n*]. As the reduction in Lemma 6.17 needs four steps and the reduction in the base case needs only three steps, it easy to guess that the following upper bound statement has to hold:

Theorem 6.18. $C\_rev'\,[n, m] \leq 4 + C\_rev'\,[n', m']$

Proof. The proof is by a strong induction over m. We have two base cases. If m is zero, then *C_rev'* [*n*, *m*] is zero by Lemma 4.17 and if m is one, then Lemma 6.13 directly yields the thesis. For the induction step we can assume the induction hypothesis $\forall\, m'\, n.\ m' < m \Longrightarrow C\_rev'\,[n, m'] \leq 4 * m'$. From Lemma 6.17 we can find natural numbers n' and m' such that $m' < m$ and $C\_rev'\,[n, m] \leq 4 + C\_rev'\,[n', m']$ hold. Together with the induction hypothesis, this yields the thesis. □

Once again, the Isar proof script for Theorem 6.18 is depicted in Figure 6.2 for comparison. The only technical difference that can be found here is that the induction rule used does not directly yield the two base cases, but we have to handle those by applying a case distinction on m first.

For completeness, we demonstrate how to derive an upper bound for the runtime complexity of *rev* from Theorem 6.18. As we can always reduce a closed application of *rev* to an application of *rev'* in the case that we provide sensible arguments to *rev*, we can see that we only have to add two to the upper bound for *rev'* to get an upper bound *rev*. The relation between *rev* and *rev'* in terms of reduction is given as follows:

```
lemma C_step_rev':
 n > 1 ⟹ R_rev' [m,n] ⟹
 ∃ m' n'. n' < n ∧ C_rev' [m,n] ≤ 4 + C_rev' [m',n'] ∧ R_rev' [m',n']
proof −
 assume n > 1 R_rev' [m,n]
 def M ≡ C_rev' [m,n]
 obtain ts :: expr list and v where ts:
  length ts = 2   rev' $$ ts →^M v   is_cval v   valid_terms [is_cval,is_list] [m,n] ts
 using fin_red_C_elem_C_set[OF C_set_finite' ‹R_rev' [m,n]›]
 by (auto simp: eval_nat_numeral M_def)
 then obtain t1 t2 where ts2:ts = [t1,t2] by (blast dest: list_length_2_D)
 with ts have t1t2: is_cval t1   is_list t2   |t1| = m   |t2| = n
 by (auto simp: valid_terms_def)+
 from ‹is_list t2› show ?thesis
 proof (cases rule: is_list.cases)
  case Nil thus ?thesis using ‹n > 1› t1t2 by auto
 next
  case (Cons x xs)
  with rev'_red_cons[OF ‹is_cval t1› ] have rev' $$ [t1,t2] →^4 rev' $$ [x ## t1,xs] by force
  with n_steps_sub_final[OF _ this cval_final] ts ts2 have rev' $$ [x ## t1,xs] →^(M − 4) v by auto
  with ‹is_cval x› ‹is_list xs› ‹is_cval t1› ‹|t1| = m› ‹is_cval v›
  have ∗:M − 4 ∈ C_set rev' [is_cval,is_list] [ |x##t1|, |xs| ]
   apply (intro C_set_I)
   apply assumption
   apply auto
  done
  from C_ge[OF C_set_finite' this] have C_rev' [m, n] ≤ 4 + C_rev' [ |x##t1|, |xs| ]
  by (auto simp: C_def M_def)
  moreover have 0 < |xs|   |xs| < n using Cons(1) ‹|t2| = n› by auto
  moreover from ∗ have R_rev' [ |x##t1|, |xs| ] unfolding reducible_def by blast
  ultimately show ?thesis by auto
 qed
qed
```

Figure 6.1.: The Isar proof script for Lemma 6.15

```
lemma rev'_linear_complexity:
  C_rev' [n, m] ≤ 4 * m
proof (induction m arbitrary: n rule: less_induct)
  case (less m)
  show ?case
  proof (cases m ≤ 1)
    case True
    thus ?thesis using complexity_0[OF _ C_set_finite'] C_rev'_base_case[of n] by (cases m) auto
  next
    case False
    with C_step_le[of m n] less obtain n' m' where *:
    m' < m   C_rev' [n, m] ≤ 4 + C_rev' [n', m'] by auto
    with less.IH[of m' n'] show ?thesis by auto
  qed
qed
```

Figure 6.2.: The Isar proof script for Theorem 6.18

Lemma 6.19. *is_cval v* $\Longrightarrow$ *Reverse.rev* \$\$ [*v*] $\rightarrow^2$ *rev'* \$\$ [*nil*, *v*]

Similar to the approach taken above, we first show the upper bound under the assumption *reducible rev* [*is_list*] [*m*].

Lemma 6.20. *reducible rev* [*is_list*] [*m*] $\Longrightarrow$ *C rev* [*is_list*] [*m*] $\leq 2 + 4 * m$

Proof. Analogous to the proof of Lemma 6.15, we show $M - 2 \in$ *C_set rev'* [*is_cval*, *is_list*] [*1*, *m*] with the help of Lemma 6.19. From the knowledge that C is always an upper bound for the elements in a corresponding finite C_{set} (Lemma 4.16), we find that *C-rev'-1-m* is an upper bound for $M - 2$, thus yielding the thesis by means of Lemma 6.14. □

The similarity between the informal proofs of Lemma 6.20 and Lemma 6.15 carries over to their Isar proof scripts just as well, as can be seen by comparison with Figure 6.3.

With this result and Lemma 4.15, the proof of the final theorem describing the complexity of *rev* is now automatic after a case distinction on whether *reducible rev* [*is_list*] [*m*] holds.

Theorem 6.21. *C rev* [*is_list*] [*m*] $\leq 2 + 4 * m$

6.3. Complexity Analysis for *concat*

A possible definition of the common list concatenation program in **PMλ** is the following:

Definition 6.22.

```
concat ≡
  μ ''concat''. FN ''xs'' ⇒ FN ''ys'' ⇒
    MATCH V ''xs'' WITH
      ''Nil'' ≫ [] ⇒ V ''ys'' ||
      ''x'' ## ''xs'' ⇒ V ''x'' ## V ''concat'' $ V ''xs'' $ V ''ys''
```

lemma *rev_linear_complexity1*:
reducible rev [*is_list*] [*m*] $\Longrightarrow$ *C rev* [*is_list*] [*m*] $\leq 2 + 4 * m$
proof −
assume *A*: *reducible rev* [*is_list*] [*m*]
def $M \equiv$ *C rev* [*is_list*] [*m*]
obtain *ts* :: *expr list* **and** *v* **where** *ts*:
length ts = 1 *rev* $$ *ts* $\rightarrow^M$ *v* *is_cval v* *valid_terms* [*is_list*] [*m*] *ts*
by (*auto intro*: *fin_red_complexity_D*[*OF C_set_finite''* ‹*reducible rev* [*is_list*] [*m*]›] *simp*: *M_def*)
then obtain *t* **where** *ts2*: *ts* = [*t*] **by** (*blast dest*: *list_length_1_D*)
with *ts* **have** *t*: *is_list t* |*t*| = *m* **by** (*force simp*: *valid_terms_def*)+
with *rev_rev'_red ts2* **have** *red*: *rev* $$ *ts* $\rightarrow^2$ *rev'* $$ [*nil*, *t*] **by** *auto*
from *t* ‹*is_cval v*› *n_steps_sub_final*[*OF ts*(2) *red cval_final*]
have ∗:$M - 2 \in$ *C_set rev'* [*is_cval*, *is_list*] [*1*,*m*]
apply (*intro C_set_I*)
apply *assumption*
apply *auto*
done
from *C_ge*[*OF C_set_finite' this*] **have** $M - 2 \leq$ *C_rev'* [*1*, *m*] **.**
thus *?thesis* **using** *rev'_linear_complexity*[*of 1 m*] **unfolding** *M_def* **by** *linarith*
qed

Figure 6.3.: The Isar proof script corresponding to Lemma 6.20

We work directly on the interesting part of the function and abstain from wrapping a *letrec*-construct around it, as in the preceding section we have already shown how to derive from a complexity upper bound for the latter version a complexity upper bound for the former version. The helpful quality of our complexity analysis for *concat* is that it is completely analogous to the complexity analysis for *rev'* to the extent that the Isar proof scripts are just duplicates of each other for the most part. Thus, we will skip proofs without further notice in the following discussion if they transfer directly from *rev'* to *concat* aside from most obvious adjustments (such as renaming or argument reordering).

The reduction behavior for the cases of an empty and a composite list are described by lemmas 6.23 and 6.24.

Lemma 6.23. *is_cval ys* $\Longrightarrow$ *concat* $$ [*nil*, *ys*] $\rightarrow^3$ *ys*

Lemma 6.24.
⟦*is_cval ys*; *is_cval x*; *is_cval xs*⟧ $\Longrightarrow$ *concat* $$ [*x* ## *xs*, *ys*] $\rightarrow^4$ *x* ## (*concat* $$ [*xs*, *ys*])

The only notable difference to *rev'* here is that the reduction result for the composite list does not contain an application of *rev'* again but rather a list constructor in its outermost position.

Similarly to Lemma 6.9 we have:

Lemma 6.25.
⟦*concat* $$ [*t1.0*, *t2.0*] $\rightarrow^k$ *v*; *is_list t1.0*; |*t1.0*| = *1*; *is_cval t2.0*; *is_cval v*⟧ $\Longrightarrow$ *t1.0* = *nil*

We introduce the abbreviating notations C_{concat} for *C rev' [is_cval, is_list]* and R_{concat} for *reducible rev' [is_cval, is_list]*. Note here that the types are swapped compared to *rev'*, as the argument that the recursion is following is the first one for *concat* and not the second one.

The analogy also extends to the upper bound result for the base case:

Lemma 6.26. *C_concat [1, n] ≤ 3*

Clearly, the finiteness lemma for the C_{set} corresponding to *concat* is proved as easily with the help of Corollary 5.26 as for rev':

Lemma 6.27. *finite (C_set concat [is_list, is_cval] [m, n])*

This enables us to prove an upper bound for the complexity of *concat* in the case where the first argument is of size greater than one.

Lemma 6.28.
⟦1 < m; R_concat [m, n]⟧ ⟹ ∃ m'<m. C_concat [m, n] ≤ 4 + C_concat [m', n] ∧ R_concat [m', n]

Proof. As the reduction behavior for the composite case differs between *concat* and *rev'*, this is the lemma in whose proof a small derivation from the analog for *rev'* has to be expected (this also reflected by the fact that the second size bound does not change here). We display the Isar proof script in Figure 6.4 for comparison with the proof of Lemma 6.15 (cf. Figure 6.1) and explicate their discrepancies. The crucial difference is that we have to argue here that the property *Concat.concat $$ [t1, t2] →⁴ x ## (Concat.concat $$ [xs, t2])* implies that there also has to be some list xs', to which *Concat.concat $$ [xs, t2]* can be reduced in $M - 4$ steps. Only then we can bound *R_concat [m, n]* with a term involving C_{concat} for a smaller size bound for the first argument, because only then can we find some C_{set} for *concat* in which $M - 4$ is contained. The remaining part of the proof then again resembles the proof of Lemma 6.15. □

Once again, we eliminate the reducibility assumption.

Lemma 6.29. *1 < m ⟹ ∃ m'<m. C_concat [m, n] ≤ 4 + C_concat [m', n]*

Thus, we arrive at the final theorem about the complexity of *concat*.

Theorem 6.30. *C_concat [m, n] ≤ 4 * m*

Proof. For completeness we give the Isar proof script in Figure 6.5 and note that one proof step is saved here compared to the proof of Theorem 6.18 in the induction step, because fortunately the *force* method can already successfully prove the whole proof obligation automatically. □

6.4. Complexity Analysis for *reverse*

The proof for the result about the upper bound for the complexity of *rev* (cf. Theorem 6.21) already gave a glimpse into how results for the complexity of subroutines can be reused to derive complexity upper bounds for larger programs that rely on these subroutines. As such a procedure will be elementary to most practical complexity analysis, we want to

```
lemma C_step_concat:
 m > 1 ⟹ R_concat [m,n] ⟹
 ∃ m'. m' < m ∧ C_concat [m,n] ≤ 4 + C_concat [m',n] ∧ R_concat [m',n]
proof −
 assume m > 1 R_concat [m,n]
 def M ≡ C_concat [m,n]
 obtain ts :: expr list and v where ts:
 length ts = 2   valid_terms [is_list, is_cval] [m,n] ts   concat $$ ts →^M v   is_cval v
 using fin_red_C_elem_C_set[OF C_set_finite' ‹R_concat [m,n]›]
 by (auto simp: eval_nat_numeral M_def)
 then obtain t1 t2 where ts2:ts = [t1,t2] by (blast dest: list_length_2_D)
 with ts have t1t2: is_list t1   is_cval t2   |t1| = m   |t2| = n
 by (force simp: valid_terms_def)+
 from ‹is_list t1› show ?thesis
 proof (cases rule: is_list.cases)
  case Nil thus ?thesis using ‹m > 1› t1t2 by auto
 next
  case (Cons x xs)
  with concat_red_step ‹is_cval t2› have concat $$ [t1,t2] →^4 x##(concat $$ [xs, t2]) by auto
  from n_steps_sub_final[OF _ this cval_final] ts ts2 have x##(concat $$ [xs,t2]) →^(M − 4) v
  by auto
  from cons_val_red[OF this cval_final[OF ‹is_cval x›]] ts ts2 obtain xs' where
  concat $$ [xs,t2] →^(M − 4) xs'   is_cval xs' by blast
  with ‹is_list xs› ‹is_cval t2› ‹|t2| = n› ‹is_cval xs'›
  have ∗: M − 4 ∈ C_set concat [is_list, is_cval] [ |xs|, n]
   apply (intro C_set_I)
   apply assumption
   apply auto
  done
  from C_ge[OF C_set_finite' this] ‹m > 1› have C_concat [m, n] ≤ 4 + C_concat [ |xs|, n]
  by (auto simp: M_def)
  moreover have 0 < |xs|   |xs| < m using Cons(1) ‹|t1| = m› by auto
  moreover from ∗ have R_concat [ |xs|, n] unfolding reducible_def by blast
  ultimately show ?thesis by auto
 qed
qed
```

Figure 6.4.: The Isar proof script for Lemma 6.28

```
theorem concat_linear_complexity:
 C_concat [m, n] ≤ 4 * m
proof (induction m rule: less_induct)
 case (less m)
 show ?case
 proof (cases m ≤ 1)
  case True
  thus ?thesis using complexity_0[OF _ C_set_finite'] C_concat_base_case[of n] by (cases m) auto
 next
  case False
  with C_step_le[of m n] less show ?thesis by force
 qed
qed
```

Figure 6.5.: The Isar proof script for Theorem 6.30

conclude this chapter by demonstrating how such a technique can be employed to show a quadratic complexity upper bound for another, more naive version of the list reversal program that relies on the list concatenation program from the last section as a subroutine. The program definition is straightforward:

Definition 6.31.
reverse ≡
 μ ''reverse''. FN ''xs'' ⇒
  MATCH V ''xs'' WITH
   ''Nil'' ≫ [] ⇒ *nil* ‖
   ''x'' ## ''xs'' ⇒ *concat* $ (V ''reverse'' $ V ''xs'') $ (V ''x'' ## *nil*)

As before, we describe the reduction behavior in the cases of empty and composite lists.

Lemma 6.32. *reverse* $$ [*nil*] $\rightarrow^2$ *nil*

Lemma 6.33.
⟦*is_cval x*; *is_cval xs*⟧
$\Longrightarrow$ *reverse* $ (*x* ## *xs*) $\rightarrow^{Suc\ (Suc\ (Suc\ 0))}$ *concat* $ (*reverse* $ *xs*) $ (*x* ## *nil*)

As the *reverse* program relies on the idea of reversing a list's tail and appending the head of the list to the result, a lemma that captures the essence of this behavior will come in handy for an analysis of the program's complexity:

Lemma 6.34.
⟦*reverse* $ *xs* $\rightarrow^n$ *ys*; *is_cval x*; *is_cval xs*⟧
$\Longrightarrow$ *reverse* $ (*x* ## *xs*) $\rightarrow^{3\,+\,n}$ *concat* $ *ys* $ (*x* ## *nil*)

Proof. The lemma follows directly from Lemma 6.33 and the reduction properties of the n small-step relation as mentioned in Chapter 4. □

Analogous to the preceding analyses we prepare

Lemma 6.35. $[\![\mathit{reverse}\ \$\$\ [t] \to^k v; \mathit{is\_cval}\ t; |t| = 1; \mathit{is\_cval}\ v]\!] \Longrightarrow t = \mathit{nil}$

and show an upper bound for the base case:

Lemma 6.36. $C\_\mathit{reverse}\ [1] = 2$

We abbreviate C *reverse* [*is_list*] as $C_{reverse}$ and *reducible reverse* [*is_list*] as $R_{reverse}$ and restate the finiteness property of the C_{set} corresponding to *reverse*.

Lemma 6.37. $\mathit{finite}\ (C\_\mathit{set}\ \mathit{rev'}\ [\mathit{is\_cval}, \mathit{is\_list}]\ [m, n])$

The proofs for the last lemmas are completely analogous to the respective proofs for *rev'* and have therefore been skipped. Unfortunately, the proof of an upper bound for the complexity of *reverse* in the case of composite list arguments does not transfer as nicely from the analysis of *rev'*. In preparation, we need to prove additional *semantic* properties about *concat* and *reverse*. Firstly, given lists as inputs, *concat* outputs a list whose size is smaller by one than the sum of the sizes of the inputs (the uncommon lemma statement produces a rule convenient for use by Isabelle proof methods):

Lemma 6.38.
$[\![\mathit{is\_list}\ xs; \mathit{is\_list}\ ys;$
$\bigwedge n\ zs.\ [\![\mathit{concat}\ \$\$\ [xs, ys] \to^n zs; \mathit{is\_list}\ zs; |xs| + |ys| = \mathit{Suc}\ |zs|]\!] \Longrightarrow \mathit{thesis}$
$]\!] \Longrightarrow \mathit{thesis}$

Proof. The proof employs a rule induction on the *is_list* predicate's rules for xs. The case of the empty list follows trivially from Lemma 6.23. For the case of a composite list $x\#\#xs$ we can first reduce *concat* \$\$ [*x ## xs*, *ys*] to *x ##* (*concat* \$\$ [*xs*, *ys*]) in four steps using Lemma 6.24. We discharge the induction hypothesis to obtain a list zs with $|xs| + |ys| = Suc\ |zs|$, such that *concat* \$\$ [*xs*, *ys*] can be reduced to zs in n steps. We now can reduce *concat* \$\$ [*x ## xs*, *ys*] to $x\#\#zs$ in $n + 4$ steps and the equation $|x\#\#xs| + |ys| = Suc\ |x\#\#zs|$ is derived easily. □

We use this to show that *reverse* produces a list of equal size from an input list.

Lemma 6.39.
$[\![\mathit{is\_list}\ xs; \bigwedge n\ ys.\ [\![\mathit{reverse}\ \$\$\ [xs] \to^n ys; \mathit{is\_list}\ ys; |xs| = |ys|]\!] \Longrightarrow \mathit{thesis}]\!] \Longrightarrow \mathit{thesis}$

Proof. Again, the proof uses a rule induction on the *is_list* predicate's rules for xs and the base case also follows trivially from Lemma 6.32. For the case of a composite list $x\#\#xs$ we can first reduce *reverse* \$\$ [*x ## xs*] to *concat* \$ (*reverse* \$ *xs*) \$ (*x ## nil*). We discharge the induction hypothesis to obtain a list ys with $|xs| = |ys|$, such that *reverse* \$\$ [*xs*] can be reduced to ys in n steps. Now we use the last result to find a list zs with $|xs| + |x\#\#nil| = Suc\ |zs|$, such that *concat* \$ *ys* \$ (*x ## nil*) can be reduced to zs. We get $|xs| + |x| = |zs|$ and from the reduction properties of the $n$small-step relation we can conclude that *reverse* \$\$ [*x ## xs*] can be reduced to zs. □

These lemmas are the only additional instruments needed to derive a complexity upper bound for *reverse* in the case of a composite input list.

Lemma 6.40.
$[\![1 < n; R\_rev'\ [m, n]]\!] \Longrightarrow \exists\, m'\, n'.\ n' < n \wedge C\_rev'\ [m, n] \leq 4 + C\_rev'\ [m', n'] \wedge R\_rev'\ [m', n']$

Proof. The general layout of the proof is similar to lemmas 6.15 and 6.28. The Isar proof script is given in Figure 6.6. We define M as *C_reverse* $[n]$, obtain a list t of size n and a constructor value v such that *reverse* \$\$ $[t] \rightarrow^M v$ holds, and employ a case analysis on the list property of t. The case of the empty list is trivial. For the composite case we suppose $t = x \#\# xs$ for a constructor value x and a list xs. We use Lemma 6.33 to get *reverse* \$\$ $[t] \rightarrow^3$ *concat* \$\$ [*reverse* \$ *xs*, *x* ## *nil*] and use Lemma 6.39 to find a number of steps k and a list ys of the same size as xs such that *reverse* \$\$ $[xs] \rightarrow^k$ *ys* holds. Then we know $k \in$ *C_set reverse* [*is_list*] [|*xs*|] and thus $k \leq$ *C_reverse* [|*xs*|]. Lemma 6.34 gives us *reverse* \$ (*x* ## *xs*) $\rightarrow^{3+k}$ *concat* \$\$ [*ys*, *x* ## *nil*]. With the help of Lemma 6.38 the last reduction result can be reduced to some list zs in a number of steps l, implying that $l \in$ *C_set concat* [*is_list*, *is_cval*] [|*ys*|, |*x* ## *nil*|] and the inequality $l \leq$ *C_concat* [|*ys*|, |*x* ## *nil*|] hold. This way we have bounded l by some complexity expression for which we have already found another useful bound given by Theorem 6.30 and thus we get $l \leq 4 * (|ys|)$. We can put our previous findings together to get *reverse* \$\$ [*x* ## *xs*] $\rightarrow^{3+k+l}$ *zs*. Hence, from the determinism properties of the $n$small-step relation, we also know $v = zs$ and $M = 3+k+l$ and therefore get an upper bound for M: $M \leq 3 +$ *C_reverse* [|*xs*|] $+ 4 * (|xs|)$. This yields the thesis by final argumentation analogous to the previous proofs. □

Once again, the reducibility assumption can be eliminated.

Lemma 6.41. $1 < n \Longrightarrow \exists\, m'\, n'.\ n' < n \wedge C\_rev'\ [m, n] \leq 4 + C\_rev'\ [m', n']$

Ultimately, we can solve the recurrences to arrive at the final upper bound theorem.

Theorem 6.42. *C_reverse* $[n] \leq 3 * n + 4 * n * n$

Proof. The general proof idea is analogous to the previous results, but the induction step requires a few easy arithmetic approximations that cannot automatically be applied very well by Isabelle, but rather have to be manually expiated in the corresponding Isar proof script (cf. Figure 6.7). Individual proof steps are solved by applications of the first-order proof method *metis* for which the particular rule sets have been found by a first-order proof search with the *sledgehammer* tool [3]. □

```
lemma C_step_reverse:
 n > 1 ⟹ R_reverse [n] ⟹
 ∃ n'. n' < n ∧ C_reverse [n] ≤ 3 + C_reverse [n'] + 4*n' ∧ R_reverse [n']
proof −
 assume n > 1 R_reverse [n]
 def M ≡ C_reverse [n]
 obtain ts :: expr list and v where ts:
 length ts = 1   reverse $$ ts →^M v   is_cval v   valid_terms [is_list] [n] ts
 using fin_red_C_elem_C_set[OF C_set_finite' ‹R_reverse [n]›]
 by (auto simp: eval_nat_numeral M_def)
 then obtain t where ts = [t] by (blast dest: list_length_1_D)
 with ts have t: is_list t   |t| = n by (force simp: valid_terms_def)+
 from ‹is_list t› show ?thesis
 proof (cases rule: is_list.cases)
  case Nil thus ?thesis using ‹n > 1› t by auto
 next
  case (Cons x xs)
  hence reverse $$ [t] →^3 concat $$ [reverse $ xs, x ## nil]
  using reverse_red_cons ‹is_list t› by (auto simp add: eval_nat_numeral)
  obtain ys k where red1:reverse $$ [xs] →^k ys   is_list ys   |xs| = |ys|
  by (auto intro: reverse_list_red[OF ‹is_list xs›])
  have ∗:k ∈ C_set reverse [is_list] [ |xs| ]
  by (rule C_set_I) (fastforce simp: ‹is_list xs› ‹is_list ys› intro: red1)+
  from C_ge[OF C_set_finite' this] have le1:k ≤ C_reverse [ |xs| ] .
  have red2:reverse $ (x ## xs) →^3 + k concat $$ [ys, x ## nil]
  using reverse_cont[OF _ ‹is_cval x›] ‹is_list xs› red1(1) by auto
  obtain zs l where red3: concat $$ [ys, x ## nil] →^l zs   is_list zs
  by (blast intro: concat_list_red[OF ‹is_list ys›] ‹is_cval x›)
  from ‹is_list zs› red3 ‹is_list ys› ‹is_cval x›
  have l ∈ C_set concat [is_list, is_cval] [ |ys|, |x ## nil| ]
   apply (intro C_set_I)
   apply fastforce+
  done
  from C_ge[OF Concat.C_set_finite' this] have l ≤ C_concat [ |ys|, |x ## nil| ] .
  hence le2:l ≤ 4 ∗ ( |ys| ) using concat_linear_complexity[of |ys| |x ## nil|] by auto
  have reverse $$ [x##xs] →^3 + k + l zs using red2 red3 by (subst relpowp_add) fastforce
  from ts(2,3) ‹ts = [t]› Cons(1) ‹is_list zs› final_red_unique1[OF this] have 3+k+l = M
  by (auto simp: cval_final)
  hence M ≤ 3 + C_reverse [ |xs| ] + 4 ∗ ( |xs| ) using le1 le2 by (auto simp: ‹|xs| = |ys|›)
  moreover have 0 < |xs|   |xs| < n using Cons(1) ‹|t| = n› by auto
  moreover from ∗ have R_reverse [ |xs| ] unfolding reducible_def by blast
  ultimately show ?thesis by (auto simp: M_def)
 qed
qed
```

Figure 6.6.: The Isar proof script corresponding to Lemma 6.40

```
lemma reverse_quadratic_complexity:
 C_reverse [n] ≤ 3 * n + 4 * n * n
proof (induction n rule: less_induct)
 case (less n)
 show ?case
 proof (cases n ≤ 1)
  case False
  with C_step_le[of n] less obtain n' where *:
  n' < n   C_reverse [n] ≤ 3 + C_reverse [n'] + 4*n' by auto
  with less.IH less.prems have C_reverse [n'] ≤ 3 * n' + 4 * n' * n' by blast
  with *(2) have C_reverse [n] ≤ 3 * (Suc n') + 4 * n' * (Suc n') by auto
  also have ... ≤ 3 * (Suc n') + 4 * (Suc n') * (Suc n')
  by (metis Suc3_eq_add_3 Suc_leD Suc_le_mono add_mult_distrib2 le_refl mult_le_mono1
    nat_mult_commute)
  also from ‹n' < n› have ... ≤ 3 * n + 4 * (Suc n') * n
  by (metis add_le_mono less_eq_Suc_le mult_le_mono2)
  also from ‹n' < n› have ... ≤ 3 * n + 4 * n * n
  by (metis Suc3_eq_add_3 Suc_le_mono add_mult_distrib less_eq_Suc_le
    mult_le_cancel2 nat_mult_le_cancel_disj)
  finally show ?thesis by auto
 next
  case True thus ?thesis using complexity_0[OF _ C_set_finite'] C_reverse_base_case less
  by (cases n) auto
 qed
qed
```

Figure 6.7.: The structured proof for Theorem 6.42

Part III.

Discussion and Conclusion

7. Discussion and Related Work

This chapter gives an overview of previous efforts related to this work and discusses how they compare to our approach. The first section presents some related approaches to define the semantics of a functional language. Afterwards, different attempts to *automatically* derive the runtime complexity of programs in functional languages are surveyed. Finally, we briefly mention previous work that tried to formalize computability theory. We want to restate that the ultimate goal of this work is actually to combine both – the formalization of a functional language together with a formalization of tools to automatically derive the (worst case) runtime complexity of programs in this language. To the best of the author's knowledge such an undertaking has yet to be completed successfully.

7.1. Comparison With Other Language and Semantics Definitions

When we defined **PMλ** and its semantics in Chapter 3, we already briefly discussed popular alternatives we could have chosen to define the semantics. We now give an overview of some previous approaches that have been made to formalize (with the help of a proof assistant) the definition of programming languages and their *operational* semantics. We note that, due to the simplicity of our language, all of these approaches in a way subsume our efforts (excluding pattern matching for some of them) such that our efforts here are not actually a new research contribution.

The definition of an operational big-step semantics for Standard ML (SML) – *The Definition of Standard ML* by Milner et al. [24] – is one of the earliest attempts to define a formalized (in the sense of formal mathematical prose) semantics for a large-scale, real-world functional language. Many efforts to mechanize a semantics for SML with the help of a proof assistant have tried to follow this specification but failed due to various reasons as described by Owens [29], including the bugs of this specification in its original [13] and its revised version [34]. Intensive work has been done to formalize the semantics of large subsets of the imperative languages Java [16] and C [27]. The scope of this work, however, has been constrained to analyze the algorithmic complexity of programs in a functional ML-style language; thus these approaches are not of greater interest for this work.

Two successful attempts have been made to formalize the semantics of a realistic ML-style language and to proof the type soundness of the language based on this formalization. The subject of study of the first approach [19] is SML. An internal language is used, into which the full SML language can be translated. However, such a translation would not fit the primary scope of our work, for the translation would introduce algorithmic overhead. Moreover, this approach mainly concentrates on the accurate modeling of SML's module system and on the verification of the type safety of the internal language. Again, both lie outside of the main theme of this work. Finally, the formalization with the Twelf proof assistant necessitated the introduction of evaluation contexts, which we

decided to abstain from for reasons explained in Chapter 3 (following the argumentation of Owens [29]).

The second approach is a direct formalization of an operational small-step semantics for $\text{OCaml}_{\text{light}}$, a significant subset of OCaml, by Owens [29]. The author has looked into this approach at the beginning stages of his work, but has found the size of the formalization too overwhelming for use in the present work. Instead, the author has decided to keep the definition of **PMλ** as simple as possible but to add a certain ML-style character of the language by adding pattern matching. Among the attempts to define a formalized operational semantics for a realistic programming language, the definition of our operational (small-step) semantics is closest to Owens's work as he also defines a small-step operational semantics for a ML-style language and employs textual substitution for the definition of these semantics. For a discussion of further approaches to formalize the definition of ML-style languages, we refer to Owens [29].

Apart from the ML universe there have been attempts to formalize the semantics for Lisp-style languages and the CRWL logic. Matthews and Findler [23] have defined an operational semantics for the core of Scheme with the help of PLT Redex, a domain-specific language for the specification of operational semantics. They employ Felleisen Hieb-style reduction contexts, whose usage we have decided to avoid as discussed above. Using *PLT Redex*, they have been able to build a test suite for their specification, but they do not provide a proof-assistant formalization of their proof. Additionally, their work concentrates on modeling special features of Scheme which are irrelevant for our work, such as *call/cc* and *dynamic-wind*.

Just as we did, Fraguas et al. [8] also used Isabelle to formalize the semantic framework CRWL. Besides the fact that they use Isabelle, their work does not have any important relation to our semantics formalization as they aim at languages that support far more advanced features than the basic functionality of ML-style languages we aim for. Moreover, the CRWL semantics internalize reduction contexts which would naturally force us to use these.

Lösch and Pitts [22] defined the language PNA (Programming with Name Abstractions) together with an operational semantics as an extension to Plotkin's PCF language [32] with names. Our language is closer to PCF than to PNA because our language just adds variable names to reference names bound under λ-abstractions, compared to PNA, where variable names are first class citizens adding some sort of meta-programming facility to the language. Our pattern matching mechanism also does not have any parallel in PNA. Still, for the definition of our programming language semantics, we most closely follow the approach taken by Lösch and Pitts.

Summarizing, intensive work has been done to formalize the operational semantics of (nearly) whole real-world functional languages, although the involved complexity (and mere size of the formalizations) can reach a high degree of sophistication. Therefore, we have chosen to base our work on a language that is among the most simplistic functional languages to keep the formalization's size to a bearable level. However, the wealth of successful formalizations available shows that the basic approach of our work could in principle be extended to a more feature-rich language; albeit this could drastically increase the complexity of the formalization.

7.2. Complexity Analysis for Functional Languages

In the first part of this section we want to briefly discuss some older approaches to automatically deriving the (mostly average) runtime complexity of programs in functional languages. The field was pioneered by Wegbreit [37] with his *Metric* system. It is able to derive the mean and variance of the runtime for a first-order subset of *Lisp* programs together with an upper and a lower bound on the runtime through statistical analysis. The user has to provide conditional distributions for tests in conditionals and statistical independence of these tests is assumed, which poses a major limitation as the user has to assure the soundness of the analysis through the system.

The *ACE* system [18] is able to derive the worst-case complexity for a subset of *FP* programs. Programs are transformed from an initial program to a function describing its complexity, which is then transformed to a non-recursive expression. The last step makes use of a library of predefined recursive definitions. If a non-recursive version is successfully obtained, it can be further transformed into a version that consists only of canonical functions such as logarithms or polynomials. Through the elimination of the assumption of statistical independence, the *ACE* system is applicable to a substantially larger set of programs than *Metric*.

A similar approach has been taken by Rosendahl [33]. His system is able to automatically derive an upper bound of the complexity of a first-order subset of *Lisp* programs. An initial transformation step derives from a program a so-called *step-counting* version, on which then an abstract interpretation is performed to arrive at a program that gives an upper bound of the program's execution time. Rosendahl presents non-mechanized proofs for the correctness of the system.

All three systems share the property of requiring input programs in a closed form, which prohibits modularity of the analysis. Numerous even more advanced systems exist whose basic approaches are similar to the *Metric* system [7, 40]. For all of the mentioned approaches it is unclear how they could be integrated into our framework as it is non-obvious how a formal verification of their correctness could be approached.

A different approach settles on automatic amortized resource analysis [10, 11]. Hoffmann et al. [10] studied a first-order fragment of Ocaml, Resource Aware ML (RAML), to obtain multivariate polynomial bounds for functions with several arguments. Their approach is not verified and it is not clear how their approach could be integrated into our framework, as complexity bounds for RAML programs are generated at compilation time.

We have found three more recent attempts that tried to automatically infer recurrences describing the upper bound of a functional program's complexity and also tried to automatically resolve these recurrences. Of course, this adds up a greater degree of sophistication compared to our approach, as in our framework both steps have to be executed manually. Nonetheless, none of these approaches have been formalized – and as the recurrence solving steps rely on computer algebra systems, this also seems to be an unrealistic goal which will not be fully achieved in the near future. Anyhow, strong evidence for the soundness for all of the approaches exists. The following gives a short summary and comparison of these approaches.

Benzinger [2] presented a system to automatically analyze the complexity of programs synthesized from proofs (so-called *proof extracts*) in the Nuprl proof development system. His system is capable of handling programs with primitive recursion, first-order func-

tions, and a subclass of higher-order functions. The programs are limited to operating on integers and (lazy) lists. Benzinger introduced a calculus that enables one to define abstract functions and lists, in which the concrete computation of subterms is abstracted by terms describing only the complexity of the subterm computation; this yields support for higher-order functions (by replacing them with first-order complexity descriptions) and built-in lists. His approach consists of two stages. First, using abstract interpretation, a symbolic evaluator simultaneously produces recurrence equations and a raw complexity expression. These are then simplified in the next step by repeatedly calling a recurrence solving subprocedure, which in part relies on the computer algebra system (CAS) *Mathematica*, to replace recursive terms with closed ones. The final output of the procedure is either a closed complexity expression or a partially simplified expression for which unresolved recurrences are reported. Benzinger states that the answer is not provably correct by means of automatic theorem proving, although he claims that this defect could be resolved in the future by adapting the system such that it produces a verifiable proof for the program complexity along the way.

In principle, a symbolic evaluator similar to Benzinger's could also be added to our framework, provided that the soundness of the evaluator is also proved. However, considering that Benzinger's system is already made up of over 3000 lines of code, such an undertaking would most likely require great formalization efforts. As Isabelle and other proof assistants currently provide no means for a CAS integration, it is clear that at least in the near future the recurrences will still have to be resolved by manual proof efforts.

A simpler possible extension to our current framework that would use a similar idea to Benzinger's symbolic evaluator can be imagined: a simple (partial) evaluation function could be defined that maps any expression to a pair of a reduced expression and the number of reduction steps required to arrive at the reduced expression. If the soundness of the function was proved, it could then be used to replace those proof steps that currently require the explicit statement of such reductions. The quality of the function would crucially determine the amount of simplification gained by this approach: on the one hand the function's range should be as large as possible, but on the other hand its reduction results should also be as "interesting" as possible, e.g. it should preferably reduce the expression to a point where the recursive definition of a function has been substituted once. Due to time constraints, the author has not managed to fully evaluate the feasibility of such an approach, but during initial experimentation he found it hard to reach a satisfying result for the second aspect while keeping the complexity of the soundness proof down to a bearable level.

A related approach that extends Benzinger's work in part was taken by Jouannaud and Xu [12]. Except that they study the complexity of programs extracted from proofs developed with the *Coq* proof assistant rather than the *Nuprl* proof assistant, their approach is in its main ideas similar to Benzinger's, although it is applicable to a greater number of programs. However, the main limitations remain: only the complexity of programs working on lists and natural numbers that use at most one recursive call can be automatically calculated. A soundness or completeness proof has not been attempted. From the similarities of the two approaches it follows that they should also be equally portable to our framework.

An entirely different approach for the automatic inference of equations describing the (worst-case) runtime complexity of functional programs was taken by Vasconcelos and

Hammond [36]. They used a type-based analysis to automatically infer first-order cost equations for a simple functional language that is closely related to ours; the main difference is that their language does not provide pattern matching, but rather provides elementary datatypes for booleans, natural numbers and lists together with corresponding primitive operations. Similarly to Benzinger's approach, a CAS must be used to obtain closed-form solutions from these equations. The analysis is based on an effect system using sized types as an extension to the Hindley-Milner polymorphic type system. This systems adds a subscript to most types which specifies an upper bound for its size. Function types carry a subscript that describes an upper bound for the evaluation cost of the function body. A type reconstruction algorithm is employed to derive for an expression in the language a cost effect, a sized type and corresponding recurrence equations. The cost model differs slightly from ours in that Vasconcelos and Hammond define the equivalent to our runtime notion only through the number of β-reduction steps appearing along a reduction sequence. However, for asymptotic analyses this difference looses any significance.

As Benzinger, Vasconcelos and Hammond have not made any attempts to formally verify the soundness or completeness of their approaches, although they claim that their results would naturally carry over from known proofs for other type and effect systems. The types of programs supported by the analysis of Vasconcelos and Hammond are similar to that of Benzinger's approaches, despite that their approach does not work for all primitive recursive programs. Due to the similarity of the language studied by Vasconcelos and Hammond, and **PMλ**, it seems reasonable to argue that their approach could also be translated into an extension for our framework. However, to retain a formalization for the whole framework, a soundness proof for the type and effect system and the type reconstruction algorithm would have to be formalized, which would again be a project requiring substantial effort.

7.3. Formalizations for Computational Theory

We briefly mention previous efforts to formalize computational theory with the help of proof assistants. None of these approaches have been used for complexity analysis – due to their intricacy they also do not seem to be easily applicable to this kind of analysis. Turing machines have been formalized with Isabelle [39] and the Matita theorem prover [1]. The former work features a short (1500 lines of code) formalization of Turing machines. However, the translations required to construct universal Turing machines encompass well over 10000 lines of code. As these comparatively simple forms of compilation require such large formalizations, it is reasonable to argue that the formalization of a compiler that translates programs in even a simple language to Turing machines would be a quite onerous project. However, such a compiler would be necessary to reason about program complexity with Turing machines. Instead of Turing Machines, a different computational model was used by Norrish [28]. He mechanized a proof for the computational equivalence of λ-calculus and recursive functions. This work is highly non-trivial so it is also undesirable to use these computation models for formally verified complexity analysis.

8. Conclusion

We constructed a framework for formally verified runtime complexity analysis of functional programs in Isabelle/HOL. Afterwards we showed how it can be used in an end-user setting by demonstrating how to analyze the worst-case of complexity of three concrete example programs. The process involved the definition of the deeply-embedded programming language **PMλ**. A corresponding small-step operational semantics assigned a meaning to the language. This semantics was then used to build a notion of runtime for the language by simply defining runtime for an **PMλ**-expression *e* as the number of small-steps it takes to arrive from *e* at a normal form. We showed that the phrase "the number of small-steps" conveys the correct meaning, as the determinism of the small-step semantics tells us that there will be always at most one such number.

From this simple runtime measure, we derived a runtime complexity measure for terminating functions applied to the correct number of arguments, i.e. we derived a complexity measure for *closed* programs. The definition imposes further restrictions on the closed programs on top of the fundamental idea that complexity analysis looks at functions with arguments that are bounded in size. Each of the functions' arguments is restricted to a certain type described by an HOL predicate and the reduction result of programs is expected to be a constructor value. The former ensures a property that would be naturally fulfilled by a more complex language that provides a type system, while the latter simplifies the complexity analysis by constraining us to "sensible" computations. We discussed previously that an extension to general normal forms as reduction results should be possible without investing too much effort. However, especially for programs employed in a high-security setting, the restriction would usually have to be postulated anyhow, as it is not desirable that programs can get stuck in a error state. The same argument holds for termination.

In order to complement the framework for complexity analysis, we had to show that programs that are given inputs of bounded size can only exhibit a finite number of execution paths if computation terminates and arrives at a value. This was necessary because the definition of our complexity measure employed the HOL Max-operator, which is only defined on finite and nonempty sets. Finally, we demonstrated how a linear complexity upper bound for a list concatenation and a tail-recursive list reversal program can be derived with our framework. Additionally, we derived a quadratic upper bound for a naive list concatenation based version of list reversal. The analysis of the latter involved the reuse of the result for the complexity of the list concatenation subprocedure. We noticed that the Isar proof scripts for the analyses for all three of the programs share a common structure for the most part. On the one hand, this means that an end user can streamline such proofs in a rote manner. On the other hand, the approach should be generalized further. For instance, this could be done by providing a symbolic evaluation function as discussed in the last section.

The size of our formalizations appears to be feasible for further extension. The core

definitions and properties of the language, of its semantics and of the complexity definition encompass around 550 lines of code (LOCs). The fundamental Isabelle theory file for list programs and the theory files for the analyses for the example programs are each around 200 LOCs. The image is only tarnished by the formalization of Chapter 5: the structured proofs for this intuitively simple theorem require 1000 LOCs.

8.1. Future Work

This work has presented a proof-of-concept for the construction of a framework that could be used for practical, formally verified complexity analysis, rather than as a tool for directly achieving the long-term goal of analyzing the complexity of typical HOL programs. Still, we have taken a small step in the right direction. This final section will suggest areas that would have to be investigated in order to make the framework suffice for the complexity analysis of real-world programs.

Probably the most important alteration to our framework required to achieve this long-term goal would be to make the deeply-embedded language expressive enough that, first, HOL programs could be translated into the language while ensuring that the translation preserves equivalence of the programs; and second that the translation would at least preserve asymptotic complexity bounds. As we cannot directly reason about the structure of HOL terms, it remains an open question how such a translation could be defined and how it could be verified that it adheres to the mentioned properties, while keeping user interaction to a minimum. Moreover, it is not clear how expressive the language should be. However, as we discussed in the last section, previous work has shown that the formalization of the operational semantics for realistic functional languages is possible, although the size of the formalizations can reach quite monstrous levels.

For our approach, any language with a corresponding small-step operational semantics would be sufficient, although its expressive power should be as limited as possible in order to minimize the size of the formalization of its semantics. A more complex language would most likely also come along with a type system of its own. In this case, of course, our complexity definition should be adopted such that it is constrained to well-typed programs rather than providing the "types" via predicates on the inputs. Additionally, if a more realistic language provided a mechanism, such as exception handling, that could yield different types of legal computations, naturally our complexity definition should be reconsidered to account for the different computation results. This list could well be continued for pages, but the important point here is that, in principle, the approach of this work could be extended to far more complex languages, although the formalization would probably balloon. We want to mention that the first-order property of **PMλ** is currently its biggest limitation. Thus, the semantics should be extended to higher-order functions and the complexity of corresponding programs should be analyzed as a first step.

The second step to make our framework applicable to practical applications would be to extend it with means of automatic complexity analysis. We have intensively discussed related previous work in this field and noted that it should in principle be possible to extend our frameworks with these or similar approaches. Again, the issue here is that the formalization of these approaches would require significant effort. We also proposed a more feasible extension: a partial function for symbolic evaluation could be defined that

would try to reduce an input expression as far as possible. It would return the number of reduction steps that are necessary for this reduction together with the resulting expression. If the soundness of this function was proved, it could – depending on its quality – replace a greater part of the argumentation required in our proof scripts. Compared to the other approaches, the formalization of such a function would be quite lightweight.

Finally, minor additions could be made to our formalization. The result for the finiteness of the number of possible reduction paths for programs with inputs of bounded size could be generalized to normal forms rather than values as reduction results. The same holds for the equivalence result between the big-step and small-step semantics.

Bibliography

[1] A. Asperti and W. Ricciotti. Formalizing Turing Machines. In L. Ong and R. d. Queiroz, editors, *Logic, Language, Information and Computation*, number 7456 in Lecture Notes in Computer Science, pages 1–25. Springer Berlin Heidelberg, Jan. 2012.

[2] R. Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(01):3–31, 2001.

[3] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems*, number 6989 in Lecture Notes in Computer Science, pages 12–27. Springer Berlin Heidelberg, Jan. 2011.

[4] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(02):56–68, June 1940.

[5] R. L. Constable, S. F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, S. F. Smith, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with The Nuprl Proof Development System*. 1986.

[6] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, Sept. 1992.

[7] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-Upsilon-Omega: An assistant algorithms analyzer. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, number 357 in Lecture Notes in Computer Science, pages 201–212. Springer Berlin Heidelberg, Jan. 1989.

[8] F. L. Fraguas, S. Merz, and J. R. Hortalá. A formalization of the semantics of functional-logic programming in Isabelle. *Technical Report SIC-4-09*, Aug. 2009.

[9] M. J. C. Gordon. *Edinburgh Lcf: A Mechanised Logic of Computation*. Springer-Verlag, 1979.

[10] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, Nov. 2012.

[11] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 185–197, New York, NY, USA, 2003. ACM.

[12] J.-P. Jouannaud and W. Xu. Automatic complexity analysis for programs extracted from Coq proof. *Electronic Notes in Theoretical Computer Science*, 153(1):35–53, Mar. 2006.

[13] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, Apr. 1993.

[14] G. Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. ACM.

[16] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006.

[17] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[18] D. Le Métayer. ACE: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, Apr. 1988.

[19] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 173–184, New York, NY, USA, 2007. ACM.

[20] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 42–54, New York, NY, USA, 2006. ACM.

[21] X. Leroy. *Functional programming languages. Part I: operational semantics*, 2006. http://pauillac.inria.fr/~xleroy/mpri/2-4/semantics.2up.pdf.

[22] S. Lösch and A. M. Pitts. Full abstraction for nominal Scott domains. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, page 3–14, New York, NY, USA, 2013. ACM.

[23] J. Matthews and R. B. Findler. An operational semantics for Scheme. *J. Funct. Program.*, 18(1):47–86, Jan. 2008.

[24] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[25] T. Nipkow and G. Klein. *Concrete Semantics*. 2014. http://isabelle.in.tum.de/~nipkow/Concrete-Semantics/concrete-semantics.pdf.

[26] T. Nipkow, Lawrence C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. 2002.

[27] M. Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-45, University of Cambridge, 1998.

[28] M. Norrish. Mechanised computability theory. In M. v. Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving*, number 6898 in Lecture Notes in Computer Science, pages 297–311. Springer Berlin Heidelberg, Jan. 2011.

[29] S. Owens. A sound semantics for $\text{OCaml}_{\text{light}}$. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems*, ESOP'08/ETAPS'08, page 1–15, Budapest, Hungary, 2008. Springer-Verlag.

[30] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, Sept. 1989.

[31] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[32] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, Dec. 1977.

[33] M. Rosendahl. Automatic complexity analysis. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 144–156, New York, NY, USA, 1989. ACM.

[34] A. Rossberg. Defects in the revised definition of Standard ML. Technical report, Universität des Saarlandes, 2001.

[35] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[36] P. B. Vasconcelos and K. Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Implementation of Functional Languages*, page 86–101. Springer, 2005.

[37] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, Sept. 1975.

[38] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, L. Théry, A. Hirschowitz, and C. Paulin, editors, *Theorem Proving in Higher Order Logics*, number 1690 in Lecture Notes in Computer Science, pages 167–183. Springer Berlin Heidelberg, Jan. 1999.

[39] J. Xu, X. Zhang, and C. Urban. Mechanising turing machines and computability theory in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, number 7998 in Lecture Notes in Computer Science, pages 147–162. Springer Berlin Heidelberg, Jan. 2013.

[40] P. Zimmermann and W. Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. Technical Report 1149, INRIA Rocquencourt, 1989.